Behavioural State Machines
Agent Programming and Engineering

Doctoral Thesis
(Dissertation)

to be awarded the degree of
Doctor rerum naturalium (Dr. rer. nat.)

submitted by

Peter Novák
from Ružomberok, Slovak Republic

approved by

Faculty of Mathematics/Computer Science and Mechanical Engineering
Clausthal University of Technology

date of oral examination
September 11th 2009

chairperson of the board of examiners
Prof. Dr. rer. nat. Jörg P. Müller

chief reviewer
Prof. Dr. rer. nat. Jürgen Dix

reviewer
Prof. Michael Fisher, Ph.D.
(University of Liverpool, UK)

D104

To Zuzana Bullová, Ján Šefránek and Jürgen Dix,
my teachers of computer science.

Contents

Preface

AIC IXH XAN.

(Jan van Eyck)

> *This is a work of an apprentice in the craft of computer science on his way to becoming a journeyman of the guild. I submit this dissertation text as the proof of the level I reached in the fine craft of Artificial Intelligence research.*

I present here an account of my research work of the last, almost five years. Initially, it was driven by a vision of applying computational logic in cognitive agents. While on one hand, I managed to build systems which indeed apply logic based technologies in practice even in non-trivial and interesting ways, the original project partly missed the vision. Instead of showing an impressive application of computational logic in a cognitive agent, I spent most of the time solving issues and removing roadblocks on the way to enabling such a demonstration. I found that in order to apply any kind of computational logic in a software system, I first need to solve some fundamental software engineering issues on the way. One of such turned out to be an agent programming framework, a language capable of integrating arbitrary KR technologies, which a programmer might choose for her or his application. While the project is certainly nowhere close to its end, I tried to stand up to Jan van Eyck's maxim and did my best to deliver a good solution. I do not only propose an agent programming framework of my taste, but also elaborate on how to use it in the target domain and demonstrate its merit in several non-trivial proof-of-concept applications. Finally, I reach beyond and sketch some of the potential future research avenues stemming from the presented work.

Each of us is partly a product of interactions with our social environment. Retrospectively, we can trace our personal intellectual evolution in the past to people who influenced us, changed our direction in a positive sense, or helped us to avoid dead-end paths. I dedicate this dissertation to my teachers, especially those who gave me the direction towards computer science and artificial intelligence. In particular, this work is dedicated to my high school teacher of programming Zuzana Bullová, my university teacher of artificial intelligence, adviser and a friend Ján Šefránek and finally my teacher of scientific research affairs and supervisor of my doctoral research Jürgen Dix.

I was fortunate to work under supervision of Prof. Jürgen Dix. His attitude to us, his assistants, promotes independent research and leaves space for personal development. At the beginning, Jürgen did not give me a concrete research topic, but rather trusted

me and guided me in the search for my own theme. I am grateful for that and I hope, I wasn't a bad bet for him either.

Nowadays, research work is more and more becoming a collaborative effort. The one presented in this dissertation very much fits this pattern. A large portion of its content is based on joint works with my colleagues and collaborators. In particular, I am glad I could co-author papers with my supervisor Jürgen Dix, my colleagues Wojtek Jamroga, Michael Köster and Tristan Behrens, as well as with Koen Hindriks, Mehdi Dastani and Nick Tinnemeier and the students of our group Bernd Fuhrmann, David Mainzer and Slawomir Dereń. I am grateful to Koen for his constructive critique of my early ideas and later fruitful collaboration. Wojtek helped me to clarify and fix many of my half-backed ideas. I very much enjoyed the friendship arising from the almost five years we spent together in Clausthal and our endless debates on anything spanning research, family, society and beyond. I am glad that I met and could share the office in the last year with Michael, a former diploma student in our group and later a colleague. Even though we share a lot of opinions on how the world should turn around, we never had a lack of topics for lengthy arguments and discussions opposing each other, mostly as a matter of principle. Besides Michael, I was also very lucky to meet students Bernd, David and Slawek, who made a great job working on various projects under my supervision and who helped me to turn the ideas presented here into working applications. Reflecting the nature of the collaborative work, parts of this dissertation are mixing the plural voice with the, otherwise prevalent, first-person narrative.

My ambition in doing research is to contribute to the state of the art at the highest level of excellence I can deliver. Examples of such works as well as invaluable sources of inspiration were in these years for me those by Koen Hindriks and Birna van Riemsdijk.

"Every review is gold dust" (Jones, 2005). I am grateful to all the anonymous reviewers of my papers. Their feedback provided me with constructive critique, fresh views and hints and thus greatly influenced the directions of my research work.

My years in Clausthal were marked by friendships to the family Mišík, Yingqian Zhang, Andreas Brüning, Zuzana Zúberová, Lenka Grygarová as well as colleagues of my wife from University of Göttingen Tanja Dučić, Simon Castorph, Klaus Giewekemeyer and their partners. They made our time in this country as pleasant as it was. Not to forget my colleagues Nils Bulling, Juan Guadarrama, Rebecca Vincent Olufunke and all the guest researchers we hosted, who maintain the family-like atmosphere of Jürgen's group.

Finally, I am indebted to my wife Eva who followed me to Germany and stood by my side all the time in the years we are together. During our stay in Germany, we received a lot of support from both our families. Thanks go to all the four of our parents, our siblings and their families, as well as to our daughter Bronislava, who brightens our days.

Peter Novák
Clausthal-Zellerfeld, July 2009

Chapter 1

Introduction

1.1 Prelude

Ape is an Airport Passenger E-assistant, a mobile humanoid robot with the task of helping passengers by providing them with flight schedules, getting them at the right gate on time and assisting with various travelling trivia. On Monday morning, Ape is busy assisting travellers at the Bratislava airport.

Bronja has a busy day as well. She was visiting her friend Boris in Bratislava over the weekend and now is about to fly back home to Brussels. Because of a traffic jam in the city, she arrived to the airport a bit later than expected and since this is her very first visit to Bratislava, she feels a bit lost. Bronja stands helplessly in the middle of the airport check-in hall. She looks around to find a BBWings airline counter when Ape arises from the crowd and approaches her with a kind offer for assistance. Bronja quickly explains her situation. Ape's internal knowledge base includes information about the BBWings ground facilities, so he explains Bronja the way to the counter B42 in terminal B check-in hall. After recognizing that she has no clue how to get there, Ape offers her guidance to the Base Line inter-terminal rail shuttle station. Bronja gladly follows.

Ape smoothly navigates through the crowd when suddenly a rushing man approaches him and asks for a direction to the newly opened airport Wi-Fi lounge. Ape stops and apologizes to Bronja for the interruption. Since the new Wi-Fi lounge was only opened today, Ape was not updated about it yet. He looks up the airport electronic yellow pages service to find an agent informed about facilities located at the airport. His request matches the profile of Bran0, the Business Register Agent maintaining the airport facilities database. Bran0 of course knows the Wi-Fi lounge whereabouts. It is located just behind the corner from Ape's actual position, right to the Café Superiore kiosk. Ape updates his internal database with the new information and explains the directions to the rushing man.

The interruption did not take more than a minute, yet Ape kindly apologizes to Bronja again. Then he asks her to follow him to the station where he

explains that the BBWings check-in counter B42 is right opposite the Base Line shuttle platform at terminal B. Bronja should be able easily find it after getting off the train. To reassure her that she will manage to check-in on time, Ape notifies Chic, the Check-In Clerk Agent of the BBWings ground personnel serving at the counter B42, about Bronja's later arrival. Right before he says goodbye to Bronja, Ape receives a new task from the airport command center. A group of elderly tourists needs help with check-in for the flight to Bucharest.

1.2 Motivation

The driving vision behind the research leading to this dissertation is that of intelligent *open multi-agent systems*. I.e., communities of coordinating, yet only loosely coupled heterogeneous actors and services. Such systems are a necessary prerequisite to enable scenarios such as the one described in the introductory example in the previous section[1]. They have to be able to accommodate autonomous entities, *agents*, such as *Ape*, *Bran0* and *Chic*, which can enter and leave the community upon their own, or their user's decision. They should also be able to transparently look-up other agents in order to communicate and coordinate with them to possibly reach common goals. Development of such systems however presumes development of individual *intelligent agents*.

One of the original long-term aims of the field of *Artificial Intelligence* (AI) is to build intelligent entities, i.e., such which are able to function and act in a manner similar to human beings. According to the standard definition by Wooldridge (2000), such intelligent agents are assumed to be *autonomous*, *proactive*, *reactive*, as well as *socially able*. In other words, most of the time they should act on their own without intervention of a human user (autonomy). They should proactively seek an action whenever a need of an opportunity arises (proactivity). Since they are embodied in a physical or a virtual environment they must be able to react to events and changes in it (reactivity). And finally, they should be able to communicate, understand, negotiate or even argue and thus coordinate with other similar agents present in the environment (social ability).

During its lifetime, AI research mainly focused on solving various particular subproblems of the original aim. It led to development of a plethora of approaches and practically applicable technologies tackling various aspects of the original goal (for a taste recall the classics by Russell and Norvig (2002)). Yet, the problem of integration of a number of the pieces of the big puzzle remains open and we only rarely encounter approaches for enabling their composition into a single functional unity.

The leitmotif of this dissertation is the problem of *programming cognitive agents*. I.e., such employing cognitive processes as the basis for their decision making and actions. In particular, I focus on systems constructing and maintaining a *mental state* (Shoham,

[1] Adapted and extended from an earlier publication with Dastani et al. (2008d).

1993), a representation of their environment, themselves or their peers, and in turn use knowledge representation and reasoning techniques as the basis for the action selection. To avoid anthropomorphic nomenclature, such systems are also sometimes coined *knowledge-intensive agents* (Jones and Wray III, 2006).

Being able to process symbolic information is a necessary prerequisite for inter-agent information exchange. Reasoning and communication about agent's mental attitudes, such as *beliefs*, *goals*, *intentions*, *commitments*, *obligations*, etc., in turn facilitates coordination among agents. It enables joint reaching of common goals, or engaging in negotiations while at the same time preserving their own autonomy. The agent's mental attitudes might be expressed using various knowledge representation technologies according to the concrete application domain. Any programming framework for such a class of agents must therefore support *programming with mental attitudes*.

In the example scenario, *Ape* needs to maintain an extensive mental state. To be able to respond to the rushing man's query and in turn properly request the information from *Bran0*, his knowledge bases must maintain a topological map of the airport. Similarly, to help *Bronja*, he should know the schedules and routes of the inter-terminal shuttle railway. *Ape* also has to be informed about various behavioural patterns of different groups of travellers at the airport. He should know various social norms corresponding to their cultural backgrounds, so that he is able to recognize *Bronja*'s anxiety, treat her in a convivial manner and proactively take actions to reassure her that everything will be alright. On the other hand he should be unobtrusive when dealing with the group of elderly tourists, while treating business travellers, who are always on a nervous move, as efficiently as possible. *Ape* also holds various desires, descriptions of states and tasks he wants to bring about in the future. While he desires to fulfill various requests of his clients, such as taking *Bronja* to the shuttle station, at the same time he probably also needs to take care of his own technical status. His batteries have to be loaded, he tries to maintain his own safety in, at times quite unfriendly, environment and occasionally he also desires to be checked by the *Airport Android Maintenance Team*.

Besides architectural specification and implementation of various knowledge bases implementing agent's mental state, the software engineering task of building an intelligent agent requires specification of agent's behaviour, i.e., its action selection mechanism. While in order to fulfill longer term goals and tasks the agent should be able to *plan* and subsequently execute possibly complex sequences of actions, at the same time it has to be able to swiftly *react* to events, changes and interruptions occurring in the environment. Efforts in robotics to marry *deliberation* and *planning* on one side with maintaining agent's *reactivity* on the other led to development of various *hybrid architectures*. Yet, an application-domain-independent, robust and flexible specification framework for encoding agent's behaviours in terms of deliberation-based action selection remains one of the open problems of cognitive robotics research.

Ape's deliberation and behaviour selection mechanisms have to be rather complex as well. To resolve *Bronja*'s situation, he needs to construct or retrieve and instantiate a

stock plan from his plan library prescribing how to assist her by providing an advice and navigating to the shuttle through the airport halls. While executing the plan and navigating to the station, *Ape* must quickly react to avoid moving people, luggage transport vehicles, carts and finally even interrupt the navigation in order to serve a high priority request by the rushing man. Actually, in order to fulfill that request, *Ape* has to enact a complex behaviour, a plan, specifying a sequence of actions for looking-up, contacting and negotiating with *Bran0* agent, reasoning about the airport topology and finally explaining the directions to the man. Besides all that, *Ape* must keep track of the lower priority goals in the context of assisting *Bronja* and all the technical maintenance of his own robotic body.

In the research leading to this dissertation I tried to tackle the following challenges:

How to build cognitive agents by integration of various existing AI technologies?

How to encode action selection mechanism in a concise and elaboration tolerant manner?

And once we have a framework tackling the previous two issues, how should programmers use it?

Clearly, these question are those of pragmatic software engineering.

As noted above, while there exists a vast body of research tackling various partial AI subproblems, integration of the various approaches in a single system remains an open problem. The first above formulated question takes on exactly this problem. It asks for a robust solution capable to accommodate heterogeneous approaches to support modelling, reasoning and generally processing information contained in components of agent's mental state.

While constructing, maintaining and using a model of the world is an important aspect of cognitive agent development, after all we require an embodied agent, be it a software or a hardware entity, to act in its environment. Hence, it must execute behaviours satisfying the needs of its user. In the simplest case, when we see an agent as a single discrete computation process, this boils down to an iterated action selection. The second challenge posed above not only asks for a solution enabling encoding of such a behaviour selection mechanism, but since its development is a pragmatic software engineering task, it must support at least the basic requirements imposed on a practical programming framework. In particular, such a framework should lead to preferably short and easily understandable programs, which also allow easy modification and restructuring when a future need arises.

The last but definitely not the least is the problem of a methodology. Provided a toolbox for programming some type of software systems, it is of a high interest to its users, in this case agent developers, to have an insight into its pragmatics. On one hand, tackling this problem requires powerful abstractions providing a suitable optics on solving software engineering problems with the toolbox. On the other, the framework should also come with at least some informal guidelines for the development process in some relevant application domains.

1.3 Agent-oriented programming: state of the art

Since Shoham's seminal paper (1993) on *agent-oriented programming*, one of the high ambitions of the agents programming community is development of a *theoretically founded programming frameworks* enabling construction of cognitive agents. However, a programming framework is an engineering tool in the first place and thus it also has to provide a pragmatic toolbox for development of practical systems. On the other hand, it is desirable to establish a tight relationship of the programming approach with a rigorous machinery for reasoning about programs written with it. The relationship should enable development of a formal methodology supporting the design process, as well as allow a deeper insight into functionality of built agent systems.

Development of programming frameworks for cognitive agents deals with the core issues of two overlapping research fields of *cognitive robotics* and *programming agent systems*. Since the background context of these two topics is around since the AI's inception, the research in the field resulted in a vast body of literature on various aspects of these topics. One of the main streams which can be observed in the state-of-the-art literature is implementation of the *Belief-Desire-Intention* (BDI) agent architecture stemming from the seminal work by Rao and Georgeff (1992). BDI, builds on the classic work on Bratman's planning theory of intention (1999). It heavily relies on anthropomorphic concepts such as beliefs, goals/desires and intentions. In turn, it prescribes architectural decomposition of agent systems into three basic knowledge bases respectively storing agent's model of its environment etc. (beliefs), descriptions of states it wants to bring about (*goals/desires*) and actions of partial plans the agent decided to execute (commitments/intentions). The most recent and relevant threads of research on BDI inspired programming frameworks of the last decade revolve around the *International Conference on Autonomous Agents and Multiagent Systems*, *AAMAS* and two collocated workshops *Programming Multi-Agent Systems*, *ProMAS* and *Declarative Agent Languages and Technologies*, *DALT*. The journal paper by Bordini et al. (2006) and the two volumes by Bordini et al. (2005a) and (2009) provide a comprehensive survey of the state of the art of the field.

1.3.1 Pragmatic approaches

The most prominent state-of-the-art approaches to building agents with mental attitudes can be divided into two main groups. The group of pragmatic engineering approaches includes frameworks, such as *Jadex* (Pokahr et al., 2005) and *JACK* (Winikoff, 2005b). These build on and extend object oriented imperative programming languages. In both cases the underlying programming language is *Java* (Sun Microsystems Inc., 2006; Arnold et al., 2000). These technologies, in particular *Jadex* and *JACK*, provide a layer of BDI agent-oriented programming constructs over *Java* and thus naturally allow programmer to mix native *Java* source code into the agent program. Thanks to this setting, developers can exploit the software engineering features of the underlying object-oriented language such as classes, packages, system calls, etc. Moreover, using the general purpose object oriented programming language makes integration with 3rd party systems and legacy software straightforward and easy to handle.

The programming comfort and the support of plethora of features and tools provided by the underlying language has, however, its price. Because in the end an agent program written with these programming frameworks is translated into plain *Java* code, its underlying semantics is also provided by the underlying language. Theoretical properties of such agent programs can thus be analyzed w.r.t. agent-oriented concepts only to a limited extent and their semantics is rather that of structured imperative programs. Moreover, these languages provide only rather weak techniques for knowledge representation and reasoning in terms of maintaining a set of objects or an object database. Even though using specialized knowledge representation methods can be in principle facilitated by integration of the corresponding KR engine into the *Java* environment, such undertaking would require a major programming effort.

1.3.2 Theoretically founded approaches

The second group of approaches to agent-oriented programming and at the same time more relevant for the scope of this dissertation, are those theoretically founded. To this group belong declarative languages such as *AgentSpeak(L)/Jason* (Rao, 1996; Bordini et al., 2007), *3APL* (Hindriks et al., 1999; Dastani et al., 2005b), *2APL* (Dastani, 2008), *GOAL* (Hindriks, 2001; de Boer et al., 2007), or *MetateM* (Fisher, 1994), to name just the most prominent. As far as the semantic issues are concerned, the situation with these languages is almost reversed in comparison to the previous group of approaches. Their semantics rely on a strong theoretical foundation based on formal computational logic. Thereby, they provide a sound basis for study of properties of resulting agent systems and also enable development of methods for their verification and model-checking.

To bridge the gap between pragmatics of software engineering and theoretical foundations, these BDI-inspired agent programming languages provide a particular set of agent-oriented features and bind them to a rule-based computational model of reactive

planning. However, since they are always built from scratch and the focus is rather on their formal semantics, the language designer's choices on theoretical side impose strong constraints on the resulting design of agent applications. As a result, they suffer from a number of problems which make software engineering with them rather difficult.

First of all, to facilitate a clear and transparent semantics these languages are always tightly coupled with a single knowledge representation technique. Most of the time this is the first-order logic on the theoretical side, what usually translates into employing a flavour of *Prolog*-like language (Shapiro and Sterling, 1994) in the implemented interpreter. However, since *different application domains require different knowledge representation technologies*, the strong bond to computational logic limits the use of non-logic based technologies, which can be sometimes more appropriate for the concrete task in question. This is usually solved by extra-language features, such as calls to *Java* interpreter in the case of *Jason*, *3APL* and *2APL*, for which there is no proper language semantics defined. Moreover, the heavy reliance on a single underlying knowledge representation technology makes the integration with the environment, as well as 3rd party or legacy systems rather difficult, if cared for at all.

Besides a fixed knowledge representation technology, these frameworks come with a fixed set of language constructs for manipulating agent's beliefs and/or goals, and their mutual relationships. The main benefit from constraining agent designers in so many ways is a clear, theoretically sound operational semantics of the language modeled after Plotkin (1981). Traditionally it is provided in terms of computation runs (traces) in a transition system over the agent's mental states. The sound and theoretically clear semantics is the basis for further study of theoretical properties of implemented agents and explorations of techniques for program verification and model checking. However, a tight formal relationship with a rigorous reasoning framework for agent behaviours is only rarely established. Works by Hindriks (2001), de Boer et al. (2007) and Dennis et al. (2007) provide a basis for further studies towards reasoning about and model checking of programs in *GOAL* (de Boer et al., 2007) and *Gwendolen* (Dennis and Farwer, 2008) respectively.

The strong accent on simple semantics and declarative features of these languages results in neglecting their pragmatic aspects. As of now, most of them provide only rudimentary constructs supporting development of application-domain-independent and reusable subprograms. In turn, it is rather difficult to structure and organize larger code bases written in such frameworks. To tackle this issue, recent works of Hindriks (2007), van Riemsdijk et al. (2006b) and Dastani et al. (2008e) introduce concepts of modules and roles in agent programming with *GOAL*, *3APL* and *2APL* respectively.

As of now there are only few reports about problems with applying theoretically founded languages in larger projects. Brief project descriptions can be found in reports from the past editions of the *Multi-Agent Programming Contest* (Dastani et al. 2005a; 2006; 2008a; 2008b), such as e.g., (Hübner and Bordini, 2008; 2008 and Astefanoaei et al., 2008). Yet, these reports do not provide deeper insights into problems and issues

with using the language frameworks. To my knowledge, the only work analyzing a use of the language *Jason* in a larger case-study is the recent report by Madden and Logan (2009). Unlike for the pragmatically oriented counterparts, according to available literature, there seems to be only little experience with the use of such frameworks. It is also important to note that their acceptance outside the research community seems to be rather low. One of the reasons might be the already discussed lack of support for mainstream software engineering techniques.

1.4 Thesis outline and contributions

Part I: Theoretical foundations

This dissertation is divided into three main parts. In the first part, I lay down the main theoretical foundations for exploring the problem of programming cognitive agents. Subsequently, I gradually build up a theory enabling formulation of the main result of this dissertation: a proposal for an alternative approach to design of agent-oriented programming languages presented later in Part II.

Similarly to Hindriks (2001) and van Riemsdijk (2006), I tackle the problem of cognitive agent programming by a formal study of special purpose programming languages for such systems. In Chapter 2, I introduce the framework of *Behavioural State Machines* (*BSM*), the theoretical foundation of this dissertation. It facilitates programming cognitive agent systems by means of synergistic exploitation of *heterogeneous technologies for knowledge representation and reasoning* within a single agent system. At the same time, it also provides a software engineering toolbox for encoding *action selection* mechanism of the agent. Similarly to other theoretically founded languages, the *BSM* approach is a framework built from scratch. Its ambition is, however, to also provide a strong support for development and use of language extensions supporting mainstream software engineering techniques, such as source code modularity and reuse, integration with 3rd party external systems or development of reusable, domain-independent design patterns.

Even though the *BSM* framework does not enforce a particular agent architecture, later in this thesis I focus exclusively on development of BDI inspired agents. Chapter 3 discusses *Modular BDI Architecture*, an instantiation of the *BSM* framework for BDI agents providing the scaffolding for extensions and experimentation later in the thesis.

Chapter 4 introduces *Dynamic CTL** (*DCTL**), a logic for verification and reasoning about *BSM* programs. *DCTL** is a novel extension of the Emerson's full branching time temporal logic *CTL** (1990) with features of *Dynamic Logic* by Harel et al. (1984). In the *BSM* framework, an agent program is encoded in terms of *mental state transformers*, i.e., nested subprograms joined by composition operators. The hierarchical structure of subprograms allows to define and instantiate macros which implement encapsulated and clearly defined agent-oriented concepts, such as e.g., a specific type of a goal. To bridge the gap between the flexible but logic-agnostic programming *BSM* framework and

*DCTL**, the logic for verification of *BSM* programs, I introduce *program annotations* in the form of formulae of *Linear Time Temporal Logic* (*LTL*) (Pnueli, 1977).

Part II: Software engineering issues
The second part of this thesis discusses the pragmatic aspects of programming agent systems with the framework of *Behavioural State Machines*. There, I propose a novel alternative approach to design of agent-oriented programming languages:

> *On the substrate of a generic language for programming reactive systems (BSM), I propose development of a library of agent-oriented design patterns.*

Such encapsulated and application-domain-independent subprograms provide a flexible and easily extensible toolbox for development of cognitive agents. Moreover, code annotations together with the logic associated with the language provide a clear formal agent-oriented semantics to the application code.

To provide substance to the proposal, Chapter 5 introduces *Jazzyk*, an implemented programming language for the *BSM* framework. Apart from the details of the language syntax, I also discuss the implemented language interpreter, its technological background and its features, such as the integrated macro preprocessor as well as considerations about its further development.

The part culminates in Chapter 6 with the introduction of a set of actual code patterns, such as an *achievement* or a *maintenance goal*, facilitating development of BDI style agents. Finally, by lifting a proposed naïve straightforward agent design methodology resulting from using such patterns, I propose *commitment-oriented programming*. A design and programming style based on the idea of specifications of *commitments towards mental attitude*s in terms of agent's behaviours and other commitments. In turn, an agent program can be seen as a specification of web of interrelated commitments.

Jazzyk, the programming language, is a concrete implemented instance of the approach to design of agent programming languages. The discussed code patterns are a first step towards a more extensive library of application-domain-independent agent-oriented constructs.

Part III: Evaluation, extensions and beyond
The final part of the dissertation is dedicated to a description of experimental applications developed with the *Jazzyk* language, discussion of extensions of the plain *BSM* framework, related work, as well as directions for future developments.

Chapter 7 provides an overview of *Jazzbot*, *UrbiBot* and *AgentContest Team*. These are three case-studies developed to demonstrate the feasibility of the proposed programming language, as well as to evaluate its practical applicability and usefulness by employing the methodological guidelines proposed in Chapter 6. *Jazzbot*, the first proof-of-concept demonstration of an agent developed with *Jazzyk*, is a virtual agent roaming in the environment provided by a first-person shooter computer game. The second case-study, *UrbiBot*, is a step towards demonstrating *Jazzyk*'s applicability in building robots

embodied in the physical environment. The code steering *UrbiBot*, a simulated two-wheeled mobile robot, is directly portable to its physical counterpart, *e-Puck* (EPFL, 2006). Finally, *Agent Contest Team* is our planned non-competing entry in *Multi-Agent Programming Contest* (*AgentContest 2009*) and implements a team of agents playing cowboys in a simulated cow herding scenario.

Chapter 8 provides a brief overview of the implemented *Jazzyk* knowledge representation modules used in the previously described experimental applications. Two KR modules *JzASP* and *JzRuby* facilitate respectively knowledge representation and reasoning by means of a non-monotonic reasoning engine for *AnsProlog** (Baral, 2003) and by means of *Ruby*, an object-oriented programming language. Additional three KR modules *JzNexuiz*, *JzURBI* and *JzMASSim* enable interaction with various environments: simulated worlds of the first-person-shooter computer game *Nexuiz* (Nexuiz Team, 2007), robot hardware and mobile robotics simulator *Webots* (Michel, 1998, 2008) and interaction with the *MASSim* server (Behrens et al., 2008, 2009a), the underlying infrastructure of the *AgentContest* series of competitions. Finally, module *JzOAAComm* facilitates inter-agent communication via SRI's *Open Agent Architecture* (Cheyer and Martin, 2001; SRI International, 2007) platform.

The remaining two chapters of the second part discuss complementary results stemming from the proposal of the *BSM* framework. Chapter 9 discusses *Probabilistic Behavioural State Machines*, a probabilistic extension of the plain *BSM* framework allowing a finer grained control of a non-deterministic choice mechanisms of *BSM* interpreters. Finally, Chapter 10 demonstrates embedding of *GOAL* language agent programs in *BSM* framework.

Discussion of the broader context, contributions of the presented work and future research vectors stemming from it concludes the dissertation. Discussion of each of the parts of the dissertation comprises a separate chapter (Chapters 11, 12, 13). The discussion is complemented by an additional Chapter 14, which provides an outlook towards applying the *BSM* framework in multi-agent system scenarios and discusses the first steps to enable such experiments. Finally, Chapter 15 wraps up the dissertation with final remarks.

The main body of the thesis is complemented by extra material in appendices. Appendix A provides technical details and the manual of the *Jazzyk* interpreter implementation. The software development kit for construction of *Jazzyk* plug-ins, KR modules, is briefly described in Appendix B.

Part I

Theoretical foundations

... in which I lay down the theoretical foundations of an agent programming framework drawing a strict distinction between a modular knowledge representation layer and agent's behaviours.

In varietate concordia.

(European Union motto)

Chapter 2

Behavioural State Machines

No single knowledge representation (KR) technology offers a range of capabilities and features required for different application domains and environments agents operate in. For instance, purely declarative KR technologies offer a great power for reasoning about relationships between static aspects of an environment, like e.g., properties of objects. However, they are not suitable for representation of topological, arithmetical or geographical information. Similarly, a relational database is appropriate for representation of large amounts of searchable tuples, but it does not cope well with representing exceptions and default reasoning.

An important pragmatic requirement on a general purpose agent-oriented programming framework is *the ability to integrate heterogeneous KR technologies* within a single agent system. An agent programming framework should not commit to a single KR technology. The choice of an appropriate KR approach should be left to the agent designer and the framework should be *modular* enough to accommodate a large range of KR techniques, while at the same time providing flexible means to encode agent's behaviours.

On the other hand, the dynamics of an environment leads to difficulties with the control of an agent. Unexpected events and changes can interrupt the execution of complex behaviours, or even lead to failure. Therefore an agile agent has to be able to reactively switch its behaviours according to the actual situation. While achievement of long-term goals requires rather algorithmic behaviours, reaction to interruptions has to be immediate. Moreover, due to only partial accessibility of the environment, some situations can be indistinguishable to the agent. It is thus vital to allow reactive nondeterministic choice between several potentially appropriate behaviours, together with arbitration mechanisms for steering the selection.

This chapter introduces the theoretical framework of *Behavioural State Machines* (*BSM*). *BSM* is a general purpose computational model based on Gurevich's *Abstract State Machines* modeled according to the presentation by Börger and Stärk (2003) and adapted to the context of agent-oriented programming. The framework is devised as a glue between a number of heterogeneous knowledge bases of an agent and its interface(s) to the environment. It facilitates assembling agent's behaviours in terms of interactions among the knowledge bases. *BSM* draws a strict distinction between the *knowledge*

representational layer of an agent and its *behavioural layer*. To exploit strengths of various KR technologies, the KR layer is kept abstract and open, so that it is possible to plug in heterogeneous modules as agent's knowledge bases. The main focus of the *BSM* computational model is the highest level of control of an agent: its *behaviours*.

The underlying semantic substrate of the *Behavioural State Machines* framework is that of a transition system over agent's mental states. A mental state is a collection of agent's partial knowledge bases or *KR modules*. The state of the environment is treated as a KR module as well. Transitions between the agent's mental states are induced by *mental state transformers*, atomic updates of mental states. An agent system semantics is then, in operational terms, a set of all enabled paths within the transition system, the agent can traverse during its lifetime. To facilitate modularity and program decomposition, *BSM* provides also a functional view on an agent program, specifying a set of enabled transitions an agent can execute in a given situation.

2.1 Syntax

A *BSM* agent consists of a set of partial knowledge bases handled by *KR modules*. A KR module is supposed to store agent's knowledge e.g., about its environment, itself or other agents, or to handle its internal mental attitudes relevant to keep track of its goals, intentions, obligations, etc. Because of the openness of the *BSM* architecture, no specific structure of an agent system is prescribed and thus the overall number and ascribed purposes of particular KR modules is kept abstract. The formal definitions capture only their fundamental characteristics.

A KR module has to provide a language of query and update formulae and two sets of interfaces: *query* operators for querying the knowledge base and *update* operators to modify it.

Definition 2.1 (KR module). A knowledge representation module $\mathcal{M} = (\mathcal{S}, \mathcal{L}, \mathcal{Q}, \mathcal{U})$ is characterized by

- a set of states $\mathcal{S}$,
- a knowledge representation language $\mathcal{L}$, defined over some domains $\mathcal{D}_1, \ldots, \mathcal{D}_n$ (with $n \geq 0$) and variables over these domains. $\underline{\mathcal{L}} \subseteq \mathcal{L}$ denotes a fragment of $\mathcal{L}$ including only ground formulae, i.e., such that do not include variables,
- a set of query operators $\mathcal{Q}$. A query operator $\models \in \mathcal{Q}$ is a mapping $\models : \mathcal{S} \times \underline{\mathcal{L}} \to \{\top, \bot\}$,
- a set of update operators $\mathcal{U}$. An update operator $\oplus \in \mathcal{U}$ is a mapping $\oplus : \mathcal{S} \times \underline{\mathcal{L}} \to \mathcal{S}$.

KR languages are compatible on a shared domain $\mathcal{D}$, when they both include variables over $\mathcal{D}$ and their sets of query and update operators are mutually disjoint. KR modules with compatible KR languages are compatible as well.

Remark 2.2. From the Definition 2.1 we have that a KR language not including variables is compatible with any other KR language.

Each query and update operator has an associated identifier. For simplicity, these are not included in the definition, however I use them throughout the text. When used as an identifier in a syntactic expression, I use informal prefix notation (e.g., $\models\varphi$ or $\oplus\varphi$), while when used as a semantic operator, formally correct infix notation is used (e.g., $\sigma\models\varphi$ or $\sigma \oplus \varphi$).

Query formulae are the syntactical means to retrieve information from KR modules.

Definition 2.3 (query). Let $\mathcal{M}_1, \ldots, \mathcal{M}_n$ be a set of compatible KR modules. Query formulae are inductively defined as follows

- if $\varphi \in \mathcal{L}_i$, and $\models \in \mathcal{U}_i$ corresponding to some $\mathcal{M}_i$, then $\models\varphi$ is a query formula,
- if ϕ_1, ϕ_2 are query formulae, so are $\phi_1 \wedge \phi_2$, $\phi_1 \vee \phi_2$ and $\neg\phi_1$.

The informal semantics is straightforward. If a ground language expression $\varphi \in \underline{\mathcal{L}}$ is evaluated to true by the corresponding query operator $\models$ w.r.t. a state of the corresponding KR module, then $\models\varphi$ is true in the agent's mental state as well. Note that non-ground formulae have to be first grounded before their evaluation (cf. Section 2.2). $\mathcal{Q}(\mathcal{A}) = \bigcup_{i=1}^{n} \mathcal{Q}_i \times \underline{\mathcal{L}_i}$ denotes the set of all ground primitive queries of a *BSM* $\mathcal{A}$.

Mental state transformers (*mst*'s) enable transitions from one state to another. A primitive mental state transformer $\oslash\psi$, typically denoted by ρ and constructed from an update operator $\oslash \in \mathcal{U}_i$ and a formula $\psi \in \mathcal{L}_i$, refers to an update on the state of the corresponding KR module. Mental state transformer is the principal syntactic construction of *BSM* framework.

Definition 2.4 (mental state transformer). Let $\mathcal{M}_1, \ldots, \mathcal{M}_n$ be a set of compatible KR modules. Mental state transformer expression (mst) is inductively defined:

- **skip** is a mst (primitive),
- if $\oplus \in \mathcal{U}_i$ and $\psi \in \mathcal{L}_i$ corresponding to some $\mathcal{M}_i$, then $\oplus\psi$ is a mst (primitive),
- if ϕ is a query expression, and τ is a mst, then $\phi \longrightarrow \tau$ is a mst as well (conditional),
- if τ and τ' are mst's, then $\tau|\tau'$ and $\tau \circ \tau'$ are mst's too (choice and sequence).

$\mathfrak{T}$ denotes the set of all mst's over the set of KR modules.

An update expression is a primitive mst. The other three (conditional, sequence and non-deterministic choice) are compound mst's. Informally, a primitive mst is encoding a transition between two mental states, i.e., a primitive behaviour. Compound mst's introduce hierarchical structure to the *BSM* framework. Similarly to set of primitive queries, $\mathcal{U}(\mathcal{A}) = \bigcup_{i=1}^{n} \mathcal{U}_i \times \underline{\mathcal{L}_i}$ denotes the set of all primitive ground update mst's of $\mathcal{A}$.

A mental state transformer encodes an agent behaviour. As the main task of an agent is to perform a behaviour, naturally an agent program can be fully characterized by a single mst and a set of associated KR modules used in it. Such a standalone mental state transformer is also called an *agent program* over a set of KR modules $\mathcal{M}_1, \ldots, \mathcal{M}_n$. A *Behavioural State Machine* $\mathcal{A} = (\mathcal{M}_1, \ldots, \mathcal{M}_n, \mathcal{P})$ is a collection of compatible agent KR modules and an associated agent program and it completely characterizes an agent system $\mathcal{A}$.

Example 2.5 (*Prolog* KR module). Let's assume $\mathcal{M}$ is a KR module implementing a *Prolog* (Shapiro and Sterling, 1994) knowledge base. It is characterized by

- $\mathcal{S}_\mathcal{M}$, the set of all states σ the *Prolog*'s interpreter memory context can be in, i.e., the set of all valid *Prolog* programs,
- $\mathcal{L}_\mathcal{M}$, the set of well formed *Prolog* formulae φ,
- $\mathcal{Q}_\mathcal{M} = \{\models_\mathcal{M}\}$, where $\models_\mathcal{M}(\sigma, \varphi) = \top$ iff the *Prolog* interpreter evaluates the *Prolog* query φ to `true`. Otherwise, $\models_\mathcal{M}(\sigma, \varphi) = \bot$.
- $\mathcal{U}_\mathcal{M} = \{\oplus_\mathcal{M}, \ominus_\mathcal{M}\}$, where $\oplus_\mathcal{M}(\sigma, \varphi) = \sigma'$, where σ' is a *Prolog* program resulting from execution of the query `?-assert(`φ`)`, in the context of the program σ. Similarly, $\ominus_\mathcal{M}(\sigma, \varphi) = \sigma'$ amounts to execution of the query `?-retract(`φ`)` in the context of the program σ,
- σ_0, an initial state of the KR module $\mathcal{M}$.

The following mental state transformer can be constructed over the single module $\mathcal{M}$

$$\models_\mathcal{M}\mathsf{pred(a)} \longrightarrow (\ominus_\mathcal{M}\mathsf{pred(a)} \circ \oplus_\mathcal{M}\mathsf{pred(b)}) \mid \oplus_\mathcal{M}\mathsf{done}$$

2.2 Semantics

Similarly to other logic based state-of-the-art BDI agent programming languages such as *AgentSpeak(L)/Jason* (Bordini et al., 2005b, 2007), *3APL* (Dastani et al., 2005b), *2APL* (Dastani, 2008) or *GOAL* (de Boer et al., 2007; Hindriks, 2009), the underlying semantics of a *BSM* is that of a labelled transition system over agent's mental states. It became a *de facto* tradition in the field of agent-oriented programming languages to

provide the semantics of a language in terms of an operational semantics modeled after Plotkin (1981). I.e., a set of semantic rules describing mental state transitions specifying how a particular agent system moves from one mental state to another by means of atomic changes of its partial knowledge bases. The framework of *Behavioural State Machines* also comes with a straightforward operational semantics. However, unlike the tradition in the field, it also specifies a crisp denotational view on agent programs written in it. Informally, a mental state transformer τ denotes a function over agent's mental state. For each mental state σ, it yields a set of new states σ' corresponding to transitions resulting from application of the various ground instantiations of the mst τ on σ. The functional view on agent programs turns out to be a powerful abstraction facilitating program compositionality and thus enabling hierarchical decomposition of agent programs.

2.2.1 Denotational view

Definition 2.6 (state). Let $\mathcal{A}$ be a *BSM* over KR modules $\mathcal{M}_1, \ldots, \mathcal{M}_n$. A state of $\mathcal{A}$ is a tuple $\sigma = \langle \sigma_1, \ldots, \sigma_n \rangle$ of KR module states $\sigma_i \in \mathcal{S}_i$, corresponding to $\mathcal{M}_1, \ldots, \mathcal{M}_n$ respectively. $\mathcal{S} = \mathcal{S}_1 \times \cdots \times \mathcal{S}_n$ denotes the space of all states over $\mathcal{A}$.

$\sigma_1, \ldots, \sigma_n$ are partial states of σ. A state can be modified by applying primitive updates on it and query formulae can be evaluated against it. Query formulae cannot change the actual agent's mental state.

According to Definition 2.1, to evaluate a formula in a state by query and update operators, the formula must be ground. Transformation of non-ground formulae to ground ones is provided by means of *variable substitution*. A variable substitution is a mapping $\theta : \mathcal{L} \to \underline{\mathcal{L}}$ replacing every occurrence of a variable in a KR language formula by a value from the corresponding domain. Variable substitution of a compound query formula is defined by usual means of nested substitution. Note however, that a variable can be substituted in subformulae of a compound formula only when languages of the corresponding subformulae share the domain of the variable in question. A variable substitution θ is *ground* w.r.t. ϕ, when the instantiation $\phi\theta$ is a ground formula.

Definition 2.7 (query evaluation). Let $\mathcal{A}$ be a BSM over KR modules $\mathcal{M}_1, \ldots, \mathcal{M}_n$. Let also $\varphi \in \underline{\mathcal{L}_i}$ be a primitive formula corresponding to a KR module $\mathcal{M}_i$ in state σ_i and let also $\models \in \mathcal{M}_i$ be a query operator of that module. Finally, let $\sigma = \langle \sigma_1, \ldots, \sigma_n \rangle$ be a mental state of $\mathcal{A}$. Then we write

- $\sigma_i \models \varphi$, iff $(\sigma_i \models \varphi) = \top$. Otherwise when $(\sigma_i \models \varphi) = \bot$, we use notation $\sigma_i \not\models \varphi$,
- $\sigma \models \phi$ iff $\phi = \models \varphi$, and $\sigma_i \models \varphi$,

The evaluation of compound query formulae on mental states inductively follows usual evaluation of nested logical formulae. I.e.,

- $\sigma \models \phi_1 \wedge \phi_2$ iff $\sigma \models \phi_1$ and $\sigma \models \phi_2$,
- $\sigma \models \phi_1 \vee \phi_2$ iff $\sigma \models \phi_1$ or $\sigma \models \phi_2$, and finally
- $\sigma \models \neg\phi$ iff it is not the case that $\sigma \models \phi$. We also write $\sigma \not\models \phi$.

Notions of an *update* and *update set* are the bearers of the semantics of mental state transformers. An update of a mental state σ is a tuple $(\oslash, \psi)$, where $\oslash$ is an update operator and ψ is a ground update formula corresponding to some KR module. The syntactical notation of a sequence of mst's joined by the operator $\circ$ corresponds to a sequence of their corresponding updates or update sets joined by the semantic sequence operator $\bullet$. Provided ρ_1 and ρ_2 are updates, also the sequence $\rho_1 \bullet \rho_2$ is an update. Additionally, there is a special no-operation update **skip** corresponding to the primitive mst **skip**.

A simple update corresponds to the semantics of a primitive mst. Sequence of updates corresponds to a sequence of primitive mst's and is a compound update itself. An update set is a set of updates and corresponds to a mst encoding a non-deterministic choice. Given an update or an update set, its application on a state of a *BSM* is straightforward. Formally,

Definition 2.8 (update application). Let $\mathcal{A}$ be a *BSM* over KR modules $\mathcal{M}_1, \ldots, \mathcal{M}_n$. The result of applying an update $\rho = (\oslash, \psi) \in \mathcal{U}(\mathcal{A})$ on a state $\sigma = \langle \sigma_1, \ldots, \sigma_n \rangle$ is a new state $\sigma' = \sigma \bigoplus \rho$, such that $\sigma' = \langle \sigma_1, \ldots, \sigma_i', \ldots, \sigma_n \rangle$, where $\sigma_i' = \sigma_i \oslash \psi$, and both $\oslash \in \mathcal{U}_i$ and $\psi \in \underline{\mathcal{L}_i}$ correspond to $\mathcal{M}_i$ of $\mathcal{A}$. Applying the empty update **skip** on the state σ does not change the state, i.e., $\sigma \bigoplus \mathbf{skip} = \sigma$.

Inductively, the result of applying a sequence of updates $\rho_1 \bullet \rho_2$ is a new state $\sigma'' = \sigma' \bigoplus \rho_2$, where $\sigma' = \sigma \bigoplus \rho_1$.

The meaning of a mental state transformer in a state σ, formally defined by the *yields* predicate below, is the update set it yields in that mental state.

Definition 2.9 (*yields* calculus). A mental state transformer τ yields an *update* ρ in a mental state $\sigma \in \mathcal{S}$ under a variable substitution θ, iff $yields(\tau, \sigma, \theta, \rho)$ is derivable in the following calculus:

$$\frac{\top}{yields(\mathbf{skip},\sigma,\theta,\mathbf{skip})} \qquad \frac{\top}{yields(\oslash\psi,\sigma,\theta,(\oslash,\psi\theta))} \qquad \text{(primitive)}$$

$$\frac{yields(\tau,\sigma,\theta,\rho), \sigma \models \phi\theta}{yields(\phi \longrightarrow \tau,\sigma,\theta,\rho)} \qquad \frac{yields(\tau,\sigma,\theta,\rho), \sigma \not\models \phi\theta}{yields(\phi \longrightarrow \tau,\sigma,\theta,\mathbf{skip})} \qquad \text{(conditional)}$$

$$\frac{yields(\tau_1,\sigma,\theta,\rho_1),\ yields(\tau_2,\sigma,\theta,\rho_2)}{yields(\tau_1 | \tau_2,\sigma,\theta,\rho_1),\ yields(\tau_1 | \tau_2,\sigma,\theta,\rho_2)} \qquad \text{(choice)}$$

$$\frac{yields(\tau_1,\sigma,\theta,\rho_1),\ yields(\tau_2,\sigma \bigoplus \rho_1,\theta,\rho_2)}{yields(\tau_1 \circ \tau_2,\sigma,\theta,\rho_1 \bullet \rho_2)} \qquad \text{(sequence)}$$

The mst **skip** yields the update **skip**. Provided a variable substitution θ, similarly, a primitive update mst $\oslash\psi$ yields the corresponding update $(\oslash, \psi\theta)$. In the case the condition of a conditional mst $\phi \longrightarrow \tau$ is satisfied in the current mental state, the calculus yields one of the updates corresponding to the right hand side mst τ, otherwise the no-operation **skip** update is yielded. A non-deterministic choice mst yields an update corresponding to either of its members and finally a sequential mst yields a sequence of updates corresponding to the first mst of the sequence and an update yielded by the second member of the sequence in a state resulting from application of the first update to the current mental state. Also note that Definition 2.9 assumes that the variable substitution θ is ground w.r.t. all the formulae occurring in the considered mst τ.

The calculus defining the *yields* predicate provides a *functional view* on a mst and it is the primary means of compositional modularity in *BSM*. Mental state transformers encode functions yielding update sets over states of a *BSM*. The collection of all the updates yielded w.r.t. Definition 2.9 comprises an update set of an agent program τ in the current mental state σ. The following definition articulates the denotational semantics of the notion of mental state transformer as an encoding of a *function* mapping mental states of a *BSM* to updates, i.e., transitions between mental states.

Definition 2.10 (mst denotational semantics). Let $\mathcal{M}_1, \ldots, \mathcal{M}_n$ be KR modules. A mental state transformer τ denotes a function $\mathfrak{f}_\tau : \sigma \mapsto \{\rho | \exists\theta : \mathit{yields}(\tau, \sigma, \theta, \rho)\}$ over the space of mental states $\mathcal{S}$.

Finally, the evolution of a *BSM* agent system in terms of transitions between mental states follows.

Definition 2.11 (step of a *BSM*). A *BSM* $\mathcal{A} = (\mathcal{M}_1, \ldots, \mathcal{M}_n, \tau)$ can make a step from a state σ to a state σ', iff the mst τ yields a non-empty update set in σ and $\sigma' = \sigma \bigoplus \rho$, where $\rho \in \mathfrak{f}_\tau(\sigma)$ is an update. We also say that $\mathcal{A}$ induces a (possibly compound) labelled transition $\sigma \xrightarrow{\rho} \sigma'$.

Remark 2.12. Definition 2.10 articulates the non-deterministic character of the *yields* calculus. Firstly, the semantic rule for choice operator | yields an update for each of the non-deterministic choice construct components. Secondly, each primitive mst, can yield an update set w.r.t. more than one ground variable substitution.

In fact, there can be an infinite number of such instantiations. Consider $\tau : \models (X > 0) \longrightarrow \oplus(p(X))$, a conditional mst over a KR module with language and semantics of first-order logic including the domain of integers. The mst consists of a query formula, which is satisfied for every X larger than 0. In turn, assertion $\oplus$ of the predicate $p(X)$ for each $X > 0$ is yielded by the conditional mst, i.e., $\forall\sigma : \mathfrak{f}_\tau(\sigma) = \{(\oplus, p(X)) : \forall X > 0\}$. Hence $|\mathfrak{f}_\tau(\sigma)| = \aleph_0$ for every $\sigma \in \mathcal{S}$.

Remark 2.13. Notice also that the provided semantics of choice and sequence operators implies associativity of both. Hence, from this point on, instead of the strictly pairwise notation $\tau_1|(\tau_2|(\tau_3|(\cdots|\tau_k)))$, I simply write $\tau_1|\tau_2|\tau_2|\cdots|\tau_k$ whenever appropriate. Similarly for the sequence operation $\circ$.

In fact, w.l.o.g. the choice and sequence rules of the *yields* calculus in Definition 2.9 can be reformulated as follows:

$$\frac{\tau{=}\tau_1|\cdots|\tau_k, 1{\leq}i{\leq}k, \mathit{yields}(\tau_i,\sigma,\rho_i)}{\mathit{yields}(\tau,\sigma,\rho_i)} \qquad \text{(choice)}$$

$$\frac{\tau{=}\tau_1\circ\cdots\circ\tau_k, \forall 1{\leq}i{\leq}k\colon \mathit{yields}(\tau_i,\sigma_i,\rho_i) \wedge \sigma_{i+1}{=}\sigma_i \bigoplus \rho_i}{\mathit{yields}(\tau,\sigma_1,\rho_1\bullet\cdots\bullet\rho_k)} \qquad \text{(sequence)}$$

2.2.2 Operational view

The underlying semantics of a *BSM* can also be seen in terms of traces within a labeled transition system over agent's mental states. Mental state transformers are interpreted as traces in a transition system over agent's mental states, where transitions are induced by updates. The notion of a *behavioural frame* formally captures the semantic structure induced by the set of KR modules of a *BSM* agent. In particular, it encapsulates the set of all mental states constructed from local states of the KR modules and applications of the corresponding update operators between them.

Definition 2.14 (behavioural frame). Let $\mathcal{A} = (\mathcal{M}_1, \ldots, \mathcal{M}_n, \mathcal{P})$ be a *BSM* over a set of KR modules $\mathcal{M}_i = (\mathcal{S}_i, \mathcal{L}_i, \mathcal{Q}_i, \mathcal{U}_i)$. The *behavioural frame* of $\mathcal{A}$ is a labeled transition system $\mathit{LTS}(\mathcal{A}) = (\mathcal{S}, \mathcal{R})$, where $\mathcal{S} = \mathcal{S}_1 \times \cdots \times \mathcal{S}_n$ is the space of mental states of $\mathcal{A}$, and the transition relation $\mathcal{R}$ is defined as follows:

$$\mathcal{R} = \{\sigma \xrightarrow{\rho} \sigma' \in \mathcal{S} \times \mathcal{U}(\mathcal{A}) \times \mathcal{S} \mid \sigma' = \sigma \bigoplus \rho\} \cup$$
$$\{\sigma \xrightarrow{\mathbf{skip}} \sigma \in \mathcal{S} \times \mathcal{U}(\mathcal{A}) \times \mathcal{S}\}$$

A tuple of KR modules $\overline{\mathcal{A}} = (\mathcal{M}_1, \ldots, \mathcal{M}_n)$, which is essential for constructing the behavioural frame is also called *behavioural template*. We will sometimes write $\mathit{LTS}(\overline{\mathcal{A}})$ instead of $\mathit{LTS}(\mathcal{A})$ since the mst $\mathcal{P}$ in $\mathcal{A}$ plays no role in the construction of the corresponding frame.

Note that $\mathit{LTS}(\mathcal{A})$ is finite (resp. enumerable) iff all the modules in $\mathcal{A}$ have finite (resp. enumerable) state spaces, languages, and repertoires of query and update operators.

The operational semantics of an agent is defined in terms of all possible computation runs induced by the iterated execution of the corresponding *BSM*. Let $\lambda = \sigma_0\sigma_1\sigma_2\ldots$ be a *trace* (finite or infinite). Then, $\lambda[i] = \sigma_i$ denotes the i-th state on λ, and $\lambda[i..j] = \sigma_i \ldots \sigma_j$ denotes the "cutout" from λ from position i to j. The ith prefix and suffix of λ are defined by $\lambda[0..i]$ and $\lambda[i..\infty]$, respectively.

Definition 2.15 (traces and runs of a BSM). Let $\mathcal{A} = (\overline{\mathcal{A}}, \tau)$ be a *BSM*. $\mathcal{T}(\mathcal{A})$ denotes the set of (complete) traces $\sigma_0\sigma_1 \ldots \sigma_k$, such that $\sigma_0 \stackrel{\rho_1}{\rightarrow} \sigma_1 \stackrel{\rho_2}{\rightarrow} \ldots \stackrel{\rho_k}{\rightarrow} \sigma_k$, and τ yields $\rho = \rho_1 \bullet \ldots \bullet \rho_k$ in σ_0, i.e., $\rho \in \mathfrak{f}_\tau(\sigma_0)$.

A possibly infinite sequence of states λ is a *run* of *BSM* $\mathcal{A}$ iff:

- there is a sequence of positions $k_0 = 0, k_1, k_2, \ldots$ such that, for every $i = 0, 1, 2, \ldots$, we have that $\lambda[k_i..k_{i+1}] \in \mathcal{T}(A)$, and
- *weak fairness*: if an update is enabled infinitely often, then it will be infinitely often selected for execution.

The semantics of an agent system characterized by *BSM* $\mathcal{A} = (\overline{\mathcal{A}}, \tau)$ is the set of all runs of $\mathcal{A}$, denoted $\mathcal{T}(\overline{\mathcal{A}}, \tau^*)$. The star superscript τ^* denotes iterated execution of τ.

The weak fairness condition of Definition 2.15 actually restates one of the standard fairness conditions (cf. e.g., (Manna and Pnueli, 1992)). It rules out traces where, for some non-deterministic choice $\tau_1|\tau_2$ in a program τ, always the same option is selected during the iterated execution of τ and the other option is always neglected.

Remark 2.16. Note that a trace of a *BSM* $\mathcal{A}$ with an agent program $\mathcal{P}$ can be defined as a result of a sequence of, possibly compound, steps of $\mathcal{A}$ according to Definition 2.11. I.e., iterated application of the function $\mathfrak{f}_\mathcal{P}$ to agent's mental states. The two views on *BSM* semantics complement each other while providing two different optics on an agent program: that of a *modifying function* and that of *runs of enabled sequences of transitions* yielded by its iterated evaluation.

Example 2.17 (mst semantics). Consider the mst τ from Example 2.5 and the behavioural frame $\overline{\mathcal{A}}$ over the single KR module $\mathcal{M}$.

τ encodes a function over the set of states of $\overline{\mathcal{A}}$. It enables transitions from states σ in which pred(a) is derivable, to states σ' differing from σ in that pred(a) is replaced by pred(b), or in which done is derivable. I.e., in all states σ in which $\models_\mathcal{M}$pred(a) holds, we have $\mathfrak{f}_\tau(\sigma) = \{(\ominus_\mathcal{M}$,pred(a))$\bullet(\oplus_\mathcal{M}$,pred(b)), $(\oplus_\mathcal{M}$,done)$\}$.

Alternatively, if σ_0 is a considered initial state of the $\overline{\mathcal{A}}$, s.t. $\sigma_0 \models_\mathcal{M}$pred(a), $\sigma_0 \not\models_\mathcal{M}$pred(b) and $\sigma_0 \not\models_\mathcal{M}$done, then $\lambda_1 = \sigma_0\sigma_1\sigma_2\sigma_3 \ldots$, where

- $\sigma_1 \not\models_\mathcal{M}$pred(a), $\sigma_1 \not\models_\mathcal{M}$pred(b) and $\sigma_1 \not\models_\mathcal{M}$done,
- $\sigma_2 \not\models_\mathcal{M}$pred(a), $\sigma_2 \models_\mathcal{M}$pred(b) and $\sigma_1 \not\models_\mathcal{M}$done and finally
- $\sigma_2 \not\models_\mathcal{M}$pred(a), $\sigma_2 \models_\mathcal{M}$pred(b) and $\sigma_1 \models_\mathcal{M}$done

is a possible run of the *BSM* $\mathcal{A} = (\overline{\mathcal{A}}, \tau)$.

Algorithm 2.1 Abstract *BSM* interpreter
input: agent program $\mathcal{P}$, initial mental state state σ_0

```
σ = σ_0
loop
    compute ϱ_P(σ) = {ρ | yields(P, σ, _, ρ)}
    if ϱ_P(σ) ≠ ∅ then
        non-deterministically and fairly choose ρ ∈ ϱ_P(σ)
        σ = σ ⊕ ρ
    end if
end loop
```

2.3 Abstract interpreter

Agents continuously perceive, deliberate and act in their environment, thus their lifecycle is open ended. According to the *BSM* semantics provided by Definition 2.11, a single step of an agent system amounts to a single application of its agent program, a mst function, to the current state of its knowledge bases. The system thus moves in a sequence of discrete steps, where each step is a single evaluation of the agent program yielding the step transition (cf. Remark 2.16). Algorithm 2.1 lists a straightforward pseudocode implementation of the *BSM* execution cycle. In a single deliberation cycle

1. the agent program interpreter computes the update set $\varrho_{\mathcal{P}}(\sigma)$ corresponding to the agent program $\mathcal{P}$ according to the Definition 2.9,

2. non-deterministically chooses an update ρ from $\varrho_{\mathcal{P}}(\sigma)$, and finally

3. updates the current mental state by applying the update ρ to it.

The omission of a variable substitution _ in the *yield*(...) expression denotes arbitrary applicable ground substitution of the set of all the free variables used in the encoding of the agent program $\mathcal{P}$.

The semantics provided in the previous section does not enforce a particular *BSM* language implementation. As follows from Remark 2.12, the *BSM* semantics and in turn also the Algorithm 2.1, is still underspecified. It leaves space for variations in different *BSM* interpreter implementations. Definition 2.15 only declares properties that execution traces produced by a *BSM* interpreter in question must fulfill. I.e., each (compound) transition must be a part of an enabled *BSM* step and the execution trace must satisfy the weak fairness condition.

Algorithm 2.1 can directly serve as a template for implementation of a concrete *BSM* engine. In order to instantiate it, designers of an efficient and compliant *BSM* interpreter must deal with the following design issues.

Shared domains In order to enable information exchange between independent KR modules, these have to be compatible, i.e., they have to include variables over the same domains (cf. Definition 2.1). While it is possible to devise many ways how to deal with this issue, it seems to be reasonable to restrict the number and cardinality of shared domains over which the KR modules can be defined. Note that domains with infinite or finite but very large cardinality can potentially lead to computation difficulties, such as the case highlighted in Remark 2.12.

Variable substitution To implement the semantics of the *yields* calculus as provided by Definition 2.9, the interpreter must be able to compute a complete ground variable substitution for the agent program. As for each primitive query over one of agent's KR modules, there can be several, possibly even an infinite number of candidates, the interpreter designer must also decide on constraining the substitution mechanism within modules.

Non-deterministic choice The chosen strategy w.r.t. establishing a suitable ground variable substitution also influences the cardinality of the update set $\mathfrak{f}_{\mathcal{P}}$ corresponding to the agent program $\mathcal{P}$. The number of candidate updates in the update set is further multiplied by non-deterministic choice operators in the program $\mathcal{P}$, which yield a set of candidate updates for each of its branches. Note that the weak fairness condition of Definition 2.15 does not specify a concrete mechanism how to choose from the number of updates in $\mathfrak{f}_{\mathcal{P}}(\sigma)$ in a state σ. As far as the fairness of the choice is secured, the interpreter is free to employ any compliant strategy of non-deterministic choice.

Various design choices for tackling the issues described above lead to various implementations of compliant *BSM* interpreters. *Jazzyk*, a *BSM* interpreter presented in Chapter 5 demonstrates one such.

2.4 Summary

This chapter is based mainly on my earlier works (Novák, 2008a,c) with some notation adaptations from the joint paper with Jamroga (Novák and Jamroga, 2009). In this chapter I introduced the theoretical foundation of the presented dissertation, the theoretical framework of *Behavioural State Machines*. It provides three major contributions to the state of the art in the area of languages for programming cognitive agents:

KR modularity: *BSM* framework provides modularity of employed knowledge representation technology as a first class concept. *BSM* is abstract enough to accommodate any required KR technology or interface to agent's environment, while only providing a means for encoding interdependencies among objects stored in the modules. Thus, instead of *extending* the language with a fixed set of mental attitudes and

their interrelationships, with *BSM* I put forward a framework which can be *instantiated* for a specific application in development.

code structure: the concept of *mental state transformer* allows for hierarchical structuring of agent program and thus facilitates *hierarchical decomposition* of agent programs. This is of particular use in the process of gradual refinement of a, possibly only partial, high level specification down to a concrete agent program written in terms of interactions between agent's knowledge bases and interfaces to the environment.

abstraction: the *functional view* on the semantics of mental state transformer provides a powerful abstraction for *BSM* agent program. Mental state transformer facilitates encapsulation of a set of particular interdependent objects stored in agent's knowledge bases together with the dynamics associated with them, a part of agent's overall behaviour. The construct of mst thus provides a similar abstract concept for programming cognitive agents as does an object in the object oriented programming paradigm. In particular, the functional semantics allows to view agent programs as functions modifying its mental states, decomposed into (or composed of) compounds of other encapsulated subfunctions, possibly with a similar nested structure.

The remainder of this dissertation provides a substance to the above claims. In particular the Chapters 4 and 6 subsequently discuss formal support for the process of specification refinement and design of BSM agent program. Both heavily rely on the functional view on subprograms, i.e., mental state transformers.

Chapter 3

Modular BDI architecture

Among the most prominent state-of-the-art architectural concepts for development of agents with mental state is the *Belief-Desire-Intention* (BDI) architecture (cf. Section 3.1). Informally, BDI prescribes decomposition of an agent system into several specialized knowledge bases storing agent's mental attitudes, such as beliefs, goals, etc. One of the main aims of this dissertation is to propose a novel approach to design of agent-oriented programming languages, with a particular focus on BDI inspired agent systems. In the heart of the proposal lies the framework of *Behavioural State Machines*, introduced in the previous chapter.

The plain *BSM* framework does not implicitly provide first-class concepts for programming cognitive agents with mental attitudes, such as e.g., *beliefs* or *goals*. To demonstrate the flexibility of the *BSM* framework, as well as to provide support for the remainder of this dissertation, in this chapter I describe *modular BDI Architecture*, an instantiation of the BDI architecture as a *BSM*. The introduced BDI-style architecture serves as a substrate for examples and case-studies described the following chapters. Subsequently, I formally introduce *Ape*, the airport assistant robot (cf. Chapter 1). Ape is a BDI inspired *BSM* agent and serves as the running example throughout this dissertation.

3.1 Belief Desire Intention

Most state-of-the-art approaches for designing and programming cognitive agents rely in one way or another on Bratman's *Belief-Desire-Intention* philosophy (1999). It builds on the theory of *intentional stance* by Dennett (1987), which postulates that agents should be assumed to maintain mental attitudes such as *beliefs* and *goals*[1] and at the same time they should be assumed to act rationally. Subsequently, we can use *practical reasoning* to predict their future actions towards achieving their goals, taking into account their beliefs about the actual state of the world. Bratman extends this viewpoint in that besides *beliefs* and *goals*, he proposes another first-class mental attitude for agent's *intentions*. I.e., the agent maintains a set of desires, together with corresponding courses of action,

[1] Even though probably incorrect from a strict philosophical point of view, here I use the terms *desire* and *goal* interchangeably.

plans, towards achieving them. Unlike agent's desires, which can possibly be in a state of mutual conflict, the set of intentions should be consistent. Thus the agent is able to adopt a new intention only when it is not in a conflict with those it already committed to. Paraphrasing van Riemsdijk (2006), existing intentions thus enjoy a certain level of inertia and form the *screen of admissibility*, a term originally coined by Bratman (1999).

Here, I focus on pragmatic aspects of engineering cognitive agents. While taking into account the philosophical background of the BDI architecture, I rather focus on treating it only as a source of inspiration, but not as a precise blueprint for designing a practical programming system for agent systems. In turn, I do not explore here the deeper philosophical context of the BDI theory, but rather try to isolate the fragmentary ideas which are useful in the practice of programming agents.

3.1.1 Beliefs, goals and intentions

As I already note above, the most common BDI-inspired agent architecture prescribes an architectural decomposition of an agent into *informational*, *motivational* and *procedural* components. Agent's *belief base*, the informational component, maintains its information about the current state of its environment, its peers or itself. A *goal base*, the motivational component, maintains descriptions of states of affairs the agent eventually wants to believe or bring about in the future. Finally, the procedural component, a library of agent's plans/intentions, glues together its behaviours in the form of actions in the environment and the corresponding goals. While the belief base describes *now*, the current state of the world, the goal base maintains descriptions of alternative *futures* the agent considers. Finally, the intentions, if treated as a first-class concept, describe the courses of actions to be taken from *now* to the *futures* the agent committed to.

Most of the state-of-the-art BDI programming systems do treat agent's beliefs and goals as first-class primitives. There, however, does not seem to be a common agreement about a crisp semantics of intentions from the software engineering point of view. Languages such as 3APL (Dastani et al., 2005b) or *AgentSpeak(L)/Jason* (Bordini et al., 2007) come with an explicit representation of agent's intentions. These are either stored in an explicit component of the agent system (e.g., *AgentSpeak(L)*), or are associated with agent's goals and stored in the agent's goal base (e.g., *3APL*). On the other hand languages such as *GOAL* (Hindriks, 2009) model only beliefs and goals as language primitives and the agent performs action selection in every deliberation step of its lifecycle. The intention of such an agent can be identified only *a posteriori* from its execution trace, a sequence of actions taken in order to reach agent's goal.

3.1.2 Agent reasoning model

Primitive language objects stored in agent's knowledge bases, such as the belief or the goal base, provide only a static snapshot of its mental state. The dynamics of the

agent system is provided by a mechanism regulating the flow of information between the system components, the *agent reasoning model*. For instance, an agent adopts, drops or modifies a goal or an intention on the ground of holding certain beliefs. *Vice versa*, it can also modify its beliefs because it acted in a certain way or perceived a change in the environment. There are various formal axiomatic systems prescribing interrelationships and dependencies among agent's mental attitudes. Their specifications usually culminate in formalization of the notion of a *commitment*. The most prominent approaches w.r.t. current state of research are the Cohen and Levesque's formalization of persistent relativized goals (1990) and the Rao and Georgeff's *I-System* (1991).

To illustrate a useful example of a formal agent reasoning model, in the following I describe the *I-System*, a set of axioms constraining interactions among agent's mental attitudes in a rational agent.

I-System

I-System is a set of eight basic axioms (A1-A8) imposing constraints on interrelationships among agent's mental attitudes. Informally, an agent should adopt only goals it believes to be an option w.r.t. to its beliefs (AI1). An agent should adopt intentions only in order to achieve its goals (AI2). If an agent has an intention to perform a certain action, it will eventually also perform it (AI3). It should be aware of the fact that it committed itself to certain goals and intentions (AI4, AI5). If an agent intends to achieve something, it also has to have a goal to intend it (AI6). It should be aware of its actions, i.e., if it performs an action, it should also later believe that it performed it (AI7). And finally, an agent should never hold its intentions forever, i.e., each intention must be eventually dropped (AI8).

The particular way how an agent handles its intentions gives rise to various models of commitment. In their original paper, Rao and Georgeff (1991) provide examples of *blind*, *single minded*, and *open minded* agent. Informally, a blindly committed agent maintains its intentions until it actually believes that it had achieved them. Single minded agent maintains them as long as it believes they are still possible to achieve. Finally, an open minded agent maintains its intentions only as long as these are still also its goals.

3.2 BDI instantiation in BSM

While the previous section briefly summarizes the core ideas underpinning the BDI architecture, in the following I sketch its instantiation as a template for *BSM* agent systems. The resulting *behavioural template* (cf. Definition 2.14 in Chapter 2) for BDI-style agent systems, *BSM*s, will serve as a substrate for the remainder of the dissertation.

3.2.1 Beliefs, goals and the body

For simplicity, instead of the complex BDI architecture introduced in the previous section, the instantiation presented here introduces only *beliefs* and *goals* as first-class primitives of the behavioural template. The procedural aspect, intentions, is treated implicitly as behaviours associated with, and triggered by particular goals. The corresponding KR modules storing objects representing the two agent's mental attitudes are a *belief base* $\mathcal{B}$ and a *goal base* $\mathcal{G}$ respectively.

Unlike other BDI inspired programming systems for cognitive agents, which focus primarily on internal reasoning of agents, I consider systems embodied in an environment. I treat agent's interaction with the environment as a first-class primitive, on a par with beliefs and goals. A *BSM* KR module providing sensor and actuator interfaces, query and update operators respectively, is the module *body* $\mathcal{E}$. The $\mathcal{E}$ stands for interactions with en $\mathcal{E}$nvironment.

In order to abstract from the system's internal semantics and KR technologies employed by its informational and motivational components, I consider only an interface specification for the introduced KR modules.

Definition 3.1 (BDI behavioural template). Consider three *BSM* KR modules $\mathcal{B}$, $\mathcal{G}$ and $\mathcal{C}$ with the following characteristics:

belief base $\mathcal{B}$: stores and reasons with formulae representing beliefs of an agent. It provides a single query operator $\models_{\mathcal{B}}$ and two update operators $\oplus_{\mathcal{B}}, \ominus_{\mathcal{B}}$ for asserting and retracting a belief formula from the belief base respectively.

goal base $\mathcal{G}$: stores and reasons with formulae representing agent's goals. Similarly to the belief base $\mathcal{B}$, it provides a single query operator $\models_{\mathcal{G}}$ and two update operators $\oplus_{\mathcal{G}}, \ominus_{\mathcal{G}}$ for asserting and retracting a goal formula from the goal base.

body $\mathcal{E}$: provides a pair $\models_{\mathcal{E}}, \oslash_{\mathcal{E}}$ of a query and an update operator. The query operator $\models_{\mathcal{E}}$ provides a sensory input for the agent, while the update operator $\oslash_{\mathcal{E}}$ facilitates performing agent's actions in the environment.

The resulting BDI style behavioural template is denoted as $\overline{\mathcal{A}_{BDI}} = (\mathcal{B}, \mathcal{G}, \mathcal{E})$.

Note that the definition above, imposes only minimal constraints on the interfaces of the belief and the goal base. Considering a KR language formula φ, it does not even require that the formula is represented in the module as a distinguished object. Rather, the formula should be only derivable from the information stored in the module according to the internal module semantics. Thereby, the behavioural template is still capable to accommodate a wide range of knowledge representation technologies, even a non-classic ones, such as e.g., an artificial neural network or a relational database.

3.2.2 Agent reasoning model

The behavioural template $\overline{\mathcal{A}_{BDI}}$, introduced in Definition 3.1, provides an architectural decomposition of a BDI-style cognitive agent system. Its dynamics is provided by a *BSM* agent program, a standalone mental state transformer, in terms of a specification of interactions between the KR modules of the system. The behavioural template induces a behavioural frame (cf. Definition 2.14) over the set of KR modules. I.e., a transition system an agent instantiating the template is allowed to traverse during its lifetime. An implemented *BSM* agent provides a specification of a concrete cut-out of the behavioural frame specific to the considered application domain by means of its program.

The *BSM* framework does not come with a fixed set of axiomatic rules regulating interdependencies among the mental attitudes of an agent. Instead, it is the responsibility of an application programmer to encode both, the application-domain-specific behaviour of the agent, as well as the specific agent reasoning model. As I discuss later in Chapter 6, it is reasonable to equip application programmers with a domain-independent library of code patterns and domain-independent guidelines for implementation of a specific model of rationality for a specific (sub-)class of agents. In the following, I discuss a set of simple informal mst templates for implementation of an agent reasoning model. The model is tailored for the behavioural template $\overline{\mathcal{A}_{BDI}}$, i.e., it does not include an explicit KR module for handling agent's base of intentions. I discussed a complete instantiation of the *I-System* including a base of intentions in the joint paper with Dix (Novák and Dix, 2006).

To indicate that the introduced rules are only application-domain-independent templates, instead of the proper *BSM* notation, I use the scheme $Q_{\mathcal{X}} \longrightarrow U_{\mathcal{Y}}$, with $\mathcal{X}, \mathcal{Y}$ being placeholders for KR module identifiers according to Definition 3.1. Figure 3.1 provides a schematic overview of the BDI behavioural template together with an example of some interaction rules.

I-System in BSM

A closer look at the flow of information in the *I-System*, reveals that on the ground of events occurring in the environment, an agent adopts beliefs, which in turn are the basis for adopting desires. Based on desires, intentions are adopted. These finally serve as the basis for considering agent's actions in the environment. This information flow can be informally described as a cycle of BDI components through which information flows $\mathcal{E} \rightarrow \mathcal{B} \rightarrow \mathcal{G} \rightarrow \mathcal{I} \rightarrow \mathcal{E}$.

Since the BDI behavioural template from the previous subsection lacks an explicit base of intentions, the cycle considered here is reduced to $\mathcal{E} \rightarrow \mathcal{B} \rightarrow \mathcal{G} \rightarrow \mathcal{E}$. I.e., an agent's perceptions of the environment are reflected in its belief base. Because of certain beliefs, it adopts or drops goals, and finally, in order to achieve a goal, it selects a particular behaviour and executes the corresponding actions, updates, in the environment. These, indirectly, lead to new perceptions or events in the environment. Straightforward set of

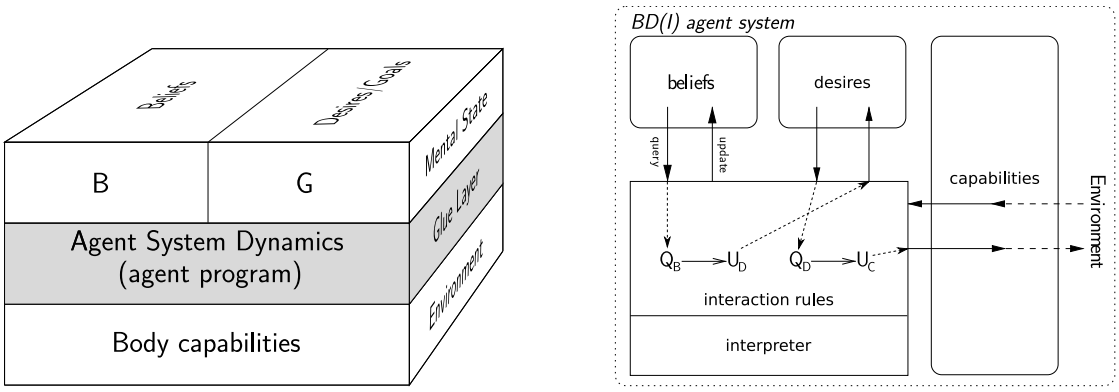


Figure 3.1: Architectural decomposition of the agent system (left) and an overview of the introduced BDI behavioural template with examples of interaction rules (right).

conditional mst's implementing the flow of information takes the form of the following schemata: $Q_{\mathcal{E}} \longrightarrow U_{\mathcal{B}}$ for perception, $Q_{\mathcal{B}} \longrightarrow U_{\mathcal{G}}$ implements goal handling, and finally $Q_{\mathcal{G}} \longrightarrow U_{\mathcal{E}}$ facilitates behaviour selection and acting. These rules directly satisfy the axioms AI1, AI2, AI3 of the *I-System*. Axiom AI7 is satisfied only indirectly, since the agent can learn about success of its actions only by perceiving changes possibly caused by it in the environment. An alternative way to implement this axiom is to keep history of actions the agent tried to perform in its belief base. This can be achieved this by interaction rules of type $Q_{\mathcal{G}} \longrightarrow U_{\mathcal{E}} \circ U_{\mathcal{B}}$ (AI3).

Axioms AI4 and AI5 are secured via accessibility of all BDI components to the *BSM* agent interpreter. The rules examine desire and intention bases by means of queries, so the agent's reasoning mechanism is "aware" of the content of its components.

Axiom AI6 can be satisfied by the execution of rules of the type $Q_{\mathcal{G}} \longrightarrow U_{\mathcal{E}}$. By evaluating the agent program, in which the actual behaviours in the environment depend on agent's goals, the action selection mechanism ensures that whenever the agent desires a goal, it also intends to execute the associated behaviour.

Commitment strategies

Rao and Georgeff (1991) provide examples of *blind*, *single minded*, and *open minded* agent. In the context introduced above, this amounts to a particular strategy of handling goals on the ground of holding certain beliefs in the agent's belief base. The set of rules above, partially implementing the *I-System*, naturally satisfies the definition of an open minded agent, i.e., it considers certain behaviours, intentions, only as long as goals triggering them are derivable from the goal base.

To implement the blind commitment strategy, the agent program must ensure that whenever it starts to believe that a goal satisfaction condition is satisfied, it must drop the goal associated with that condition. Finally, implementation of a single minded com-

mitment strategy requires the agent to be able to reason about its own future evolutions within its belief base. Such introspective techniques are however beyond the scope of this work.

Apart from rational agents, in the *BSM* framework we can also model *irrational*, in the sense of *not rational w.r.t. I-System*, agent reasoning in our framework. Consider an example of a *servant* agent, into which goals can be "implanted" from outside as commands it should follow, although it doesn't necessarily believe they are an option w.r.t. its beliefs. This can be implemented by allowing rules of the type $Q_{\mathcal{E}} \longrightarrow U_{\mathcal{G}}$. Informally, such a rule can be interpreted as *"upon perceiving a command C, the agent should adopt C as a goal"*. The agent is, however, still able to deliberate about the goal in relation to the other goals it already has and their mutual interactions.

A slightly stricter version of the servant agent, is a *slave* agent. Upon perceiving a signal from outside, e.g., a message from another agent, the slave directly reacts by performing the corresponding reactive behaviour of the type $Q_{\mathcal{E}} \longrightarrow U_{\mathcal{E}}$. Note that the last scheme corresponds to a purely reactive agent model. Slave agents resemble more software objects in their traditional meaning in object oriented programming. They simply invoke the requested procedure without having a "free will" not to do so.

As an extreme case of irrational reasoning, a *religiously fanatic* agent can be considered. Such an agent doesn't want to believe facts which are in conflict with its other goals. According to its current goals, it is thus able to change its beliefs by executing rules of the type $Q_{\mathcal{G}} \longrightarrow U_{\mathcal{B}}$.

Irrational agents, such as the slave, feature only a constrained autonomy. They thus do not comply with classic definitions of an agent. Mixing rationality with a certain degree of non-rational behaviour, at least w.r.t. certain aspects of the functionality, can, however, lead to more efficient implementations of agent systems. The advantage of the liberal approach of the *BSM* framework is that an agent developer is free to implement either a very strict model of agent reasoning, or to adapt it by mixing in shortcuts to the flow of information whenever needed.

3.3 Ape the Airport E-Assistant

The following example provides the architecture for *Ape*, the *Airport E-Assistant* robot from Chapter 1. The running example of *Ape* will be used throughout the following chapters of the dissertation.

Example 3.2 (behavioural template for *Ape*). *Ape* instantiates the BDI behavioural template $\overline{\mathcal{A}_{BDI}}$ (cf. Definition 3.1). In particular, *Ape*'s belief and goal bases are implemented as *Prolog* programs[2]. It's interface to the environment is facilitated by an interface to a *Java* program interpreted by a virtual machine.

[2]For illustrative proposes, I chose *Prolog*. The language formulae provide declarative reading and thus are rather intuitive.

Listing 3.1 Programs implementing *Ape*'s KR modules: belief base $\mathcal{B}$ (left), goal base $\mathcal{G}$ (bottom) and body $\mathcal{E}$ (right).

```
%%% Fragment of Ape's belief base B %%%
% Information about Ape himself
at(self, coor(123,456,678)).
object(self). person(self).
low_battery :-
        energy(Level),
        Level < 50.

% Information about the airport infrastructure
at('base_line_terminal', coor(98,78,54)).
at('BBWings_counter', coor(43,53,67)).
comm_gateway('wlan2', 'yellow_gtw0', 4200).

% Information about objects around
obstacle(Object) :- \+ person(Object).

% Information about passengers
businessman(Person) :-
        person(Person),
        wears(Person, suit),
        \+ holds(Person, luggage).

tourist(Person) :-
        person(Person),
        wears(Person, casual),
        \+ holds(Person, laptop_bag).

patient(Person) :-
        tourist(Person) ;
        woman(Person) ;
        child(Person).

anxious(Person) :-
        businessman(Person) ;
        elderly(Person).

% Asserted facts
object('Bronja').
person('Bronja').
woman('Bronja').
wears('Bronja', casual).
holds('Bronja', backpack).
```

```
/*** Fragment of Ape's body implementation E ***/
public class Camera {
    /* Perceptions */
    public static boolean see(String oID) {...};
    public static
        boolean distance(String oID, Integer dist) {...} ;
    public static
        boolean recognize(String oID, String type) {...};
    public static
        boolean wears(String pID, String type) {...};
    public static
        boolean holds(String pID, String oID) {...};
}

public class Audio {
    /* Perceptions */
    public static
        boolean hears(String oID, String msg) {...};

    /* Actions */
    public static void say(String msg) {...};
}

public class Body {
    /* Perceptions */
    public static  boolean getBattery(int level) {...};

    /* Actions */
    public static void turn(Integer angle) {...};
    public static void forward(int dist) {...};
    public static void backward(int dist) {...};

    /* HW management */
    public static void boot() {...};
    public static void shutdown() {...};
}

public class Gps {
    /* Perceptions */
    public static
        boolean location(int X, int Y, int Z) {...};
}
```

```
%%% Fragment of Ape's goal base G %%%
% Detection of conflicting goals
conflict(X) :-
        desire(achieve(at(X))),
        desire(achieve(at(Y))),
        X \== Y.

% Identification of concrete goals
goal(X) :-
        desire(X),
        \+ conflict(X).

% Ape's concrete desires
desire(maintain(safety)).
desire(maintain(energy)).
desire(achieve(has_task)).
desire(achieve(at('base_line_terminal'))).
desire(achieve(whereIs('wifi_lounge'))).
```

Listing 3.2 Agent program implementing *Ape*'s behaviour for guiding passengers who need a help to a destination point.

```
/*** Perceptions ***/
⊨_E camera.see(ObjectID) ⟶ ⊕_B object(ObjectID) |
⊨_E camera.recognize(ObjectID, human) ⟶ ⊕_B person(ObjectID) |
⊨_E gps.location(X,Y,Z) ⟶ ⊕_B at(me, coor(X,Y,Z)) |


/*** Goal commitment strategy ***/
%% Adopt condition %%
not ⊨_G goal(achieve(at(Place))) ∧ ⊨_B person(Person) ∧ ⊨_E audio.hears(Person, 'get me to ' + Place)
                ⟶ ⊕_G desire(achieve(at(Place))) |

%% Drop condition %%
⊨_G goal(achieve(at(Place))) ∧ ⊨_B at(Place, coor(X,Y,Z)), at(self, coor(X,Y,Z))
                ⟶ ⊖_G desire(achieve(at(Place))) |


/*** Act ***/
⊨_G goal(achieve(at(Place))) ∧
not ⊨_B at(Place, coor(X,Y,Z)), at(self, coor(X,Y,Z)) ∧
⊨_B at(Place, coor(X,Y,Z)), angleTo(coor(X,Y,Z), Angle)
                ⟶ (
                        %% The way is free %%
                        not (⊨_E camera.see(ObjectID) ∧ ⊨_E camera.recognize(ObjectID, obstacle))
                                        ⟶ (⊘_E body.turn(Angle) ∘ ⊘_E body.forward(3);) |

                        %% Obstacle ahead %%
                        ⊨_E camera.see(ObjectID) ∧ ⊨_E camera.recognize(ObjectID, obstacle)
                                        ⟶ (
                                                %% Avoid obstacle by stepping back and %%
                                                %% either turning left or right %%
                                                ⊘_E body.backward(3) ∘
                                                (⊘_E body.turn(90) | ⊘_E body.turn(−90)) ∘
                                                ⊘_E body.forward(3)
                                        )
                )
```

Consider initialization of the individual KR modules with the programs listed in Listing 3.1. The interfaces of the individual KR modules are implemented as follows.

belief and goal bases $\mathcal{B}, \mathcal{G}$: the corresponding KR modules implement the interface of the KR module $\mathcal{M}$ introduced in Example 2.5. The only difference is that from now on, the operators corresponding to $\mathcal{B}$ and $\mathcal{G}$ will be subscripted by the corresponding module identifier.
In the initial state, as specified in Listing 3.1, when *Ape* spots *Bronja*, among others he adds also the facts $\oplus_{\mathcal{B}}$person('Bronja') and $\ominus_{\mathcal{B}}$holds('Bronja', backpack) about her to his belief base. Since both $\models_{\mathcal{B}}$patient('Bronja') and $\models_{\mathcal{B}}$tourist('Bronja') now evaluate to $\top$, in result, *Ape* believes that *Bronja* is a patient tourist. Similarly, *Ape*'s goals include a desire to get to the *Base Line* terminal, i.e., $\oplus_{\mathcal{G}}$goal(achieve(at('base_line_terminal'))).

body $\mathcal{E}$: the query operator $\models$ takes a plain *Java* code snippet in the form of a string φ and passes it to the running *Java* VM started with the initial program. The query code must return a Boolean value according to the *Java* language semantics. The update operator $\oslash\varphi$ behaves exactly the same way, except it does not care for the return value. I.e., update formulae do not have to return any value and are treated as arbitrary blocks of code.
When *Ape* spots *Bronja*, he derives from his body module for example the following formulae $\oslash_{\mathcal{E}}$see('Bronja'), $\models_{\mathcal{E}}$isPerson('Bronja'). In order to guide her to the *Base Line* shuttle terminal, *Ape* should at some point navigate through the space by executing (sequences of) updates, actions, such as e.g., $\oslash_{\mathcal{E}}$turn(90) or $\oslash_{\mathcal{E}}$forward(6).

Listing 3.2 shows a fragment of a possible *Ape*'s *BSM* agent program. *Ape* perceives the environment and notices objects and humans around, as well as regularly updates his current position. Whenever a person asks him for assistance, e.g., to get to a certain destination, *Ape* adopts a goal to fulfill the request. When he eventually realizes that the task is fulfilled, he drops the goal. Finally, in order to get to the destination, *Ape* has a choice of two mutually exclusive behaviours to either move forward or to avoid the obstacle, if he detects one in front of him. The last mst implementing *Ape*'s behaviour in the environment demonstrates how a *BSM* code can be hierarchically structured.

3.4 Summary

Above, I describe a *modular BDI architecture*, an instantiation of the BDI architecture as a *BSM* based agent template. The bulk of this chapter is based on my joint work with Dix (Novák and Dix, 2006). The main contributions of the discussion presented in this chapter follow.

KR modularity: inherited from the *BSM* framework itself, the scheme of modular BDI behavioural template allows to plug-in different KR technologies for agent's *belief base* $\mathcal{B}$ and the *goal base* $\mathcal{G}$. Moreover, it facilitates an easy integration with various environments the agent can be embodied in through the *body module* $\mathcal{E}$.

model of rationality: I show how a chosen model of rationality can be implemented as a fragment of the agent's *BSM* program. Because of the liberal nature of the *BSM* framework, a developer is able to flexibly adapt the model of rationality w.r.t. parts of the agent program (beliefs, goals and actions) to the particular needs of the application. In this sense, the language for encoding *BSM* agent programs, mental state transformers, can be seen as a *meta-language* for implementation of agent's deliberation cycle and its reasoning model.

Chapter 4

Logic for Behavioural State Machines

Source code modularity and reusability are one of the principal concerns of pragmatic software engineering. To support reusability, especially in teams of programmers, the code must provide clear interfaces and crisp *semantic characterization* of its functionality. The maxim of such semantic characterization is a non-ambiguous language of formal logic. A tight relationship between a programming framework and a logic for reasoning about programs created in it is vital for a formal study of engineering non-trivial agent systems.

According to the semantics of *Behavioural State machines* (cf. Definition 2.15), a *BSM* program specifies a set of traces, a cut-out from the corresponding labeled transition system over the space of agent's mental states. To enable reasoning about execution traces of such programs, in this chapter, I introduce *Dynamic Computation Tree Logic DCTL**. *DCTL** is a novel extension of the full branching time temporal logic *CTL** (Emerson, 1990) with features of *Dynamic Logic* (Harel et al., 1984). To bridge the gap between the flexible, but logic-agnostic programming framework of *Behavioural State Machines* and *DCTL**, the logic for verification of *BSM* programs, I introduce *program annotations* in the form of formulae of *Linear Time Temporal Logic* (*LTL*) (Pnueli, 1977). While *LTL* provides a tool for relating mental state transformers to formulae of temporal logic, *DCTL** allows expressing and reasoning about properties of executions of agent programs. For simplicity, I consider only ground *BSM* agent programs, i.e., mental state transformers without variables.

4.1 Linear Time Temporal Logic LTL

Mental states of *BSM* agents are composed of partial states, theories in logic-agnostic[1] KR languages of the corresponding KR modules. For example, as demonstrated by Example 3.2, the interface of one module can be based on *Java*, another on *Prolog*, while queries and mst's of yet another module can be given in the assembly language. Later in this chapter, I show how a relationship between such KR modules and logical formulae can be obtained by means of *LTL* annotations. Before doing so, however, this

[1] I do not assume any relationship between these languages and mathematical logic.

section introduces a variant of logic *LTL* by Pnueli (1977), used in the remainder of this chapter. First, an extension of behavioural frames with an interpretation of basic logical statements follows.

Definition 4.1 (behavioural model). Let $LTS(\mathcal{S}, \mathcal{R})$ be the behavioural frame of a behavioural template $\overline{\mathcal{A}} = (\mathcal{M}_1, \ldots, \mathcal{M}_n)$. The *behavioural model* of $\overline{\mathcal{A}}$ is defined as $\overline{LTS}(\overline{\mathcal{A}}) = (\mathcal{S}, \mathcal{R}, \Pi, \pi)$, where $LTS(\overline{\mathcal{A}}) = (\mathcal{S}, \mathcal{R})$ is the behavioral frame of $\overline{\mathcal{A}}$, $\Pi = \{\mathsf{p} \mid \phi \in \mathcal{Q}(\overline{\mathcal{A}})\}$ is the set of atomic propositions, and $\pi : \Pi \to 2^{\Pi}$ defines their valuations so that they correspond to primitive queries: $\pi(\mathsf{p}_\phi) = \{\sigma \in \mathcal{S} \mid \sigma \models \phi\}$.

Behavioural model of a *BSM* is defined as the behavioural model of the underlying behavioural template: $\overline{LTS}(\overline{\mathcal{A}}, \mathcal{P}) = \overline{LTS}(\overline{\mathcal{A}})$.

Informally, the notion of behavioural model translates the logic-agnostic structure of a behavioural frame over a *BSM* states into a Kripke model (Kripke, 1963) over which modal logic formulae can be interpreted.

LTL (Pnueli, 1977) enables reasoning about properties of execution traces by means of temporal operators $\bigcirc$ (in the next moment) and $\mathcal{U}$ (until). Additional operators $\Diamond$ (sometime in the future) and $\Box$ (always in the future) can be defined as $\Diamond\varphi \equiv \top\,\mathcal{U}\,\varphi$ and $\Box\varphi \equiv \neg\Diamond\neg\varphi$. In order to allow capturing the nature of sequential composition of mental state transformers, in the following I introduce a version of *LTL* that includes the "chop" operator $\mathcal{C}$ similar to that by Rosner and Pnueli (1986). I.e., when constructing an aggregate annotation for $\tau_1 \circ \tau_2$, the use of the chop operator $\varphi_1\,\mathcal{C}\,\varphi_2$ enforces that the formula φ_1 referring to τ_1 is fulfilled *before* the execution of τ_2, characterized by φ_2, begins.

Definition 4.2 (*LTL* syntax). Formally, the version of *LTL* is given by the following grammar

$$\varphi ::= \mathsf{p} \mid \neg\varphi \mid \varphi \wedge \varphi \mid \bigcirc\varphi \mid \varphi\,\mathcal{U}\,\varphi \mid \varphi\,\mathcal{C}\,\varphi$$

Other Boolean operators in *LTL* syntax, such as disjunction $\vee$, material implication $\rightarrow$, etc., are defined in the usual way.

The logic's semantics is defined as a satisfaction relation between an *LTL* formula and a *BSM*, together with its trace through the clauses below.

Definition 4.3 (*LTL* semantics). Let $\mathcal{A}$ be a BSM and $\lambda \in \mathcal{T}(\mathcal{A})$ is its trace. Then

- $\mathcal{A}, \lambda \models \mathsf{p}$ iff $\lambda[0] \in \pi(\mathsf{p})$ in $\overline{LTS}(\mathcal{A})$,
- $\mathcal{A}, \lambda \models \neg\varphi$ iff $\mathcal{A}, \lambda \not\models \varphi$,
- $\mathcal{A}, \lambda \models \varphi_1 \wedge \varphi_2$ iff $\mathcal{A}, \lambda \models \varphi_1$ and $\mathcal{A}, \lambda \models \varphi_2$,
- $\mathcal{A}, \lambda \models \bigcirc\varphi$ iff $\mathcal{A}, \lambda[1..\infty] \models \varphi$,

- $\mathcal{A}, \lambda \models \varphi_1 \mathcal{U} \varphi_2$ iff there exists $i \geq 0$, such that $\mathcal{A}, \lambda[i..\infty] \models \varphi_2$, and $\mathcal{A}, \lambda[j..\infty] \models \varphi_1$ for every $0 \leq j < i$,
- $\mathcal{A}, \lambda \models \varphi_1 \mathcal{C} \varphi_2$ iff there exists $i \geq 0$, such that $\mathcal{A}, \lambda[0..i] \models \varphi_1$ and $\mathcal{A}, \lambda[i..\infty] \models \varphi_2$.

LTL formula φ is *valid in* $\mathcal{A}$ (written $\mathcal{A} \models \varphi$) iff φ holds on each trace $\lambda \in \mathcal{T}(\mathcal{A})$.

4.2 Dynamic Computation Tree Logic DCTL*

Later on, the *LTL* formulae will be used to capture the semantic characterization of *BSM* mst's. The idea is straightforward: to facilitate reasoning about *BSM* programs, these should provide associated annotations in the language of *LTL* logic. Since each annotation is assigned to a particular mst, there is no point in referring to the mst in the object language. However, a richer logic for *reasoning about programs* and their relationships is needed. Namely one, which allows to address a particular program explicitly. To this end, I extend the branching-time logic *CTL** (Emerson, 1990) with explicit quantification over program executions. In the extension, $[\tau]$ stands for "*for all executions of* τ". The complementary form "*there is an execution of* τ" can be defined as $\langle\tau\rangle\varphi \equiv \neg[\tau]\neg\varphi$. As the agenda of the logic resembles that of "dynamic *LTL*" by Henriksen and Thiagarajan (1999), the resulting logic is called "*dynamic CTL**", *DCTL** in short.

Definition 4.4 (*DCTL syntax).** The syntax of *DCTL** extends that of *LTL* as follows

$$\begin{array}{rcl} \theta & ::= & \mathsf{p} \mid \neg\theta \mid \theta \wedge \theta \mid [\tau]\varphi \\ \varphi & ::= & \theta \mid \neg\varphi \mid \varphi \wedge \varphi \mid \bigcirc \varphi \mid \varphi \mathcal{U} \varphi \mid \varphi \mathcal{C} \varphi \end{array}$$

where τ is a program or an iterated program.

Definition 4.5 (*DCTL semantics).** Let $\mathcal{A}$ be a *BSM* and $\lambda \in \mathcal{T}(\mathcal{A})$ be its trace. The *DCTL** semantics extends that of *LTL* (cf. Definition 4.3) with the clauses

- $\mathcal{A}, \lambda \models \theta$ iff $\mathcal{A}, \lambda[0] \models \theta$,
- $\mathcal{A}, \sigma \models \mathsf{p}$ iff $\sigma \in \pi(\mathsf{p})$,
- $\mathcal{A}, \sigma \models \neg\theta$ iff $\mathcal{A}, \sigma \not\models \theta$,
- $\mathcal{A}, \sigma \models \theta_1 \wedge \theta_2$ iff $\mathcal{A}, \sigma \models \theta_1$ and $\mathcal{A}, \sigma \models \theta_2$,
- $(\overline{\mathcal{A}}, \mathcal{P}), \sigma \models [\tau]\varphi$ iff for each $\lambda \in \mathcal{T}(\overline{\mathcal{A}}, \tau)$, s.t. $\lambda[0] = \sigma$, we have that $(\overline{\mathcal{A}}, \tau), \lambda \models \varphi$.

*DCTL** formula θ is *valid in* $\mathcal{A}$ iff $\mathcal{A}, \sigma \models \theta$ for every state σ. We also write $\mathcal{A} \models \theta$.

The following proposition shows the relationship between *LTL* and *DCTL**:

Proposition 4.6. *For every BSM* $\mathcal{A} = (\mathcal{M}_1, \ldots, \mathcal{M}_n, \tau)$ *and an LTL formula* φ*, we have*

$$\mathcal{A} \models_{\text{LTL}} \varphi \textit{ iff } \mathcal{A} \models_{\text{DCTL}^*} [\tau]\varphi.$$

The proof straightforwardly follows from *LTL* being subsumed by *DCTL**.

Note the last point of the Definition 4.5. It extends the standard *CTL** logic with semantic definition for formulae resembling *Dynamic Logic* formulae. Unlike in *Dynamic Logic*, however, such formulae do not specify what happens right *after* the execution of the program τ, but rather speak out about what happens *during* execution of the program. This difference is crucial as it allows a sound translation of logic-agnostic *BSM* programs into the language of logic and subsequently reason about their executions.

The notion of *semantic consequence* is the basis for relating different characterizations of the same *BSM*.

Definition 4.7 (semantic consequence). Formula ψ is a semantic consequence of φ, i.e., $\varphi \Rightarrow \psi$, iff for every *BSM* $\mathcal{A}$, $\mathcal{A} \models \varphi$ implies $\mathcal{A} \models \psi$.

Usually, I will use the notion of semantic consequence to write formulae of the form $[\tau]\varphi \Rightarrow [\tau^*]\psi$. That is, if formula φ correctly describes possible executions of program τ, then ψ holds for all possible iterated executions of the program.

4.3 Temporal annotations and verification of BSMs

The *BSM* framework allows us to encode agent programs in terms of compound mst's interpreted in a behavioural model over a behavioural template, a set of KR modules. Now, the idea is to use *LTL* and *DCTL** for reasoning about execution traces in such models. To bridge the gap between the mental states of a *BSM* and interpreted states of behavioural models, I introduce *Annotated Behavioural State Machines*, i.e, *BSM*s enriched with *LTL* annotations of primitive queries and updates occurring in the corresponding agent program. The basic methodological assumption behind the proposal is that a KR module supplies a set of basic test and procedures for programming agent with it. I.e., the set of all primitive query and update formulae which can be constructed from the module KR language $\mathcal{L}$ and its sets of operators $\mathcal{Q}$ and $\mathcal{U}$. Complementary, the corresponding annotations provide a re-interpretation of these from logic-agnostic programming KR languages into a single language for reasoning about properties of agent programs.

Definition 4.8 (annotated *BSM*). *Annotated BSM* is a tuple $\mathcal{A}^{\mathfrak{A}} = (\mathcal{M}_1, \ldots, \mathcal{M}_n, \mathcal{P}, \mathfrak{A})$, where $\mathcal{A} = (\mathcal{M}_1, \ldots, \mathcal{M}_n, \mathcal{P})$ is a *BSM*. *Annotation function*

$$\mathfrak{A} : (\mathcal{U}(\mathcal{A}) \cup \mathcal{Q}(\mathcal{A})) \rightarrow \textit{LTL}$$

assigns an *LTL* annotation to each primitive query and update occurring in $\mathcal{A}$.

Technically, it suffices to annotate only the queries and mst's that occur in the program $\mathcal{P}$ of the considered *BSM*. Annotations of primitive queries and mst's are meant to be provided by agent developer(s), according to their insight and expertise.

In order to enable reasoning about larger agent programs, higher level characterizations can be extracted from the lower level ones or even from primitive annotations. Given a complex mst τ, its annotation is determined by the combination of the annotations of its subprograms w.r.t. the outermost composition operator in τ.

Definition 4.9 (aggregation of annotations). Let $\mathcal{A}^{\mathfrak{A}}$ be an annotated *BSM*. We extend the function $\mathfrak{A}$ to provide also *LTL* annotations for compound queries and mst's recursively as follows

- let ϕ, ϕ' be queries, then $\mathfrak{A}(\neg\phi) = \neg\mathfrak{A}(\phi)$, $\mathfrak{A}(\phi \wedge \phi') = \mathfrak{A}(\phi) \wedge \mathfrak{A}(\phi')$, and $\mathfrak{A}(\phi \vee \phi') = \mathfrak{A}(\phi) \vee \mathfrak{A}(\phi')$,
- let ϕ be a query and τ be an mst, then $\mathfrak{A}(\phi \longrightarrow \tau) = \mathfrak{A}(\phi) \rightarrow \mathfrak{A}(\tau)$,
- let τ_1, τ_2 be mst's, then $\mathfrak{A}(\tau_1 | \tau_2) = \mathfrak{A}(\tau_1) \vee \mathfrak{A}(\tau)$ and $\mathfrak{A}(\tau_1 \circ \tau_2) = \mathfrak{A}(\tau_1)\, \mathcal{C}\, \mathfrak{A}(\tau)$.

Note that in order to capture the composition of annotations for a sequential mst, I use the *LTL chop* operator. The resulting formula states that exactly from the state right after the finished execution of τ_1 on, the remaining trace segment is characterized by the annotation of the second component of the sequence, τ_2.

As already noted above, annotations are not intended to be just arbitrary logical formulae. They should capture the relevant aspects of the queries and programs that they are assigned to. Thus, the annotations in $\mathcal{A}^{\mathfrak{A}}$ are assumed to be *sound* in the following sense.

Definition 4.10 (soundness of annotations). Let $\mathcal{A}^{\mathfrak{A}} = (\mathcal{M}_1, \ldots, \mathcal{M}_n, \mathcal{P}, \mathfrak{A})$ be an annotated *BSM*.

1. $\mathfrak{A}$ is sound in $\mathcal{A}^{\mathfrak{A}}$ w.r.t. a query φ iff $\mathfrak{A}(\varphi)$ holds in exactly the same mental states of $\overline{LTS}(\mathcal{A})$ as φ,
2. $\mathfrak{A}$ is sound in $\mathcal{A}^{\mathfrak{A}}$ w.r.t. program τ iff $\mathfrak{A}(\tau)$ holds for all traces from $\mathcal{T}(\mathcal{M}_1, \ldots, \mathcal{M}_n, \tau)$.

$\mathfrak{A}$ is sound in $\mathcal{A}^{\mathfrak{A}}$ iff it is sound w.r.t. a program $\mathcal{P}$ in $\mathcal{A}^{\mathfrak{A}}$. Note that $\mathfrak{A}$ is sound in $\mathcal{A}^{\mathfrak{A}}$ iff $\mathcal{A} \models [\mathcal{P}]\mathfrak{A}(\mathcal{P})$.

Proposition 4.11. *If $\mathfrak{A}$ is sound for every primitive query and update in $\mathcal{A}^{\mathfrak{A}}$, then $\mathfrak{A}$ is sound in $\mathcal{A}^{\mathfrak{A}}$.*

Proof. For every mst τ (resp. query φ), the soundness of $\mathfrak{A}$ w.r.t. τ (resp. φ) follows by induction on the structure of τ (resp. φ). Note that the aggregation rules in Definition 4.9 preserve soundness. □

Provided an *LTL* specification and an annotated *Behavioural State Machine*, we are usually interested whether the runs generated by the machine satisfy the specification.

Definition 4.12 (BSM verification). Let $\mathcal{A}^{\mathfrak{A}} = (\mathcal{M}_1, \ldots, \mathcal{M}_n, \mathcal{P}, \mathfrak{A})$ be an annotated *BSM* and $\phi \in LTL$ be a specification. We say that the iterated execution of $\mathcal{A}^{\mathfrak{A}}$ satisfies the specification Φ iff

$$\mathcal{A} \models [\mathcal{P}^*]\phi.$$

The following proposition turns out to be helpful in verification of *BSM* consisting of a non-deterministic choice of mst's.

Proposition 4.13. *Let $\overline{\mathcal{A}}$ be a behavioural template, $\mathfrak{A}$ be a sound annotation function w.r.t. $\overline{\mathcal{A}}$ and τ_1, τ_2 be mst's. Then,* $[(\tau_1|\tau_2)^*]\Box(\Diamond\mathfrak{A}(\tau_1) \wedge \Diamond\mathfrak{A}(\tau_2))$.

Proof. Follows immediately from the weak fairness condition, which ensures that from any point on, in an mst $\tau_1|\tau_2$, both τ_1, as well as τ_2 will be executed infinitely often. □

4.4 Verifying Ape

Example 4.14. Recall *Ape*'s *BSM* agent program from Example 3.2, Listing 3.2. Let $\mathcal{A}_{Ape} = (\mathcal{B}, \mathcal{G}, \mathcal{E}, \mathcal{P})$ be a *BSM*, implementing *Ape*'s behaviour.

Let's assume that the corresponding annotated *BSM* $\mathcal{A}^{\mathfrak{A}}_{Ape} = (\mathcal{A}_{Ape}, \mathfrak{A})$ uses the "human centric" annotation function $\mathfrak{A}$ as specified in Table 4.1. Moreover, *Ape*'s programmer knows that iterated execution of the complex behaviour τ_{to_dest}, shown in Listing 4.1, implements *Ape* wandering around the airport towards the destination point. I.e., iterated execution of τ_{to_dest} eventually brings about that *Ape* arrives to the desired destination. Formally,

$$\mathfrak{A}((\tau_{to\_dest})^*) = \Diamond\, at(dest)$$

A single execution of τ_{to_dest} is supposed to take *Ape* closer to the destination point. However, since the environment is unpredictable and *Ape*'s actions can fail, the programmer often cannot say too much about a single execution of an action in the environment. Yet, repeated attempts to perform the behaviour can still eventually satisfy to the desired aim.

Rewriting the agent program $\mathcal{P}$ in terms of *DCTL** formulae it yields, we have

$$\begin{aligned}[\mathcal{P}] \quad & ((see(person) \rightarrow \bigcirc\, knows(person)) \wedge \\ & (not\, goto(dest) \ \wedge \ knows(person) \rightarrow \bigcirc\, goto(dest)) \wedge \\ & (goto(dest) \ \wedge \ at(dest) \rightarrow \bigcirc\, \neg goto(dest))).\end{aligned}$$

primitive query/update		$\mathfrak{A}(\tau)$
$\models_{\mathcal{E}}$camera.see(ObjectID)	$\mapsto$	$\top$
$\models_{\mathcal{E}}$camera.recognize(ObjectID, human)	$\mapsto$	$see(person)$
$\oslash_{\mathcal{E}}$body.recognize(ObjectID, obstacle)	$\mapsto$	$\top$
$\models_{\mathcal{E}}$says(Person, 'get me to ' + Place)	$\mapsto$	$\top$
$\models_{\mathcal{E}}$gps.location(X,Y,Z)	$\mapsto$	$\top$
$\oslash_{\mathcal{E}}$body.forward(Steps)	$\mapsto$	$\top$
$\oslash_{\mathcal{E}}$body.backward(Steps)	$\mapsto$	$\top$
$\oslash_{\mathcal{E}}$body.turn(Angle)	$\mapsto$	$\top$
$\models_{\mathcal{B}}$object(ObjectID)	$\mapsto$	$\top$
$\models_{\mathcal{B}}$person(Person)	$\mapsto$	$knows(person)$
$\models_{\mathcal{B}}$at(Place, coor(X,Y,Z)), at(self, coor(X,Y,Z))	$\mapsto$	$at(dest)$
$\models_{\mathcal{B}}$at(Place, coor(X,Y,Z)), angleTo(coor(X,Y,Z), Angle)	$\mapsto$	$\top$
$\oplus_{\mathcal{B}}$person(Object)	$\mapsto$	$\bigcirc\, knows(person)$
$\models_{\mathcal{G}}$goal(achieve(at(Place)))	$\mapsto$	$goto(dest)$
$\oplus_{\mathcal{G}}$desire(achieve(at(Place)))	$\mapsto$	$\bigcirc\, goto(dest)$
$\ominus_{\mathcal{G}}$desire(achieve(at(Place)))	$\mapsto$	$\bigcirc\, \neg goto(dest)$

Table 4.1: Specification of the annotation function $\mathfrak{A}$ for the primitive query and update formulae occurring in Example 3.2.

Listing 4.1 An excerpt from Listing 3.2. τ_{to_dest}, a subprogram of the complete *BSM* agent program $\mathcal{P}$, implements *Ape*'s behaviour of moving towards a destination.

```
τ_to_dest = (
                %% The way is free %%
                not (⊨_E camera.see(ObjectID) ∧ ⊨_E camera.recognize(ObjectID, obstacle))
                        ⟶ (⊘_E body.turn(Angle) ∘ ⊘_E body.forward(3);) |

                %% Obstacle ahead %%
                ⊨_E camera.see(ObjectID) ∧ ⊨_E camera.recognize(ObjectID, obstacle)
                        ⟶ (
                                %% Avoid obstacle by stepping back and %%
                                %% either turning left or right %%
                                ⊘_E body.backward(3) ∘
                                (⊘_E body.turn(90) | ⊘_E body.turn(−90)) ∘
                                ⊘_E body.forward(3) ∘
                        )
)
```

At the same time, we also know

$[\mathcal{P}^*] \quad (goto(dest) \ \wedge \ \neg at(dest) \rightarrow \Diamond at(dest))$.

Straightforwardly, from the program $\mathcal{P}$, and the additional information about the annotation function $\mathfrak{A}$, we can derive

$$[\mathcal{P}^*] \ (see(person) \rightarrow \Diamond at(dest))$$

Informally, whenever *Ape* meets a person which needs assistance with getting to a destination, he will eventually bring about a state in which they together arrived to the destination point.

4.5 Summary

The work presented in this chapter is a result of my joint work with Jamroga (Novák and Jamroga, 2009). I introduce *DCTL**, a hybrid of full computation tree temporal logic *CTL** by Emerson (1990) and *Propositional Dynamic Logic* due to Harel et al. (1984). *DCTL** is tailored to facilitate reasoning about ground *BSM* agent programs. To bridge the gap between logic-agnostic query and update primitives, I introduce *LTL* annotations of the primitive constructs. Subsequently, characterizations of higher level mental state transformers, can be extracted from the lower level ones or even from primitive annotations.

The presented logic, together with the verification result for *DCTL** is the basis for introducing *BSM* design patterns I discuss in Chapter 6.

Part II

Software engineering issues

... in which I describe a concrete instance of the proposed novel approach to design of agent-oriented programming languages. On the substrate of a generic language for programming reactive systems, I introduce a set of agent oriented design patterns.

Let reactivity rule over deliberation!

(The moral of this story)

Chapter 5

Jazzyk

The framework of *Behavioural State Machines* provides only formal means for encoding of agent programs. In order to be able to use the framework in practice, programs must be encoded in a machine readable code and subsequently executed according to the *BSM* semantics. To this end, I introduce in this chapter a programming language called *Jazzyk*, together with its interpreter. Finally, I discuss the use of macros defined over the basic programming language. Macros transparently facilitate reusability and modularization of the language without modifying its core semantics.

5.1 Language

Jazzyk closely implements the syntax of *Behavioural State Machines* introduced in Chapter 2. The *BSM* framework provides only a mathematical notation, which needs to be translated into a machine readable character set. The programming language thus consists of a set of keywords and delimiters denoting 1) declaration and initialization of agent's KR modules, and 2) encoding of the associated agent program. The concrete form of the keyword set and supplementary language notation is rather arbitrary and completely at liberty of the language designer. In the language proposal, I commit to the following two language design principles:

1. programs written with the language should have an intuitive syntax and thus should be easily readable also to laymen, and

2. the structural notation should be inspired by that used in mainstream imperative structured programming languages, such as *C* or *Java*.

Of course, the above two principles are rather a matter of my own personal taste and preferences. Even though the readability requirement might lead to a "talkative" programming language, I believe that when applied to an experimental academic language it serves a better purpose than a dense notation cluttered with punctuation marks.[1]

[1] In contrast to e.g., *Jason* language by Bordini et al. (2007).

```
program       ::= (statement)*
statement     ::= module_decl | module_notify | mst*
module_decl   ::= ‘declare’ ‘module’ <moduleId> ‘as’ <KRModuleType>
module_notify ::= ‘notify’ <moduleId> on
                  (‘initialize’ | ‘finalize’ | ‘cycle’) formula
mst           ::= ‘nop’ | ‘exit’  |
                  update | block | sequence | choice | conditional
block         ::= ‘{’ mst ‘}’
sequence      ::= mst ‘,’ mst
choice        ::= mst ‘;’ mst
conditional   ::= ‘when’ query_exp ‘then’ mst [‘else’ mst]
query_exp     ::= query_exp ‘and’ query_exp |
                   query_exp ‘or’ query_exp |
                  not ‘query’ | ‘(’ query_exp ‘)’ | query
query         ::= ‘true’ | ‘false’ |
                  <operatorId> <moduleId> [variables] formula
update        ::= <operatorId> <moduleId> [variables] formula
formula       ::= ‘[{’ <arbitrary string> ‘}]’
variables     ::= ‘(’ (<identifier> ‘,’)* <identifier> ‘)’ | ‘(’ ‘)’
```

Figure 5.1: *Jazzyk* syntax definition. The specification follows the EBNF variant including regular expression specification, also called EBNF for XML and formally defined by W3C (2008b).

5.1.1 Well formed programs

The *Jazzyk* syntax straightforwardly follows that of the *BSM* framework. Figure 5.1 lists the EBNF syntax of the *Jazzyk* language.

The core of the *BSM* syntax is provided by the Definition 2.4 of the syntax of mental state transformers. In order to enable efficient use of such programs, few technical issues have to be handled as well. In particular, a *Jazzyk* program consists of a sequence of statements, which can take form of a *module declaration*, *module notification* or encode a *mental state transformer* which must be encoded in the programming language.

KR module handling

KR modules have to be declared and subsequently bound to the corresponding plug-ins implementing their functionality in a KR language of choice. *Module declaration statement* module_decl facilitates declaration of a module identified by moduleId and indicates that it is an instance of a plug-in identified by KRModuleType identifier.

Before a query or an update operation is invoked on a KR module, it should be initialized by some initial state. This state is encoded as a corresponding KR language formula, i.e., a block of a programming language code. Similarly, when a module is about to be shut down, it might be necessary to perform a clean-up of the knowledge

base handled by the module. In order to allow KR module initialization and shut-down, *Jazzyk* introduces so called *KR module notifications*. They take a form of a statement declaring a formula, a code block, to be executed when the KR module is loaded (**initialize**) or unloaded (**finalize**).

Before an agent program is executed, all the initialization notifications are executed and thus the initial states of corresponding KR modules are set. Similarly, the finalization notifications are executed right before the agent program finishes. This is the case either 1) after an explicit invocation of the program end instruction (see below), 2) after an error during program interpretation, or 3) after the last deliberation cycle was performed in the case a limit on the number of agent deliberation cycles was specified (cf. *Jazzyk* manual in Appendix A). The finalization should clean-up the states of KR modules, i.e., release resources possibly held by them, disconnect devices, etc.

Additionally, as a purely technical feature, also a notification after each deliberation cycle (**cycle**) is provided. It should serve strictly technical purposes like e.g., possible query cache clean-up or time counter increment, in the case the KR module implements such optimization techniques.

Mental state transformers

In the core of the *Jazzyk* syntax are rules of conditional nested mental state transformers of the form $query \longrightarrow mst$. These are translated in *Jazzyk* as "**when** query_exp **then** mst". Mst's can be joined using a sequence '`,`' and choice '`;`' operators, corresponding to *BSM* operators $\circ$ and $|$ respectively. The operator precedence can be managed using curly braces '`{`', '`}`', resulting in an easily readable syntax of nested code blocks similar to code blocks of e.g., the *C* language. Compound query expressions are a straightforward translation of the syntax of *BSM* query formulae. They can be joined by Boolean operators **and**, **or** and **not** and nested using parentheses '`(`' and '`)`'.

Each KR module provides a set of named query and update operators, identifiers of which are used in primitive query and update expressions. To allow the interpreter to distinguish between arbitrary strings and variable identifiers in primitive query and update expressions, *Jazzyk* allows programmers to provide an explicit declaration of a list of variables used in them. The list in parenthesis follows the module identifier in the query invocation construct.

A standalone update expression can be seen as a shortcut for a *BSM* rule of the type $\top \rightarrow$<update>. For programmers' convenience *Jazzyk* introduces the usual syntactic sugar of "**when**–**then**–**else**" for conditional mst's.

Supplementary keywords

To allow for the **skip** mst, *Jazzyk* includes the reserved keyword `nop`, no operation. Finally, the keyword `exit` denotes the program end instruction and its execution stops the agent's deliberation cycle. It also triggers execution of KR modules' finalization notifications and finally stops the language interpreter.

The *Jazzyk* syntax closely matches that of the theoretical *BSM* framework. Note however, two minor differences between *Jazzyk* and *BSM* agent programs. Unlike the *BSM* syntax, *Jazzyk* also allows empty agent programs. Furthermore, the agent program syntax, the root level mental state transformer allows interleaving declaration statements and mst definitions. In result, the root level mst can be specified as a sequel of mst's without explicit composition operators between them. Such a set of mst's is treated as a non-deterministic choice mst. This feature supports agent code modularity described in a closer detail in the Subsection 5.2.3 below. Briefly, agent programs are allowed to be composed of a number of subprograms possibly defined in separate files which can be included into the main program. Since the program developer cannot know *a priori* whether the included file, a code module, contains a full fledged mst or only various declarations, omission of the composition operator naturally yields a non-deterministic choice mst.

5.1.2 Valid programs

Besides checking the well-formedness of agent programs described in the previous section, the implemented *Jazzyk* interpreter also checks *program validity*, i.e., a set of not purely syntactic conditions imposed on agent programs.

Definition 5.1. A well-formed *Jazzyk* program is also *valid* when it satisfies the following conditions

1. query expressions can involve only query operators of the already declared KR modules,
2. similarly, primitive update mst's can involve only update operators of already declared KR modules,
3. a KR module identifier can be used in a query or an update mst only after it was previously declared,
4. finally, it must be the case that each variable occurring in a primitive update mst τ had to already occur in a query expression of a higher level mst. I.e., in a preceding mst of the sequence the update belongs to $(\cdots \circ \tau \circ \cdots)$, or in a query expression of some conditional mst including τ $(\phi \longrightarrow (\cdots \longrightarrow (\cdots \tau \cdots) \cdots))$.

The first two conditions straightforwardly follow from the *BSM* syntax. Since these conditions are not of a purely syntactical nature, *Jazzyk* interpreter must check them additionally after successfully parsing the program. Condition 3 supports good practices in programming, i.e., *use only what was already declared before*. Finally, the last condition 4 secures groundness of primitive update executions (cf. Definition 2.8 and Definition 2.9).

5.2 Interpreter

As already noted in Chapter 2, in order to allow efficient implementation of a *BSM* interpreter, several pragmatic issues have to be addressed first. In the following, I describe how I dealt with these issues in the implemented *Jazzyk* interpreter.

5.2.1 Principles

Shared domains To enable general, yet efficient information exchange between KR modules of an agent, the *Jazzyk* interpreter restricts shared domains of KR modules to a single domain. Namely that of ASCII character sequences, strings. Character strings are available in all mainstream programming languages ranging from *Prolog*, through *LISP* or *Scheme* to *Java*, *C* and *C++*, etc. Moreover, the advantage of strings is their universality and versatility. For most data types available in various KR technologies, there either already exists a well established conversion or a serialization technique to strings, or it is rather straightforward to come up with one. By employing binary-to-text encoding techniques, such as e.g., Base64 due to Josefsson (2006), it is for example possible to transfer even binary data objects between KR modules of a *Jazzyk* agent.[2]

Variable substitution To handle the inherent non-determinism of the *BSM* framework (cf. Remark 2.12), *Jazzyk* interpreter imposes constraints on KR modules' query operators implementation. When a query operator is invoked on the actual state of a KR module, it must return a *single valid ground variable substitution*, in the case such exists. The returned variable substitution is supposed to be the best according to the internal semantics of the KR module. Although in theory this constraint severely limits the *BSM* semantics implementation in the *Jazzyk* interpreter, it allows for an efficient and fast agent program interpretation. The actual precise mechanism for selecting the ground variable substitution by query operators of a KR module should be documented by KR module developers.

Non-deterministic choice A *BSM* interpreter needs to choose from the set of updates yielded by the agent program. As discussed above, a query operator implementation must return a single variable substitution, when such exists. Thus, the *Jazzyk* interpreter does not need to calculate the complete yielded update set in a bottom-up fashion as defined by the *yields* calculus (Definition 2.9), i.e., yielded updates are supposed to be collected from primitive mst's up to the root mst. Instead, the interpreter processes the agent program in a top-down fashion. When facing a choice among members of a non-deterministic choice mst, the interpreter randomly selects one of them and tries to recursively apply it to the current mental state. Chosen sequence mst's are simply

[2] We actually apply this technique in practice in the *Urbibot* agent described in Chapter 7. There we transfer Base64 encoded images from a camera sensor to agent's belief base for further processing.

executed as a sequel of mst applications. In the case the chosen member mst does not yield an update, the interpreter chooses a different one until no other choice member is left, or an update is yielded. A mst yields no update when it does not directly contain a primitive mst, possibly as a sequence member, and all queries of conditional mst's contained in it evaluate to `false` ($\bot$). In turn, the first yielded update is chosen for execution. Thanks to the random mst selection from choice mst's performed according to a uniform selection probability distribution, the weak fairness property imposed by the Definition 2.15 is also satisfied.

Query evaluation

Finally, to further optimize the agent program execution, the *Jazzyk* interpreter evaluates compound query expressions from left to right. I.e., to evaluate a conjunction of queries, the left-most is invoked first, and only when it evaluates to `true` ($\top$), the interpreter proceeds to the second one, etc. In the case some of the conjuncts evaluates to `false` ($\bot$), the interpreter breaks the compound query evaluation and returns `false` as well. The conjunction members right from the false one are not further executed.

Similarly, for a compound query disjunction, the interpreter searches for the first left-most primitive query which evaluates to `true` ($\top$). When it finds one, it breaks the query evaluation and returns `true`. Note that only the variable substitution of the left-most disjunct evaluated to `true` is taken to the right hand side of the conditional mst.

Finally, similarly to *Prolog* variable substitution, in the case a negation of a query formula evaluates to true, no variable substitution is yielded and taken to evaluation of the right hand side of the conditional mst.

5.2.2 Algorithm

The above discussed simplifications of the original *BSM* semantics were introduced to make the process of agent program interpretation more efficient and more transparent to the programmer. Algorithm 5.1 lists the commented pseudocode of the core of the actual *Jazzyk* interpreter implementation, i.e., algorithm for applying a mental state transformer to a mental state. In the case of primitive, conditional, block and **skip** mst's, the interpreter directly follows the *BSM* semantics' *yields* calculus in a top-down fashion. In the case of a non-deterministic choice mst, the interpreter randomly permutes the order of choice members and then tries to apply its members one after another until one of them executes, yields, a primitive update. Since the interpreter introduces a special mst **exit** for terminating the agent's lifecycle, the sequence mst handler must additionally check whether a sequence member does not yield an **exit** primitive which immediately breaks and terminates the interpreter. Algorithm 5.2 finally lists the straightforward sketch of the actual *Jazzyk* interpreter pseudocode.

Finally, without a rigorous formal proof, the following proposition sketches the correspondence between the *Jazzyk* interpreter as described by Algorithm 5.2 and the *BSM*

Algorithm 5.1 Pseudocode of the mst application in a mental state.
input: mental state transformer τ and current mental state $\sigma = (\sigma_1, \ldots, \sigma_n)$
return value: $\top$ indicates that τ yielded some update, $\bot$ indicates that τ does not yield any update in σ and finally $\times$ indicates that τ yielded the **exit** instruction

```
function APPLY_MST(τ)
    executed ← ⊥
    switch (τ)
        case 'update_⊘ module_k [{ψ}]':                    ▷ primitive update
            invoke update_⊘(ψ) on σ_k of M_k               ▷ update the current state:
            executed ← ⊤                                   ▷ i.e., σ = (σ_1, ..., σ_k ⊘ ψ, ..., σ_n)

        case 'when φ then τ':                              ▷ conditional
            if EVAL_QUERY(φ) then                          ▷ if the query is true in σ, then ...
                executed ←APPLY_MST(τ)                     ▷ ... apply τ
            end if

        case 'τ_1; τ_2':                                   ▷ non-deterministic choice
            i, j ←RANDOM({1,2})                            ▷ randomly assign i, j ∈ {1, 2} s.t. i ≠ i
            executed ←APPLY_MST(τ_i)                       ▷ try the first choice
            if executed = ⊥ then                           ▷ if the first choice yielded no update...
                executed ←APPLY_MST(τ_j)                   ▷ ... try the second one
            end if

        case 'τ_1, τ_2':                                   ▷ sequence
            executed ←APPLY_MST(τ_1)                       ▷ apply the first sequence member
            if executed ≠ × then                           ▷ if it didn't yield exit...
                executed ← executed ∧ APPLY_MST(τ_2)       ▷ ... apply also the second
            end if

        case '{τ'}':                                       ▷ block
            executed ←APPLY_MST(τ')                        ▷ simply apply τ'

        case 'nop':                                        ▷ skip mst
            executed ← ⊤                                   ▷ nop yields no operation

        case 'exit':                                       ▷ interpreter exit
            executed ← ×                                   ▷ returning × terminates the interpreter's cycle

    end switch

    return executed
end function
```

Algorithm 5.2 Pseudocode of the deliberation cycle of the *Jazzyk* interpreter.
input: *Jazzyk BSM* agent $\mathcal{A} = (\mathcal{M}_1, \ldots, \mathcal{M}_n, \mathcal{P})$

```
procedure JAZZYK_INTERPRETER(A = (M_1, ..., M_n, P))
    LOAD(M_1, ..., M_n)                          ▷ load agent's KR modules
    INITIALIZE(M_1, ..., M_n)                    ▷ execute initialize notifications

    repeat                                       ▷ interpreter: agent's deliberation cycle
        step_done=APPLY_MST(P))                  ▷ single BSM step
    until step_done = ×                          ▷ exit was invoked

    FINALIZE(M_1, ..., M_n)                      ▷ execute finalize notifications
    UNLOAD(M_1, ..., M_n)                        ▷ unload agent's KR modules
end procedure
```

semantics provided by Definition 2.11.

Proposition 5.2 (*Jazzyk* vs. *BSM*). *Let $\mathcal{A} = (\mathcal{M}_1, \ldots, \mathcal{M}_n, \mathcal{P})$ be a BSM and $\mathcal{A}_J$ the corresponding Jazzyk agent.*

The function APPLY_MST($\mathcal{P}$) *interpreting the agent $\mathcal{A}_J$ executes a primitive mst* `'update`$_{\oslash}$ `module`$_{\mathcal{M}_i}$ `[{`ψ`}]'` *in a state σ and returns $\top$ if and only if the BSM $\mathcal{A}$ induces a labelled transition $\sigma \overset{\oslash_i \psi}{\rightarrow} \sigma'$ for some σ'.*

Proof sketch. Informally, except for handling the newly introduced primitive mst **exit**, the core difference between the function APPLY_MST (cf. Algorithm 5.1) and the *yields* calculus (cf. Definition 2.9) is the fashion in which choice mst's are processed in order to extract primitive updates from them. While the *yields* calculus facilitates collecting of (derivation) all the updates enabled in a particular mental state, the function APPLY_MST rather searches for a *single applicable primitive update mst*. When it finds such, it immediately executes it and only indicates success or failure to do so.

I assume that the function RANDOM shuffles the choice members according to the uniform distribution, thus it is fair. Therefore, various invocations of APPLY_MST on the same mst τ in the same state σ result in eventually selecting every member of the choice mst with equal probability. I.e., in the case both yield an update, both updates will be selected for execution with a non-zero probability.

Note that the only case when a primitive update is not yielded by an agent program in a given mental state, is when it is "shielded" by a query in a higher level conditional mst it is contained in, which evaluates to `false` ($\bot$). However, when *Jazzyk* interpreter executes a primitive update mst statement, all the higher level queries "above" the statement had to evaluate to `true` ($\top$). In turn, every executed primitive update statement cannot be "shielded". In turn, when interpretation of the agent $\mathcal{A}_J$ results in execution of a

primitive update mst statement, *BSM* $\mathcal{A}$ also yielded an update corresponding to the update statement. □

5.2.3 Macro preprocessor

Jazzyk interpreter was designed to provide a lightweight modular agent-oriented programming language. Besides the *vertical* modularity, i.e., modularity w.r.t. heterogeneous KR technologies, *Jazzyk* interpreter support a *horizontal* modularity, i.e., modularity of the source code. For a robust programming language it is desirable to provide syntactical means to manipulate large pieces of code easily. Composition of larger programs from components is a vital means for avoiding getting lost in the, so called, "spaghetti code".

To support the horizontal modularity, *Jazzyk* interpreter integrates a powerful macro preprocessor *GNU M4* (M4 team, 2009). Before a *Jazzyk* program is fed to the interpretation cycle (Algorithm 5.2), its source code is processed by the *M4* preprocessor. The preprocessor expands and interprets all the *M4* specific syntactic constructs.

The language of *Jazzyk* programs is extended by the full *M4* language syntax. At the same time, the syntax of the *Jazzyk* language itself remains simple and clear, as the macro preprocessor language introduces an entirely new layer of the language. Thereby, I follow the *tiered approach* to designing a programming language (Meyer, 1990). I.e., the program execution machinery features a layered structure. The *core language interpreter* with a *compiler* integrating a *macro preprocessor*. The compiler translates programs written in an extended language into an equivalent well-formed program in the core language.

By integration of the *M4* macro preprocessor into the *Jazzyk* interpreter, this gains several important features almost "for free": definition of macros and their expansion in the source code, possibility of a limited recursive macro expansion, conditional macro expansion, possibility to create code templates, handling file inclusion in a proper operating system path settings dependent way, limited facility for handling strings and regular expressions, etc.

M4 macros are defined using the standard *GNU M4* syntax, i.e., by using the **define** meta-macro with two arguments. The first defining the macro identifier, followed by the body of the macro. The body is later expanded at places of reference. To support code templates, *M4* allows specification of parametrized macros. The arguments are referred to in a positional manner. E.g., `$3` refers to the third argument passed to the macro at the point of instantiation. Furthermore, to distinguish arguments from arbitrary strings belonging plainly to the expanded macro body, they are enclosed in quotes `‘...’` or brackets `[...]`. Furthermore, *M4* macro preprocessor allows to redefine the quote pairs.

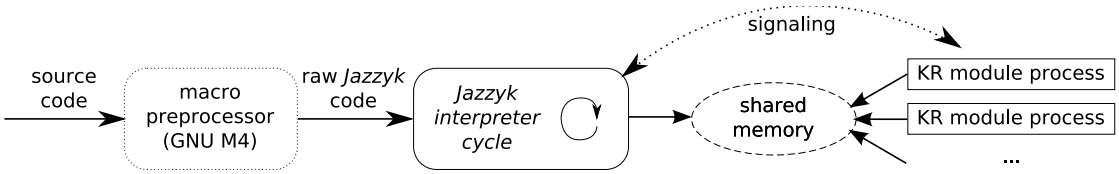


Figure 5.2: *Jazzyk* compiler-interpreter scheme.

5.2.4 Implementation

Technically, *Jazzyk* interpreter is written in *C++* as a standalone command line tool. It was implemented in a portable way, so it can be compiled, installed and relatively easily ported to most POSIX compliant operating systems. As of now, the interpreter was already ported to *Linux*, *Windows/Cygwin* and successfully compiled on *Mac OS X* platform as well. The first public release of the project was published in late 2007 under the open-source *GNU GPL v2* license and is hosted at `http://jazzyk.sourceforge.net/`, including all to date implemented KR module plug-ins and demonstration applications. As of the time of writing this dissertation, the newest published stable version of the *Jazzyk* interpreter is 1.20. It was successfully used in several applications, also described later in this dissertation (cf. Chapter 7).

To support implementation of 3rd party KR modules, I also published a KR module software development kit *Jazzyk SDK* (cf. Appendix B) including template of a trivial KR module together with all associated compile/package/deploy scripts.

The KR modules are implemented as shared dynamically loaded libraries (DLL) installed as standalone packages on a host operating system. When a KR module is loaded, the *Jazzyk* interpreter forks a separate process to host it. The communication between the *Jazzyk* interpreter and the set of the KR module subprocesses is facilitated by an OS specific shared memory subsystem. This allows loading multiple instances of the same KR module implemented in a portable way. Figure 5.2 depicts the technical architecture of the *Jazzyk* interpreter.

Finally, to simplify debugging of agent programs, *Jazzyk* interpreter implements a full-featured error reporting following the *GNU C++ Compiler* error and warning reporting format (GCC team, 2009). That allows an easier integration of the interpreter with various standardized integrated development environments (IDE) or programmers' editors, such as e.g., Eclipse, Emacs, or Vim.

Appendix A provides a brief overview of technical details of the implemented *Jazzyk* interpreter, its additional features, as well as the full listing of the included *Jazzyk* manual.

Listing 5.1 Sketch of KR module declarations and notifications for the *Jazzyk* implementation of the *Ape* robot.

```
/*** Module declarations ***/
declare module beliefs as Prolog
declare module goals as Prolog
declare module body as Java

/*** Module notifications ***/
notify beliefs on initialize [{
    %%% Fragment of Ape's belief base B (cf. Listing 3.1, left ) %%%
    ...
}]

notify goals on initialize [{
    %%% Fragment of Ape's goal base G (cf. Listing 3.1, bottom) %%%
    ...
}]

notify body on initialize [{
    /*** Fragment of Ape's body implementation E (cf. Listing 3.1, right) ***/
    ...
    body.boot();
}]

notify body on finalize [{
    body.shutdown();
}]
```

5.3 Ape in Jazzyk

Example 5.3. The sequel of listings 5.1, 5.2 and 5.3 list the agent program source code from Example 3.2 in Chapter 3 rewritten in *Jazzyk* with *M4* macros.

Listing 5.1 provides the *Jazzyk* code implementing *Ape*'s KR modules declarations and notifications. The code listed in Listing 3.1 is fed directly into corresponding KR modules' initialize notifications. Additionally, when *Ape* is to be switched off, it needs to shut down the hardware devices of its body, hence the finalization notification.

The subsequent Listing 5.2 provides an overview of the set of *M4* macros implementing parts of *Ape*'s behaviour. Macros ADOPT_GOAL, DROP_GOAL, HANDLE_GOALS and ACT implement respective parts of *Ape*'s agent program, as it was listed in Listing 3.2. Additionally, macros HANDLE_GOALS, AVOID and ACT use other macros in their bodies. In particular, HANDLE_GOALS makes use of the two fragmentary macros ADOPT_GOAL and DROP_GOAL implementing the respective goal handling techniques. Avoiding obstacles implementation AVOID instantiates TURN and STEP primitives. And finally, ACT exploits macros TURN, STEP and AVOID implementing reusable parts of *Ape*'s behaviour. Parametrized macro PERCEIVE, STEP and TURN demonstrate use of macro arguments. Since the statements for perception of atomic sensory inputs use the same pattern, the reusable macro PERCEIVE

enables for shorter agent program. The macro specifies three arguments. The first stands for the list of variables shared among the left hand and the right hand side of the conditional. The second and the third stand for the query and update formulae passed to the body module $\mathcal{E}$ and the belief base $\mathcal{B}$. Note that in the case of the belief base $\mathcal{B}$ (resp. the goal base $\mathcal{G}$), the query and update operators $\models_{\mathcal{B}}$ ($\models_{\mathcal{G}}$) and $\oplus_{\mathcal{B}}$, $\ominus_{\mathcal{B}}$ ($\oplus_{\mathcal{G}}$, $\ominus_{\mathcal{G}}$) are denoted by the identifiers query, assert and retract respectively. In the case of the interface to the *Ape*'s hardware $\mathcal{B}$ the operators $\models_{\mathcal{E}}$ and $\oslash_{\mathcal{E}}$ are denoted as sense and act respectively.

Finally, Listing 5.3 provides implementation of the agent program itself. The corresponding macros are expanded by the *Jazzyk* interpreter *in situ*. Additional macros instantiated inside higher level macros are correctly expanded as well.

The use of macros in the example improves reading of the program. Later, Chapter 6 further describes use of parametrized macros for implementation of more advanced design patterns.

5.4 Summary

This chapter provides an overview of the *Jazzyk* programming language, together with its implemented interpreter. The bulk of the chapter is mainly based on my earlier paper (Novák, 2008c).

While the framework of *Behavioural State Machines* allows for a variety of interpreter implementations, the presented one focuses on efficiency and simplicity. In the interpreter development, I followed the *tiered approach* to design a programming language. Thus, the core programming language is extensible by means of exploiting the integrated macro preprocessor. It also facilitates further source code modularity and source code reusability.

In the current incarnation the interpreter provides only basic facilities for programmer's comfort. As of the latest version 1.20, it lacks an integrated development environment (IDE) and the macro debugging facilities could be also improved, to name just a few pending improvements. In the interpreter development, I focused on the barebones functionality to enable rapid prototyping and implementation of proof-of-concept demonstration applications implementation discussed later in Chapter 7. The successful language use in several non-trivial case-studies also shows the potential of the *Jazzyk* language.

Listing 5.2 Set of reusable *M4* macros implementing parts of *Ape*'s behaviour.

```
define('PERCEIVE', 'when sense body($1) [{$2}] then assert beliefs($1) [{$3}]')

define('ADOPT_GOAL', '
        when not query goals(Place) [{goal(achieve(at(Place)))}] and
                query beliefs(Person) [{person(Person)}] and
                query body(Person,Place) [{audio.hears(Person, 'get me to ' + Place)}]
        then assert goals(Place) [{desire(achieve(at(Place)))}]
')

define('DROP_GOAL', '
        when query goals(Place) [{goal(achieve(at(Place)))}] and
                query beliefs(Place) [{at(Place, coor(X,Y,Z)), at(self, coor(X,Y,Z))}]
        then retract goals(Place) [{desire(achieve(at(Place)))}]
')

define('HANDLE_GOALS', 'ADOPT_GOAL ; DROP_GOAL')

define('STEP', 'act body [{body.forward($1);}] ')

define('TURN', 'act body($1) [{body.turn($1)}]')

define('AVOID', '
        STEP(−1) ,
        { TURN(90) ; TURN(−90) } ,
        STEP(1)
')

define('ACT', '
        when query goals(Place) [{goal(achieve(at(Place)))}] and
                not query beliefs(Place,X,Y,Z) [{at(Place, coor(X,Y,Z)), at(self, coor(X,Y,Z))}] and
                query beliefs(Place,X,Y,Z,Angle) [{at(Place, coor(X,Y,Z)), angleTo(coor(X,Y,Z), Angle)}]
        then {
                        when not (sense body(ObjectID) [{camera.see(ObjectID)}] and
                                sense body(ObjectID) [{camera.recognize(ObjectID, obstacle)}] )
                        then { TURN(Angle) , STEP(3) }
                        else AVOID
        }
')
```

Listing 5.3 *Ape*'s agent program (root level mst) implementation.

```
PERCEIVE('ObjectID', 'camera.see(ObjectID)', 'object(ObjectID)') ;
PERCEIVE('ObjectID', 'camera.recognize(ObjectID, human)', 'person(ObjectID)') ;
PERCEIVE('X,Y,Z', 'gps.location(X,Y,Z)', 'at(me, coor(X,Y,Z))') ;

HANDLE_GOALS ;

ACT
```

Chapter 6

BSM design patterns: commitment-oriented programming

In the previous chapter, I introduced *Jazzyk*, a simple programming language for programming reactive systems integrating heterogeneous knowledge bases in a single agent system. In order to evaluate whether a proposed programming language or a framework is of any practical use, the question of an *application methodology* is of a major importance. In the case of the *BSM* framework and in turn also the *Jazzyk* language, we are interested in the following question:

How to efficiently use the programming framework to build BDI-style cognitive agents?

Moreover, we should be interested in such methods and guidelines, which encompass a range of application domains. I.e., such, which can help us to design and implement BDI-inspired agents ranging from embodied robots, through pure software personal assistant agents to e.g., embedded agents of intelligent buildings.

Mainstream programming languages such as *Java*, *Pascal*, or *C++* support developers by providing guidelines for developing systems in two important ways. On one hand, they feature *powerful abstractions*, such as objects, classes, procedures, or functions. Encoding programs in terms of such concepts facilitates more intuitive understanding of and informal reasoning about 3rd party code, without looking into deep implementation details. On the other hand, these languages facilitate development of application-domain-independent *reusable modules*, subprograms, and thus speed up design, prototyping and implementation of complex applications. To provide a similar level of support for agent programmers in *Jazzyk*, reusable subprograms must also come with a clear description of their intended functionality, i.e., a formal semantic functionality specification.

The *BSM* framework, and in turn also *Jazzyk*, already provides necessary ingredients to facilitate development of application-domain-independent subprograms and their subsequent reuse. On the one hand, the *Jazzyk* interpreter, described in Chapter 5, integrates a state-of-the-art macro preprocessor allowing implementation of meaningful *Jazzyk* subprograms as reusable macros. On the other hand, Chapter 4 introduced a temporal logic tailored for the reasoning about fragments of *BSM* and *Jazzyk* agent programs.

Previously, in Chapter 3, I only informally discussed ways *how* the theoretical framework of *Behavioural State Machines* can be used for implementation of intelligent agents, such as *Ape* (cf. Section 3.3). This chapter discusses *methodology* of development of BDI inspired cognitive agents.

Firstly, I recapitulate the naïve methodology for designing cognitive agents stemming from the example of *Ape*, discussed in previous chapters. The presented methodological discussion is a result of practical experimentation with *Jazzyk* in non-trivial case-studies, I describe later in Chapter 7.

Different applications will always have different specialized requirements on methodology and software development process. Therefore, I do not provide an ultimate recipe for programming BDI agents with *Jazzyk*. However, the successful attempts to apply the framework in practice already provide sound evidence for the pragmatic usefulness of the *BSM* framework and *Jazzyk* language.

Subsequently, on a background of re-implementing the *Ape* example in a more formal way, I try to put the naïve methodology on a more solid foundation by introducing *Jazzyk* (*BSM*) *code patterns* for BDI-style agents. The core idea behind the patterns, code templates, respects the already mentioned tiered approach to design of programming languages (cf. Chapter 5). By exploiting the integrated *M4* macro-preprocessor, the patterns are implemented as parametrized macros in the *Jazzyk* language. For each such a reusable fragment of code, I provide a crisp *DCTL** characterization of their functionality w.r.t. the parameters passed to them. The chapter culminates with the construction of code patterns implementing *commitments towards achievement* and *maintenance goals*.

Further lifting the methodology, I finally propose *commitment-oriented programming*, a novel design approach for development of agents with mental attitudes. I.e., a method for designing agent programs in terms of interdependencies between specifications of commitments towards agent's mental attitudes, such as e.g., goals. The notion of a *commitment* is for commitment-oriented programming about the same as the notions of a class and objects are in the realm of object-oriented programming. In turn, the notion of a commitment as a language construct, a blueprint for useful fragments of application code, provides also an intuitive tool for agent program decomposition, i.e., a methodological guideline.

Besides presenting the agent-oriented code patterns and the generalization towards the notion of commitment-oriented programming, one of the main contributions of this chapter is demonstration of a *novel approach to the design of agent programming languages*. The mainstream agent-oriented programming languages feature a fixed set of carefully selected first-class agent-oriented features supported by the language. I instead propose a flexible and extensible generic language for reactive systems (e.g., the *BSM* framework) which can be subsequently extended by special purpose, perhaps even application specific, programming constructs. Unlike the traditional approach, my proposal does not require customization of the language semantics and the interpreter. Thereby,

it provides a versatile tool which programmers can customize according to their particular needs. In this sense, this chapter presents the culmination of the theoretical research towards this dissertation and integrates the fragments presented in the previous chapters. The remaining chapters provide additional substance to the claim of usefulness of *Jazzyk*, as well as provide some extensions stemming from practical experiences with using *Jazzyk* for non-trivial applications.

6.1 Ape example revisited: naïve methodology

In Example 3.2, I introduce a fragment of *Ape*'s *BSM* program. Subsequently, Example 5.3 shows the same example rewritten with more advanced techniques employing the *Jazzyk*'s macro preprocessing facility. Until now, I did not thoroughly discuss the development process leading to the example program. However, the *Jazzyk* re-implementation of *Ape* provides clues about the basic construction and hierarchical decomposition of agent programs.

Ape's agent program in Example 5.3 roughly implements the classic agent deliberation cycle *perceive-reason-act* (cf. e.g. (Russell and Norvig, 2002)). In fact, without loss of generality, the presented code rather implements a modified reasoning cycle, which can summarized by the statement *"In a single step, either perceive, reason about goals, or act."* The standard *perceive-reason-act* sequel is reflected in the implemented program by the flow of information in the agent system as discussed in Subsection 3.2.2. I.e., the information read from the environment is reflected in the agent's belief base, subsequently the agent uses its beliefs to modify its goals, which finally trigger actions in the environment again. In turn, these can cause events the agent might later perceive again.

From this perspective, the simple BDI architecture of *Ape* robot includes only a single purely internal mental attitude, its *goals*. Because beliefs reflect the agent's perceptions and possibly derived knowledge about the world, the belief base is in a way subordinate to input from agent's sensors, i.e., the environment. Goals play a central role in analytical decomposition of an agent system into an implemented program. This observation stands in accord with the state-of-the-art literature on agent-oriented software engineering, embodied in methodologies such as e.g., *Tropos* or *MaSE* (for a survey, cf. e.g., (Bergenti et al., 2004)).

In the joint paper with Köster (Novák and Köster, 2008), we summarize the naïve, goal-centric, methodology used for designing BDI-style agent systems similar to *Ape*. It can be summarized into the following sequel of steps:

1. Choose the *internal architecture* of the agent. Assuming the KR module interfacing the agent with its body/environment is fixed, the developer should decide on the number and type of agent's knowledge bases implementing its belief and goal bases.

2. Identify the *set of agent's goals* and implement the internals of the goal base.

I.e., encoding of interactions between the goals, subgoals and possibly even tasks, w.r.t. the employed chosen KR technology.

3. Design the *set of behaviours* τ_{act} supposed to achieve those goals. The behaviours are triggered by their corresponding goal base conditions.

4. Identify the *adoption and satisfaction conditions* for these goals and design their concrete respective commitment strategies as an mst τ_{cs}.

5. Identify the relevant part of *agent's beliefs* w.r.t. the conditions associated with the goals and subsequently design the agent's belief base including appropriate relationships and interactions among the beliefs w.r.t. the employed KR technology.

6. Design the *model of perception* τ_{perc} by identifying the sensory information, percepts, of the agent and link them to assertions or retractions of the corresponding beliefs.

7. Finally, construct the *root level agent program* by appropriately structuring and combining the mental state transformers τ_{perc}, τ_{cs} and τ_{act} into a control cycle, the *agent's reasoning cycle*.

Note that in essence the informal methodological steps presented above also respect the flow of information in the agent system. Agent's goals are the central element of any BDI system. After their identification, the rest of the system implementation can follow relatively straightforwardly. On one side, goals are manipulated only when the agent has a reason to do so, i.e., believes it is necessary. On the other, its actions are purposeful, i.e., triggered by the goals.

6.2 Jazzyk BSM code patterns

Ape's re-implementation in *Jazzyk* listed in listings of Example 5.3 already provides hints about macros or code patterns which can support the naïve methodology for development with *BSM*. Below, I loosely follow the steps of the methodology and formally introduce a set of code patterns supporting development of agents based on the behavioural template of the modular BDI architecture, as introduced in Chapter 3. The patterns allow encoding of agent's functionality in terms of a web of interdependencies between agent's beliefs, goals and behaviours, without being bound to a specific KR language of the underlying KR modules. The main result of this section is introduction of code templates formalizing the notions of *achievement* and *maintenance* goals.

Furthermore, for each of the introduced patterns, I provide also a formal characterization of its functionality in the form of a *DCTL** specification. The specifications almost immediately follow from the compositionality of *DCTL** formulae (cf. aggregation and

verification of *DCTL** formulae in Definition 4.9 and Proposition 4.13), therefore I do not provide their formal proofs.

The connection between the application-domain-independent and reusable code templates for BDI-style agents and their respective semantic characterizations provides one of the main contributions of this dissertation. Discussion of the individual patterns is accompanied by fragments of *Ape*'s program, the dissertation's running example.

Strict reliance on proper quoting and positional notation of parameters in the original *M4* syntax would only clutter the following discourse. Therefore, for the sake of readability, in the remainder of this chapter, I deviate from the original *Jazzyk* macro syntax. Instead, I use the following, simplified, syntax for macro definitions:

```
define <macro−identifier>(<parameters>)
        ...
        /* body */
        ...
end
```

The syntax resembles function specification in structured programming languages. After the macro definition keyword **define**, the macro identifier follows in front of the macro body, together with an optional list of macro's parameters. The definition ends with the keyword **end**. Whenever the context is clear, I also omit the quotes around the macro parameters. They will be referred to by their corresponding argument identifiers declared in the list of parameters.

Furthermore, to emphasize the modular nature of the introduced code templates, I use a notation mixing *Jazzyk* with the, rather mathematical, syntax of the *BSM* framework. I.e., the *Jazzyk* syntax is interleaved with query and update formulae written as $\models_{\mathcal{B}}$ or $\oplus_{\mathcal{G}}$ (cf. Definition 3.1). The concrete identifiers of declared KR modules, together with their query and update operators used in patterns, remain abstract and can be instantiates in the particular behavioural template.

6.2.1 Reusable capabilities

Agents act in an environment by executing primitive updates of the KR module representing the interface to their actuators. Their physical abilities are therefore determined by the range of well formed formulae in the corresponding KR language, together with the set of update operators of the module. *Capabilities* are mst's constructed from this universe. Primitive or compound, they encapsulate standalone, meaningful and reusable behaviours of the agent. To enable their parametrization and thereby finer grained control of primitive actions, they can be possibly conditioned by agent's beliefs. I.e., they can include conditional mst's constructed from lower level mst's of the form $Q_{\mathcal{B}} \longrightarrow U_{\mathcal{E}}$, $Q_{\mathcal{E}} \longrightarrow U_{\mathcal{E}}$ or plain updates $U_{\mathcal{E}}$.

In order to enable formal treatment of agent's reusable capabilities in larger agent programs, in Chapter 4, I introduce program annotations of primitive query and update formulae. Recall, the formal annotation function $\mathfrak{A} : (\mathcal{U}(\mathcal{A}) \cup \mathcal{Q}(\mathcal{A})) \rightarrow LTL$, defined

in Definition 4.8. Furthermore, semantic characterizations of compound mst's can be extracted by annotation aggregation as provided by Definition 4.9.

It is easy to see that the amount of effort to annotate every single primitive query and update formula occurring in an agent program does not scale up well with larger programs. Therefore, it is easier to simply extend the possibility for a programmer to annotate also higher level mst's and subsequently only *assume* soundness of such annotations (cf. Definition 4.10). This allows programmers to flexibly *choose* the level of abstraction from which he or she wants to formally treat the resulting agent program.

Since the soundness of higher level annotation is only presumed, the benevolent approach comes with a threat that the provided annotations of capabilities are actually incorrect w.r.t. the primitives included in them. I.e., the higher level annotation actually couldn't be derived, aggregated, from the lower level ones. I believe, such concerns, however, actually belong to the nature of program design process and can be tackled by standard software engineering techniques, such as e.g., unit testing, etc.

Example 6.1 (*Ape*'s capabilities). In Example 5.3, *Ape*'s program includes several basic capabilities, implemented as macros STEP, TURN and AVOID. Additionally, the corresponding annotation function of the primitive query formulae and primitive mst's can be found in Table 4.1.

Ape is a rather complex system capable of non-trivial interaction with its environment. According to the introductory story from Chapter 1, he is also able to search for people who seem to be in a need of a help. He monitors the energy level of its own batteries and in the case of a need, charge them at the base station. In order to retrieve information about various facilities at the airport he can also communicate with other agents.

To facilitate such a range of behaviours, without considering the deeper implementation issues in detail, Table 6.1 lists some of the capabilities and more complex behaviours enabling such activities. Furthermore, these come with the corresponding characterizations providing descriptions of their intended functionality.

Ape's basic capabilities include moving straight along its current orientation axis (STEPS), turning (TURN), waiting for an approaching person (WAIT), initiating the battery charging when at the home base (CHARGE) and communicating with other agents via the local airport network (INFORM). Furthermode, their combinations, together with conditions on robot's beliefs yield more complex capabilities, such as avoiding obstacles (AVOID), leading a person to a destination (LEAD), homing to the charging base (HOME), solving requests (SOLVE) and replying to the requester (REPLY).

In particular, AVOID capability could be implemented as a combination of STEPS and TURN, together with conditions relying on *Ape*'s beliefs about the airport topology and his actual sensor inputs. The capability τ_{to_dest}, introduced in Example 4.14 together with the corresponding annotation, implements a basic capability of *Ape* for moving towards a destination. It could further be exploited by LEAD capability, together with AVOID and WAIT behaviours and the information about the destination where the guided passenger

capability τ		$\mathfrak{A}(\tau) \Rightarrow \ldots$
STEPS(Steps)	$\mapsto$	$(at(CurrPos) \wedge \neg see(obstacle)) \rightarrow$ $[\mathsf{STEPS}(Steps)]\Diamond(at(Pos) \wedge \Delta(CurrPos, Pos, Steps))$
TURN(Angle)	$\mapsto$	$azimuth(CurrAngle) \rightarrow$ $[\mathsf{TURN}(Angle)] \bigcirc azimuth(CurrAngle + Angle)$
AVOID	$\mapsto$	$see(obstacle) \rightarrow [\mathsf{AVOID}^*]\Diamond \neg see(obstacle)$
WAIT(Person)	$\mapsto$	$at(Person, PPos) \wedge at(MyPos) \wedge \Delta(PPos, MyPos, 10) \rightarrow$ $[\mathsf{WAIT}(Person)^*]\Diamond at(Person, MyPos)$
LEAD(Person,Place)	$\mapsto$	$[\mathsf{LEAD}(Person, Place)^*]\Diamond(at(Person, Place)$
CHARGE	$\mapsto$	$at(homebase) \rightarrow [\mathsf{CHARGE}]\Diamond charged$
HOME	$\mapsto$	$low\_battery \rightarrow [\mathsf{HOME}^*]\Diamond charged$
INFORM(Receiver, Msg)	$\mapsto$	$[\mathsf{INFORM}(Receiver, Msg)] \bigcirc told(Receiver, Msg)$
SOLVE(Request)	$\mapsto$	$\neg knows(Request) \rightarrow [\mathsf{SOLVE}(Request)^*]\Diamond(knows(Request)$
REPLY(Receiver,Request)	$\mapsto$	$knows(Request) \wedge answer(Request, Answer) \rightarrow$ $[\mathsf{REPLY}(Receiver, Request)] \bigcirc told(Receiver, Msg)$

Table 6.1: Some of *Ape*'s basic capabilities, together with their corresponding semantic characterizations. The helping predicate $\Delta(Pos_1, Pos_2, Steps)$ is true when $|Pos_1 - Pos_2| > Steps$.

wants to get to. Similarly to LEAD, the macro HOME for returning to the charging station would also exploit capabilities for moving to a certain destination. Finally, *Ape*'s social behaviours SOLVE and REPLY rely on macros for sending information to a receiver INFORM, as well as *Ape*'s generic aural sensory abilities. REPLY capability would amount to merely communicating the already known answer regarding the request to the receiver. Solving requests could range from simply retrieving the information from *Ape*'s knowledge bases, to rather complex behaviours including e.g., yellow pages look-up of agents capable of solving the request, communicating with them, etc.

For the sake of brevity, in this chapter's running example I abstract from deeper details of *Ape*'s functionality. My intention is rather to sketch a credible and at the same time reasonably detailed demonstration of code patterns utilization. Note also that the logic

*DCTL**, as defined in Chapter 4, is based on plain propositional logic formulae. Thus, the assumptions about semantic characterizations of the macros with variables listed in Table 6.1 should be understood as formulae schemata instantiated for each specific instance of a tuple of macro arguments.

6.2.2 Perception

To enable reasoning about the environment and reacting to its changes, an agent must constantly perceive its environment and store the retrieved information in its belief base. The state of the belief base reflects the current state of the world from a subjective point of view of the agent. As already noted in Chapter 3, perception can be encoded in *BSM*s implementing the modular BDI architecture as rules of the form $Q_{\mathcal{E}} \longrightarrow U_{\mathcal{B}}$. I.e., successful information retrieval from the agent's sensory interface yields the corresponding update of its belief base. The following code pattern implements the domain-independent macro for agent's perception.

Definition 6.2 (PERCEIVE). Given a query formula $\varphi_{\mathbf{E}} \in \mathcal{L}_{\mathcal{E}}$ evaluated on agent's body module, together with the corresponding belief base update formula $\psi_{\mathbf{B}} \in \mathcal{L}_{\mathcal{B}}$, the macro implementing agent's *perception* straightforwardly follows

```
define PERCEIVE(φ_E, ψ_B)
        when ⊨_E [{φ_E}] then ⊕_B [{ψ_B}]
end
```

The macro PERCEIVE fulfills the following property:

$$\mathfrak{A}(\models_{\mathcal{E}}\varphi_{\mathbf{E}}) \rightarrow [\mathsf{PERCEIVE}(\varphi_{\mathbf{E}}, \psi_{\mathbf{B}})] \bigcirc \mathfrak{A}(\oplus_{\mathcal{B}}\psi_{\mathbf{B}}) \tag{6.1}$$

To support a more sophisticated implementation of perception mechanisms, consider the following more abstract version of the PERCEIVE pattern.

Definition 6.3 (PERCEIVEX). Let's assume that ϕ is a possibly compound query formula constructed from the set of primitive body interface queries $Q_{\mathcal{E}}$, together with primitive belief conditions $Q_{\mathcal{B}}$. I.e., ϕ does not contain a primitive subformula of the form $\models_{\mathcal{G}}\varphi$. Let also τ be a, possibly compound, mst modifying only the belief base of the agent, i.e., τ does not contain (inductively in the whole mst decision tree) an mst of the form $\oslash_{\mathcal{E}}\psi$ or $\oslash_{\mathcal{G}}\psi$. The following macro implements complex perception

```
define PERCEIVEX(φ, τ)
        when φ then τ
end
```

Straightforwardly, the macro PERCEIVEX fulfills the following property:

$$\mathfrak{A}(\phi) \rightarrow [\mathsf{PERCEIVEX}(\phi, \tau)^*]\Diamond\mathfrak{A}(\tau) \tag{6.2}$$

Both macros PERCEIVE and PERCEIVEX are implemented as simple conditional mst's. Their intended functionality is derived solely from the syntactic conditions imposed on them by the corresponding definitions. In a real implementation, it would be difficult and probably even useless to explicitly check whether the macros' parameters satisfy the conditions imposed on them. However, it still makes sense to use such code templates as a means to indicate the design intention behind the implemented subprogram.

Example 6.4 (*Ape*'s perception). In order to be able to deliberate about its internal mental attitudes and subsequently take actions in the environment, *Ape* must first be able to perceive the current state of the world. The following instantiations of the PERCEIVE pattern allow *Ape* to capture the relationship between elements of the physical reality and their belief counterparts.

```
PERCEIVE('camera.see(ObjectID)', 'object(ObjectID)')
PERCEIVE('camera.recognize(ObjectID, human)', 'person(ObjectID)')
PERCEIVE('gps.location(X,Y,Z)', 'at(me, coor(X,Y,Z))')
PERCEIVE('body.getBattery(Level)', 'energy(Level)')
PERCEIVE('body.hears(ObjectId, Msg)', 'says(ObjectID, Msg)')
```

Such elementary perception statements can be aggregated into higher level macros implementing compound perceptions. Such compounds take care for perceiving larger aspects of the environment dynamics. The following macros demonstrate this simple idea.

```
define PERCEIVE_EYE
        PERCEIVE('camera.see(ObjectID)', 'object(ObjectID)') ;
        PERCEIVE('camera.recognize(ObjectID, human)', 'person(ObjectID)')
end

define PERCEIVE_LOCATION
        PERCEIVE('gps.location(X,Y,Z)', 'at(me, coor(X,Y,Z))') ;
        PERCEIVE('camera.distance(ObjectID, Dist)', 'distance(ObjectID, Dist)')
end
```

The mst PERCEIVE_EYE joins the perception statements by the non-deterministic choice operator. In a single step the agent can either sense objects it can see, or perform object and face recognition. Compound perception of one, right after another, would be achieved by joining the two statements by the sequence operator.

Note also that while the template PERCEIVEX can be directly implemented as a self contained piece of *Jazzyk* code, the definition of the simpler PERCEIVE macro is not self contained, because it lacks specification of variables. For the sake of readability, I omit treatment of variables also in the remainder of this chapter. As I already noted in the previous chapter, the integrated *GNU M4* macro preprocessor includes basic string manipulation API. In principle, it is therefore possible to implement an automatic mechanism for recognizing variables directly from query and update formulae and then passing them to the individual module invocations. The macro PERCEIVEX is immune to this issue, since it takes a complete compound query, update, or mst statements as arguments.

6.2.3 Goal-oriented behaviours

The problem of designing a *BSM* agent performing a specified range of behaviours, can then be seen as the problem of managing activation, deactivation and interleaving of the capabilities encoded as *BSM* mental state transformers. In each step of its execution, the agent performs a *selection of a reactive behaviour* to execute. To allow for an explicit behaviour management, i.e., activation and deactivation of agent's capabilities, each capability mst should be triggered only when appropriate. Thus, behaviour activation becomes purposeful. Behaviour is triggered because an agent has a goal and the behaviour is supposed to take the agent closer to the achieving it. In the remainder of this section, I argue that the notion of a *commitment towards a goal* provides a technical basis for management and programming of behaviour selection.

The explicit representation of a goal is a formula derived from the agent's goal base. The following code pattern implements agent's capability τ triggered by derivation of the goal formula $\varphi_{\mathbf{G}}$ from agent's goal base.

Definition 6.5 (TRIGGER). Let $\varphi_{\mathbf{G}} \in \mathcal{L}_{\mathcal{G}}$ be a goal formula and τ be an mst satisfying the following independence condition

$$[\tau]\mathfrak{A}(\tau) \Rightarrow (\mathfrak{A}(\models_{\mathcal{G}}\varphi_{\mathbf{G}}) \rightarrow [\tau^*]\Box\mathfrak{A}(\models_{\mathcal{G}}\varphi_{\mathbf{G}})) \tag{6.3}$$

I.e., iterated execution of τ does not change the derivability of the associated goal formula $\varphi_{\mathbf{G}}$ from the agent's goal base.

Then, the following code template implements triggering of the *goal oriented behaviour* τ according to derivability of the corresponding goal formula $\varphi_{\mathbf{G}}$.

```
define TRIGGER(φ_G, τ)
        when ⊨_G φ_G then τ
end
```

Straightforwardly, when an agent has a goal $\varphi_{\mathbf{G}}$, iterated execution of TRIGGER always eventually leads to satisfaction of the annotation of τ. I.e.,

$$[\tau]\mathfrak{A}(\tau) \Rightarrow (\mathfrak{A}(\models_{\mathcal{G}}\varphi_{\mathbf{G}}) \rightarrow [\mathsf{TRIGGER}(\varphi_{\mathbf{G}}, \tau)^*]\Diamond\mathfrak{A}(\tau)) \tag{6.4}$$

The TRIGGER code pattern allows conditional activation and deactivation of the capability τ, depending on the derivability of the condition $\varphi_{\mathbf{G}}$ from the agent's goal base.

Similarly to the PERCEIVE pattern, even though the TRIGGER template takes the form of a simple conditional mst, by its use, programmers can indicate the intended functionality of the code chunk.

Example 6.6 (triggering *Ape*'s behaviours). To assist passengers at the airport, *Ape* performs a variety of behaviours, basic capabilities (cf. Example 6.1), implementing aspects of its functionality. These capabilities are purposeful, i.e., *Ape* executes them in

order to achieve a goal, to perform a task, or to maintain a certain state of affairs. Binding execution of the capabilities to their corresponding purposes, goals, is demonstrated by following instantiations of the TRIGGER pattern.

```
TRIGGER('goal(achieve(at(Person,Place)))', 'LEAD(Person,Place)')
TRIGGER('goal(achieve(inform(Person,Request)))', 'SOLVE(Request)')
TRIGGER('goal(achieve(inform(Person,Request)))', 'REPLY(Person,Request)')
TRIGGER('goal(maintain(energy))', 'HOME')
```

I.e., whenever one of *Ape*'s goals is to guide a person to a certain place, he can execute the behaviour LEAD to achieve this goal. Similarly, the two behaviours SOLVE and REPLY are enabled for execution, when *Ape* is about to provide information to some person, possibly on her or his request. Finally, the execution of the homing behaviour HOME maintains *Ape*'s energy level sufficient for further functioning.

Note that the two behaviours SOLVE and REPLY in the Example 6.6 above, suitable for achieving one and the same goal: to provide an information to a person. The semantic characterizations of the SOLVE behaviour, provided in Table 6.1, does not specify that its single execution directly leads to *Ape* knowing the answer to the request. I.e., only the following weaker characterization holds

$$\neg knows(Request) \rightarrow [\mathsf{SOLVE}(Request)^*]\Diamond(knows(Request).$$

In turn, instantiation of such a macro in a *BSM* shouldn't be understood as a *plan* for direct achievement of the corresponding goal, but rather as a complex behaviour *performing a step towards achieving the goal*. A behaviour τ can be considered a proper *plan* for achieving a goal $\varphi_{\mathbf{G}}$ if its semantic characterization looks similar to the following

$$[\tau]\Diamond\varphi_{\mathbf{G}},$$

or alternatively

$$[\tau]\top\, \mathcal{C}\, \Box\varphi_{\mathbf{G}}.$$

I.e., during a single execution of the behaviour τ, the goal formula eventually becomes true and possibly it stays true till the end of the behaviour execution. Then, having several behaviours $\tau_1, \ldots, \tau_n$ with such semantic characterizations, we could construct a proper, possibly non-deterministic, plan by combining them into a compound mst by using choice and sequence operators.

Example 6.7. The following compound behaviour can be considered a proper plan.

```
define AVOID_PLAN
        STEP(−3) ,
        { TURN(60) ; TURN(−60) } ,
        STEP(5)
end
```

The complex behaviour implements *Ape*'s strategy for avoiding simple obstacles such as a piece of luggage on the ground and could possibly be a part of the complex behaviour AVOID. *Ape* first moves 3 steps backwards, then non-deterministically chooses a direction to avoid the obstacle and subsequently moves forward and thus passes the obstacle. Under circumstances (cf. below), the AVOID_PLAN behaviour can satisfy the following semantic characterization (also cf. Table 6.1).

$$see(obstacle) \rightarrow [\mathsf{AVOID\_PLAN}]\Diamond\neg see(obstacle)$$

In practice, however, more complex sequential plans can become impractical. They might lead to longer, uninterruptible executions, during which preconditions for successful plan execution might become invalid and thus, the plain can fail. In the case of the AVOID_PLAN behaviour, this can happen whenever there is another obstacle located in the way of the chosen avoidance maneuver (right or left). Moreover, extensive use of sequential behaviours goes against the reactive spirit of the *BSM* framework (cf. Chapter 1).

6.2.4 Commitment strategy primitives

Above, I introduced patterns for encoding interdependencies between the environment and agent's beliefs on one hand ($Q_{\mathcal{E}} \longrightarrow U_{\mathcal{B}}$), and between agent's goals and its actions in the environment on the other ($Q_{\mathcal{B}} \longrightarrow U_{\mathcal{E}}$). What remains to complete the information flow cycle in the modular BDI architecture (cf. Chapter 3), is to provide simple code templates for encoding relationships between agent's beliefs and the goals it pursues ($Q_{\mathcal{B}} \longrightarrow U_{\mathcal{G}}$).

Explicit goal representation allows conditional activation and deactivation of behaviours. However, to directly manage when exactly such a goal formula should be derivable, programmer must choose an explicit algorithm, *a goal commitment strategy*.

A goal commitment strategy explicitly encodes the *reasons* for a goal adoption as well as its dropping. When an agent believes it can adopt a goal, it should also add its explicit representation, a goal formula, to its goal base. Similarly, when it believes the goal can be dropped, it should remove the corresponding formula from the goal base. The following two code patterns provide a toolbox for encoding an appropriate commitment strategy for a given goal formula.

Definition 6.8 (ADOPT and DROP). Let $\varphi_{\mathbf{G}} \in \mathcal{L}_{\mathcal{G}}$ be a goal formula and $\psi_{\oplus}, \psi_{\ominus} \in \mathcal{L}_{\mathcal{B}}$ be formulae derivable from agent's belief base. Macros ADOPT and DROP, implementing respectively *goal adoption* and *goal dropping* are defined as follows.

```
define ADOPT(φ_G, ψ_⊕)
        when ⊨_B ψ_⊕ and not ⊨_G φ_G then ⊕_G [{ φ_G }]
end

define DROP(φ_G, ψ_⊖)
        when ⊨_B ψ_⊖ and ⊨_G φ_G then ⊖_G [{ φ_G }]
end
```

For the ADOPT and DROP macros we can formulate the properties below. That is, the following formulae are valid in every annotated *BSM*.

$$\begin{aligned} \mathfrak{A}(\models_{\mathcal{B}} \psi_{\oplus}) &\rightarrow [\mathsf{ADOPT}(\varphi_{\mathbf{G}}, \psi_{\oplus})^*]\Diamond \mathfrak{A}(\models_{\mathcal{G}} \varphi_{\mathbf{G}})) \\ \mathfrak{A}(\models_{\mathcal{B}} \psi_{\ominus}) &\rightarrow [\mathsf{DROP}(\varphi_{\mathbf{G}}, \psi_{\ominus})^*]\Diamond \neg \mathfrak{A}(\models_{\mathcal{G}} \varphi_{\mathbf{G}})) \end{aligned} \tag{6.5}$$

Note the use of behaviour iterations in the pattern characterizations. Stricter versions, employing the $\bigcirc$ modality instead of $\Diamond$, could be formulated for non-iterated executions of the two patterns. However, as will become clear in the next subsections, the weaker iterated variant will serve the purpose better.

Example 6.9 (*Ape*'s goal commitment strategies). According to Example 6.6, *Ape* executes its basic capabilities to reach its goals. To explicitly manage activation and deactivation of the behaviours, *Ape*'s programmer must encode the dynamics of the goals themselves. This can be achieved by appropriately instantiating the ADOPT and DROP patterns as follows.

```
ADOPT('goal(achieve(at(Person,Place)))', 'wants(Passenger,at(Place))')
DROP('goal(achieve(at(Person,Place)))', 'at(Person,Place)')
DROP('goal(achieve(at(Person,Place)))', 'refused(Person)')

ADOPT('goal(achieve(inform(Person,Request)))', 'asks(Person,Request)')
DROP('goal(achieve(inform(Person,Request)))','answer(Request,Msg),told(Person,Msg)')

ADOPT('goal(maintain(energy))', 'true')
```

Ape adopts the goal to guide a passenger to a particular destination at the airport, when he is asked to do so. The goal is considered to be satisfied when the passenger finally arrives to the requested place. Furthermore, the goal is considered unachievable, whenever the person for some reason refuses *Ape*'s help or leaves. Similarly, *Ape* adopts the goal to provide an answer to a given request whenever a passenger asks him a question and the goal is is dropped after *Ape* finds out the answer and informs the passenger. Finally, there is no condition triggering the goal to maintain a sufficient energy level of batteries. Hence, *Ape* keeps this goal throughout his whole lifetime.

Together with the associated commitment strategy, a goal oriented behaviour implements a commitment towards a goal, a goal pattern. Instantiations of the ADOPT and DROP patterns serve as primitives for encoding the exact commitment strategy. In turn, the goal formula becomes an explicit representation, a mere placeholder, for the goal. In the following two subsections, I introduce two code patterns for particularly useful goal types, the *achievement goal* and the *maintenance goal*.

6.2.5 Achievement goal

The notion of an *achievement goal* is one of the central constructs of agent-oriented programming. Provided, an agent does not believe that a goal satisfaction condition is

true in a given point of time, adopting an achievement goal specifies that the agent desires to eventually make it true. After having satisfied the goal, it can be dropped. If the agent believes the goal is unachievable, it can retract its commitment to it. Additionally, after reaching the goal, the agent must be capable of recognizing that it was indeed satisfied. The following code pattern implements the commitment towards an achievement goal.

Definition 6.10 (ACHIEVE). Let $\varphi_{\mathbf{G}} \in \mathcal{L}_{\mathcal{G}}$ be an agent's goal formula and $\varphi_{\mathbf{B}}, \varphi_{\oplus}$, $\varphi_{\ominus} \in \mathcal{L}_{\mathcal{B}}$ be the corresponding satisfaction, adoption and drop conditions derivable from the agent's belief base. Let also τ be a capability triggered by $\varphi_{\mathbf{G}}$. Finally, let τ_p be a possibly complex perception behaviour constructed from instantiations of the PERCEIVE, resp. PERCEIVEX pattern. The following pattern implements the notion of a *commitment towards an achievement goal*, constructed from these ingredients.

define ACHIEVE($\varphi_{\mathbf{G}}$, $\varphi_{\mathbf{B}}$, $\psi_{\oplus}$, $\psi_{\ominus}$, τ_p, τ)
    τ_p ;
    TRIGGER($\varphi_{\mathbf{G}}$, τ) ;
    ADOPT($\varphi_{\mathbf{G}}$, $\psi_{\oplus}$) ;
    DROP($\varphi_{\mathbf{G}}$, $\varphi_{\mathbf{B}}$) ;
    DROP($\varphi_{\mathbf{G}}$, $\psi_{\ominus}$)
end

Let $\varphi_{\mathbf{E}} \in \mathcal{L}_{\mathcal{E}}$ be a perception formula. Let's assume that τ is a behaviour causing $\varphi_{\mathbf{E}}$, i.e.,

$$[\tau^*]\Diamond\Box\mathfrak{A}(\models_{\mathcal{E}}\varphi_{\mathbf{E}}) \tag{6.6}$$

Let's also assume that τ_p is a perception behaviour, which allows to recognize validity of $\varphi_{\mathbf{E}}$ and reflect it in the agent's belief base. Formally,

$$\mathfrak{A}(\models_{\mathcal{E}}\varphi_{\mathbf{E}}) \rightarrow [\tau_p]\Diamond\mathfrak{A}(\models_{\mathcal{B}}\varphi_{\mathbf{B}}) \tag{6.7}$$

We can formulate the following semantic characterization of the ACHIEVE macro

$$\mathfrak{A}(\models_{\mathcal{G}}\varphi_{\mathbf{G}}) \rightarrow [\mathsf{ACHIEVE}(\varphi_{\mathbf{G}}, \varphi_{\mathbf{B}}, \psi_{\oplus}, \psi_{\ominus}, \tau_p, \tau)^*]\Diamond\big((\mathfrak{A}(\models_{\mathcal{B}}\varphi_{\mathbf{B}}) \vee \mathfrak{A}(\models_{\mathcal{B}}\varphi_{\ominus}))\,\mathcal{U}\,\neg\mathfrak{A}(\models_{\mathcal{G}}\varphi_{\mathbf{G}})\big) \tag{6.8}$$

The code template for the commitment towards an achievement goal combines a goal oriented behaviour, with a corresponding commitment strategy and perception. In a single execution step, either the agent perceives, handles the goal commitment, or performs the behaviour for its achievement. If the triggered behaviour is indeed "appropriate" w.r.t. the considered goal (6.6) and the agent is able to recognize that the goal is satisfied (6.7), iterated execution of τ eventually leads to the goal satisfaction and its subsequent elimination. The goal can of course be dropped also prematurely, whenever the agent recognizes it as unachievable, or irrelevant, i.e., the agent starts to believe that $\varphi_{\ominus}$ holds. This, however, can be only caused by execution of other, not directly related, behaviours.

For the sake of simplicity, the condition 6.6 is rather strong. It requires τ to be such a behaviour that its iterated execution can make $\varphi_{\mathbf{E}}$ true and afterwards keep it valid

forever. Actually, it would suffice if $\varphi_{\mathbf{E}}$ holds sometimes at the end of execution of τ, after which the perception mst τ_p is executed. Note also that the formula requires only execution of τ. It might well be true that when other behaviours are executed, they can make $\varphi_{\mathbf{E}}$ eventually invalid. Here, I also neglect the dynamics of the environment, which is in general unpredictable. However, since I do not consider an explicit model of an environment in the presented setting, for the purposes of reasoning about agent program executions, I ignore the environment dynamics. It is relatively straightforward to see how the setting could be transparently extended in this direction.

Example 6.11 (*Ape*'s achievement goals). Finally, we are ready to combine bits of the puzzle from Example 6.6 and Example 6.9 and to formulate *Ape*'s achievement goals for guiding a passenger to the requested destination and for resolving a request.

```
ACHIEVE(
        'goal(achieve(at(Person,Place)))',                          /* goal formula */
        'at(Person,Place)',                                  /* satisfaction condition */
        'wants(Passenger,at(Place))',                            /* adopt condition */
        'refused(Person)',                                        /* drop condition */
        '{                                                    /* complex perception */
                PERCEIVE_LOCATION ;
                PERCEIVE('body.hears(ObjectId, Msg)', 'says(ObjectID, Msg)')
        }',
        'LEAD(Person,Place)'                                 /* triggered behaviour mst */
)

ACHIEVE(
        'goal(achieve(inform(Person,Request)))',
        'answer(Request,Msg),told(Person,Msg)',
        'asks(Person,Request)',
        'refused(Person)',
        'PERCEIVE('body.hears(ObjectId, Msg)', 'says(ObjectID, Msg)')',
        '{ SOLVE(Request) ; REPLY(Person,Request) }'
)
```

When *Ape* believes he was asked by a passenger to guide her or him to a particular destination, wants(Passenger,at(Place)) becomes true in his belief base. In reaction to that, *Ape* eventually adopts the goal goal(achieve(at(Person,Place))). The goal subsequently triggers the behaviour LEAD, designed to perform a step towards eventually getting to the destination. At the moment when *Ape* realizes they arrived to the destination, he drops the goal. *Ape* recognizes that he and the passenger arrived to the destination (at(Person,Place)) by means of performing the sensing behaviour PERCEIVE_LOCATION. The goal is dropped also when he realizes the goal is unachievable because the passenger for whatever reason turned down *Ape*'s assistance.

Similarly, in the case of the second achievement goal supposed to provide an answer to a request. Note however that the behaviour triggered by the achievement goal to reply to the passenger, goal(achieve(inform(Person,Request))), is a non-deterministic choice of a step towards solving the request and replying to the passenger. This can be done, because the semantic characterizations of SOLVE and REPLY macros are mutually exclusive w.r.t. the

truth value of *knows*(*Request*). Thus, in the case when REPLY will be performed before *Ape* even finds an answer to the request, REPLY yields no operation. We assume here that the semantic characterization of REPLY provided in the Table 6.1 is complete.

The provided implementation of a commitment towards an achievement goal implements a commitment strategy similar to that of the specification of *persistent relativized goal* P-R-GOAL, defined by Cohen and Levesque (1990) as follows.

$$\begin{aligned}(\mathsf{P{-}R{-}GOAL}\ \mathsf{x}\ \mathsf{p}\ \mathsf{q}) \stackrel{def}{=}\ & (\mathsf{GOAL}\ \mathsf{x}\ (\mathsf{LATER}\ \mathsf{p})) \wedge (\mathsf{BEL}\ \mathsf{x}\ \neg\mathsf{p}) \wedge \\ & (\mathsf{BEFORE}\ [(\mathsf{BEL}\ \mathsf{x}\ \mathsf{p}) \vee (\mathsf{BEL}\ \mathsf{x}\ \Box\neg\mathsf{p}) \vee (\mathsf{BEL}\ \mathsf{x}\ \neg\mathsf{q})] \\ & \neg(\mathsf{GOAL}\ \mathsf{x}\ (\mathsf{LATER}\ \mathsf{p}))).\end{aligned}$$

That is, whenever the agent x believes it is appropriate (q), it adopts the persistent relativized goal to eventually achieve p (LATER p). Furthermore, before dropping the goal, the agent either believes that it is achieved, or it is unachievable, or the reason for its adoption q does not hold anymore.

In the proposal above, unlike for the P−R−GOAL, in order to instantiate the macro ACHIEVE a programmer must explicitly encode the goal drop condition $\varphi_{\ominus}$ for the case when the agent believes the goal is unachievable or not needed anymore. In the basic setup of the modular BDI architecture discussed here, I do not assume introspective capabilities of an agent.

6.2.6 Maintenance goal

Another particularly useful commitment strategy towards a goal is *persistent maintenance* of a certain condition imposed on agent's beliefs. Provided a belief condition $\varphi_{\mathbf{B}}$ is an intended consequence of a capability τ, its violation should trigger the behaviour τ supposed to re-establish its validity. Moreover, in the proposal below, the maintenance efforts should never be dropped. I.e., in the case the goal cannot be derived from the goal base, it should be re-instantiated.

Definition 6.12 (MAINTAIN). Let $\varphi_{\mathbf{G}} \in \mathcal{L}_{\mathcal{G}}$ be a goal formula and $\varphi_{\mathbf{B}} \in \mathcal{L}_{\mathcal{B}}$ be the maintained belief condition. Let also τ be a behaviour capable of re-establishing the condition $\varphi_{\mathbf{B}}$. Finally, let τ_p be a possibly complex perception behaviour, capable of recognizing validity of the maintained condition in the environment. The following code pattern implements the notion of a *commitment towards a maintenance goal*.

define MAINTAIN($\varphi_{\mathbf{G}}$, $\varphi_{\mathbf{B}}$, τ_p, τ)
    τ_p ;
    when not $\models_{\mathcal{B}}\varphi_{\mathbf{B}}$ **then** TRIGGER($\varphi_{\mathbf{G}}$, τ) ;
    ADOPT($\varphi_{\mathbf{G}}$, $\top$)
end

Similarly to Definition 6.10, let $\varphi_{\mathbf{E}} \in \mathcal{L}_{\mathcal{E}}$ be a sensory input bound to the belief $\varphi_{\mathbf{B}}$. Let's also assume that the purpose of the goal oriented behaviour τ is to eventually re-establish validity of the maintained condition $\varphi_{\mathbf{B}}$ and thus maintain the goal $\varphi_{\mathbf{G}}$. I.e.,

τ and τ_p satisfy conditions 6.6 and 6.7 from Definition 6.10. The code pattern MAINTAIN satisfies the following property:

$$[\mathfrak{A}(\models_{\mathcal{G}}\varphi_{\mathbf{G}}) \rightarrow \mathsf{MAINTAIN}(\varphi_{\mathbf{G}}, \varphi_{\mathbf{B}}, \tau_p, \tau)^*](\Box(\neg\mathfrak{A}(\models_{\mathcal{B}}\varphi_{\mathbf{B}}) \rightarrow \Diamond\mathfrak{A}(\models_{\mathcal{B}}\varphi_{\mathbf{B}})) \tag{6.9}$$

It can again be the case that the execution of the behaviour τ might not bring about the validity of the maintained condition immediately. In the case the condition is violated, τ's iterated execution should eventually re-establish its validity. Moreover, when, for whatever reason, the goal $\varphi_{\mathbf{G}}$ is dropped, the macro MAINTAIN re-instantiates it in the goal base, i.e., adopts it again.

Example 6.13 (*Ape*'s maintenance goal). To complement the two achievement goals from Example 6.11 and to be able to serve airport passengers during long days, *Ape* must maintain a sufficient level of charge in his batteries.

```
MAINTAIN(
        'goal(maintain(energy))',
        '\+ low_battery',
        'PERCEIVE(
                'body.getBattery(Level)',
                'energy(Level)'
        )',
        'HOME'
)
```

Whenever *Ape* notices that the battery charge dropped below the critical threshold, i.e., the predicate low_battery becomes derivable from his belief base (cf. Listing 3.1), he enables execution of the homing and charging behaviour HOME. The perception pattern implementing the connection between the consequences of the behaviour HOME and the maintained condition is taken from Example 6.4. The condition \+ low_battery is to be read as "it is not the case that the battery is low".

6.3 Commitment-oriented programming

The *BSM* code patterns for *commitment strategies* towards an achievement and maintenance goals allow an agent programmer to clearly and concisely express relationships between individual mental attitudes of the designed agent by means of self-contained pieces of source code. In the previous section, I gradually designed a set of application-domain-independent subprograms, which are finally composed into meaningful patterns. To do so, I exploited the *compositionality* of mst's provided by the BSM framework, together with the macro preprocessor's facilities for code encapsulation and reuse.

Compositionality of programs is a crucial means for modular software engineering. As I show in this section, it allows scaling up the promoted development method towards large applications. In order to support hierarchical decomposition of agent systems, we need a powerful abstraction, allowing arbitrary nesting of constructs in a way similar

to that of e.g., object-oriented languages. An object can consists of, or refer to, other objects which, in turn, can keep further relations to other objects. A design abstraction enabling composition of constructs, so that the resulting entity can be further composed into objects of the same type, allows modular application development by possibly teams of independent developers.

The goal patterns ACHIEVE and MAINTAIN, as considered in definitions 6.10 and 6.12, do not straightforwardly employ such hierarchical nesting of arbitrary depth. Yet, it is relatively easy to observe that there is nothing standing in a way to use compound mst's consisting of other goal specifications as the behaviours triggered in order to reach a goal. I.e.,

> *Besides executing a commitment towards reaching a goal, the goal can also be satisfied by reaching other goals, i.e., subgoals.*

Lifting this observation even higher leads to an abstraction based on the notion of *commitment*. I.e., a programmer designs an agent system in terms of *a web of interrelated commitments*. I call such an agent programming style *commitment-oriented programming* (*COP*). The essential idea behind *commitment-oriented programming* is a straightforward generalization of the observation about the goals above.

> *Commitment towards a mental attitude is a behaviour describing relationships among a set of mental attitude objects, execution of which leads to a well defined change of the attitude in focus. Furthermore, the change can be often satisfied by a composition of commitments towards other related mental attitudes, subcommitments.*

Depending on the central focus of the design methodology in consideration, similar commitment definitions could be defined towards various mental attitudes, such as e.g., *goals, obligations*, *norms*, etc. The embedded compositionality allows hierarchical decomposition of higher level commitments into lower level ones, until execution of concrete low level capabilities can be specified directly. For instance, a commitment to stand up to an obligation might involve maintaining the obligation, a maintenance goal, combined with a commitment to achieve the subject of the obligation, a goal.

A particularly useful instance of this abstraction is a *commitment towards a goal* presented in the previous section. A commitment specification to a goal is defined in terms of interrelationships between involved percepts, beliefs, goals and actions, and possibly also in terms of other goals. In line with the nomenclature for *COP*, the goal-centric programming style presented above could be coined *goal oriented programming*.

To treat goals as the central element of a methodology for construction of BDI-style cognitive agents is rather characteristic to BDI inspired agent architectures. In terms of the code patterns defined in the previous section, it means that regardless of the actual nature of the goal pattern in consideration, a goal oriented behaviour τ triggered by

the goal can be constructed from agent's capabilities, as well as instances of other goal commitment specifications, such as e.g., ACHIEVE and MAINTAIN patterns. The final agent program for *Ape*'s functionality demonstrates this technique.

Example 6.14 (Finally *Ape*: putting it altogether). In the sequel of examples earlier in this section, I gradually developed subprograms implementing parts of *Ape*'s complex behaviour. namely, achievement goals for guiding a passenger to the desired location, resolving his/her request (both in Example 6.11) and for maintaining a sufficient level of energy in *Ape*'s batteries (Example 6.13). These provide some of the building blocks, out of which *Ape*'s overall functionality can be built. In the following, I assume that these goal behaviours are wrapped into higher level macros GOAL_GUIDE_PASSENGER, GOAL_RESOLVE_REQUEST and GOAL_ENERGY respectively. Furthermore, I assume similarly constructed macros GOAL_ROAM implementing the commitment towards moving to a random location in the airport hall. Finally, GOAL_APPROACH_PASSENGER implements the behaviour for approaching a passenger who seems to need assistance. The following two macros implement commitments towards the behaviour for assisting passengers and searching for new clients respectively.

```
/* General purpose goal for assisting passengers at the airport */
define GOAL_ASSIST
        ACHIEVE(
                'goal(achieve(satisfied(Person)))',
                'says(Person,''thank you'')',
                'meet(Person)',
                'departed(Person)',
                'PERCEIVE_EAR(Person)',
                '{
                        GOAL_GUIDE_PASSENGER(Person) ;
                        GOAL_RESOLVE_REQUEST(Person)
                }'
        )
end

/* General purpose goal for seeking passengers in a need for assistance */
define GOAL_SEEK
        MAINTAIN(
                'goal(maintain(helping))',
                'has_client',
                '{ PERCEIVE_OBSERVE ; PERCEIVE_PHONE }',
                '{
                        GOAL_ROAM ;
                        GOAL_APPROACH_PASSENGER
                }'
        )
end
```

In the first case GOAL_ASSIST, *Ape* has a desire to satisfy passenger's request. In order to do so, he actively listens to what the passenger says and depending on the current request, he either makes a steps towards resolving it by communicating with other airport service agents or a step towards guiding the client to the desired location. Similarly, in the second

case GOAL_SEEK, when *Ape* has nothing to do, he roams around the airport premises, identifies passengers who potentially need assistance and proactively approaches them. During this activity, he actively observes the surroundings and checks its integrated cellular phone for potential incoming requests from the central command.

Finally, since *Ape* is a service robot, an example main program for *Ape* could consist of persistently maintaining a reasonable workload. This decomposes into an infinite cycle of seeking potential clients, assisting them and maintaining the energy level at the same time.

```
/* Ape's root level agent program */
MAINTAIN(
        'goal(main_activity)',
        'false',
        '{
                GOAL_SEEK ;
                GOAL_ASSIST ;
                GOAL_ENERGY
        }'
)
```

Note that even though I describe the introduced subprograms in terms of loose *plans*, *Ape* executes, in the program listings above all the behaviours are connected by the non-deterministic choice operator. This means that *Ape*'s lower level behaviours are actually interleaved and in a single program execution cycle *Ape* always executes only one of them. I.e., he makes a step towards achieving or maintaining the corresponding goals. In the case, the lower level mst's would be joined by the sequence operator down to the primitive capabilities, *Ape*'s functionality could be interpreted in terms of proper, uninterruptible plans. However, since interruptibility of behaviours is an essential element of programming robust embodied agents, such a scheme would lead to problems with reactivity of the whole system.

Flexibility and adjustable level of abstraction

The encapsulated specifications of commitments towards goals implement standalone self-contained modules. In general, the decomposition design principle of commitment-oriented programming promotes designing higher level goals in terms of lower level ones, down to basic capabilities. On the other hand, however, this style of agent system design also allows programmer to liberally decide the *level of granularity*. Below that level, the agent program will be decomposed only into basic behaviours, capabilities. The promoted style does not enforce the all-encompassing view, such as *everything is a commitment*, rather, depending on the nature of the designed system, a designer should be free to decide what is the appropriate level of abstraction under which the agent's functionality is to be implemented as low-level, perhaps more efficient, hard-wired *Jazzyk* code. I.e., which aspects are better to be modeled in terms of commitments or similar high level notions and which should remain opaque capabilities of the agent.

In the sequel of examples in this section, behaviours such as CHARGE or AVOID were

modeled as basic capabilities used later in implementation of goals GOAL_ENERGY or GOAL_GUIDE_PASSENGER. That does not mean that in a different scenario, even such primitive behaviours couldn't be modeled as lower level of commitments composed of even more primitive commitments and basic capabilities.

6.4 Summary

Example 6.14 closes the sequel of steps following the proposed methodology for development of cognitive agents with the *BSM* framework. Besides proposing a methodology for programming cognitive agent systems with the *BSM* framework, the list below summarizes the main contributions of this chapter.

1. I gradually develop a set of high level language constructs, corresponding to complex mental attitudes of an agent, such as the notions of *commitments towards achievement* and *maintenance goal*s. The introduced high level language constructs rely on the semantics of the lower level behaviours used as their components. Descriptions of lower level behaviours provide a clear semantic characterization of the more complex, higher level, functionality.

2. Generalizing the naïve methodology of agent-oriented design presented troughout the previous chapters, I propose *commitment-oriented programming*. This programming paradigm promotes development of instance methodologies centered around a particular mental attitude. In turn, an agent program is designed in terms of a web of interdependent commitment specifications.

3. In order to enable reasoning about agent programs written in terms of high level goals, designers are free to decide the level of specificity of the *program annotations* of basic behaviours. Since these can be composed from even lower level constructions, their semantic characterization can be either derived from the characterizations of the components, or directly specified by the designer. Thereby, agent system developers are free to *adjust* the level of abstraction on which the program speaks in terms of well characterized commitments and below which the agent's functionality is implemented in terms of primitive, rather *ad-hoc*, behaviours.

This chapter is a culmination of the presented dissertation. It builds on my earlier papers with Jamroga and Köster (Novák and Köster, 2008; Novák and Jamroga, 2009). In fact, the chapter puts together the theoretical efforts of the previous parts of the dissertation. I finally demonstrated the *alternative approach to designing a high level agent-oriented programming language*. Instead of choosing a fixed set of agent-oriented features to be implemented in a language interpreter, I propose a purely syntactic approach to constructing high level language. It goes hand in hand with the semantic specification of the introduced constructs, patterns, in terms of program annotations.

Part III

Evaluation, extensions and beyond

... which finally demonstrates that the language can be practically applied in a variety of domains and thus provides a proof-of-concept for the proposed framework.

Chapter 7

Case studies

In the second part of this dissertation, I tried to approach the problem of pragmatics of the framework of *Behavioural State Machines*. In particular, I proposed the approach of *commitment oriented programming*, an informal methodological approach for development of cognitive agent systems with mental attitudes, supported by domain-independent code patterns. While the question of methodology is of the highest importance for engineering real-world systems, in order to provide substrate for the theoretical claims, a practical evaluation of the approach is needed.

To provide a proof-of-concept for the *BSM* framework, as well as to drive and nurture the research towards the methodology of development with *Jazzyk*, I designed and managed the development of three case-studies *Jazzbot*, *Urbibot* and *AgentContest team*. In the course of their development, we collected a body of experience with programming in *Jazzyk*. Especially the first proof-of-concept project, *Jazzbot*, served as the driver behind the methodological results presented in Chapter 6.

In this chapter, I briefly describe the three implemented demonstration applications. *Jazzbot*, the first case study, is a virtual bot in a simulated 3D environment of an open-source first person shooter computer game *Nexuiz* (Nexuiz Team, 2007). *Urbibot*, on the other hand, was developed as a step towards programming mobile robots. It is an agent program steering a model of a simulated small mobile robot in a 3D environment provided by a physical robotic simulator *Webots* (Cyberbotics Inc., 2009). The final application, *AgentContest team*, is a step towards experiments with multi-agent systems based on the *BSM* framework. It is our planned, non-competing, entry to the *Multi-Agent Programming Contest 2009* (Behrens et al., 2009b).

7.1 Jazzbot

Jazzbot is a virtual agent embodied in a simulated 3D environment of the first-person shooter computer game *Nexuiz* (Nexuiz Team, 2007). Its task is to explore a virtual building, search for certain objects in it and subsequently deliver them to the base. At the same time, *Jazzbot* is supposed to differentiate between other players present in the building and seek safety upon being attacked by an enemy player. When the danger disappears, it should return back to the activities interrupted by the attack.

Figure 7.1: Screenshot of the *Jazzbot* virtual agents in the virtual building of the *Nexuiz* game environment.

Jazzbot is a goal-driven BDI inspired cognitive agent developed with the *Jazzyk* language. The *Nexuiz* death-match game takes place in a virtual building containing various objects, such as e.g., weapons, flags or armor kits. The game engine is also capable of simulating diverse terrains like solid floor or liquid and provides basic means for inter-player interaction. Because of its accessibility, *Nexuiz* is published under the open source GNU GPL licence, we chose the *Nexuiz* game server as the simulator for *Jazzbot* case-study. *Jazzbot* was our first larger proof-of-concept application for the *BSM* framework and *Jazzyk*. Figure 7.1 depicts a screenshot of the *Jazzbot* agent acting in the simulated environment. Demonstration videos and source code can be found on the project website (Novák, 2009a).

Jazzbot's behaviour is implemented as a *Jazzyk* program. In the experimental scenario, the bot searches for a particular item in the environment, which it then picks up and delivers to the base point. For simplicity, the item to search for is hardcoded into *Jazzbot*'s belief base in design time. During the search phase the agent tries to always move to unexplored segments of the environment. On the other hand, when it tries to deliver the found item, it exploits a path planning algorithm to compute the shortest path to the base point. Hence, during the search phase, in every step the bot semi-randomly selects a direction to move to a previously unexplored part of the building and in the case there is none such, it returns to the nearest waypoint from which an unexplored direction exists. The behaviour for environment exploration is interrupted, whenever the bot feels under attack, i.e., an enemy player attempts to shoot at it. Upon that, it triggers emergency behaviours, such as running away from the danger. After the sense of emergency fades away, it returns back to its previously performed goals of item search or delivery.

Jazzbot's agent program was developed according to the naïve methodology, discussed previously in Chapter 6. It served as a vehicle for development of the methodology

Listing 7.1 Implementation of *Jazzbot*'s control cycle.

```
/* The actual Jazzbot agent program */
PERCEIVE , HANDLE_GOALS , ACT
```

Listing 7.2 Code snippet from the *Jazzbot* agent program.

```
define('ACT','{
        /* The bot searches for an item, only when it does not have it */
        when ⊨_G [{ task(search(X)) }] and not ⊨_B [{ hold(X) }] then SEARCH('X');

        /* When a searched item is found, it picks it */
        when ⊨_G [{ task(pick(X)) }] and ⊨_B [{ see(X) }] then PICK('X') ;

        /* When the bot finally holds the item, it deliver it */
        when ⊨_G [{ task(deliver(X)) }] and ⊨_B [{ hold(X) }] then DELIVER('X') ;

        /* Simple behaviour triggers without guard conditions */
        when ⊨_G [{ task(wander) }] then WALK ;
        when ⊨_G [{ task(safety) }] then RUN_AWAY ;
        when ⊨_G [{ task(communicate) }] then SOCIALIZE
}')
```

and the associated code patterns Therefore, only preliminary versions of the actual code patterns were used in it. The main control cycle, listed in Listing 7.1, consists of three steps that are executed sequentially. First, the bot reads its sensors (perception). Then, if necessary, deliberates about its goals and finally, it selects a behaviour according to its actual goals and beliefs (act). Listing 7.2 provides an example code implementing selection of goal oriented behaviours, realized as parametrized macros, triggered by *Jazzbot*'s goals. While the bot's goals simply trigger behaviours for walking around, danger aversion and social behaviour, executions of behaviours finally leading to getting an item are additionally guarded by belief conditions.

Figure 7.2 provides an overview of *Jazzbot*'s architecture. The agent features a belief base, consisting of two KR modules for representation of agent's actual beliefs and storing the map of the environment. The goal base encodes interrelationships between various agent's goals. Finally, bot interacts with the simulated environment by a module connecting it to a remote game simulation server.

JzNexuiz KR module (cf. Section 8.3), the *Jazzbot*'s interface to the environment, the body, provides the bot with capabilities for sensing and acting in the virtual world served by a *Nexuiz* game server. The bot can move forward, backward, it can turn or shoot. Additionally, *Jazzbot* is equipped with several sensors: GPS, sonar, 3D compass and an object recognition sensor. The module communicates over the network with the *Nexuiz* game server and thus provides an interface of a pure client side *Nexuiz* bot. I.e., the bot can access only a subset of the perceptual information a human player would have available.

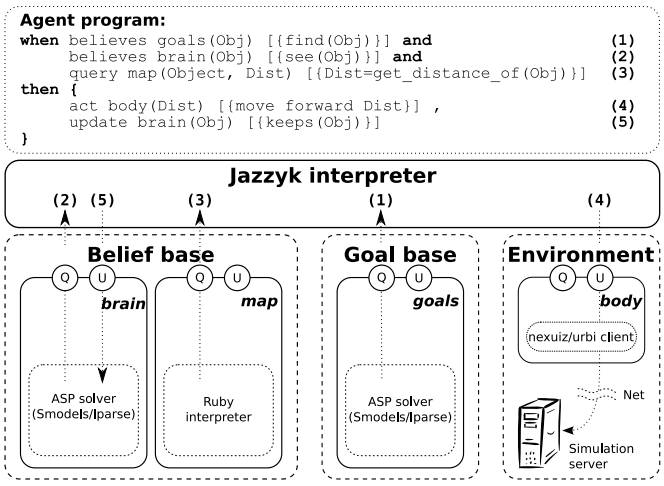


Figure 7.2: Shared internal architecture of *Jazzbot* and *Urbibot* agents.

Jazzbot's *belief base* is composed of two modules, *JzASP* (cf. Section 8.1) and *JzRuby* (cf. Section 8.2). The first integrates an *Answer Set Programming* solver *Smodels* by Syrjänen and Niemelä (2001) and contains an *AnsProlog** (Baral, 2003) logic program reflecting agent's beliefs about itself, the environment, objects in it and other players. The second, based on an interpreted object oriented programming language *Ruby* (Matsumoto, 2009), stores the map of the agent's environment. Listing 7.3 lists a code chunk implementing a part of the *AnsProlog** component of the *Jazzbot*'s belief base.

To represent *Jazzbot*'s information about the topology of its environment, the agent uses a *circle-based waypoint graph* (*CWG*) (Rabin, 2004) to generate the map of its environment. *CWG*'s are an improved version of *waypoint graphs*, extended with a radius for each waypoint. The radius is determined by the distance between the avatar and the nearest obstacle. This technique ensures, especially in large rooms and open spaces, a smaller number of waypoints within the graph, what in turn speeds up the path search algorithm. Figure 7.3 shows a graphical representation of the *CWG* for a sample walk of the *Jazzbot* agent from the spawn point to the point marked by the arrow.

Additionally, each waypoint stores a list of objects present within its range, as well as about walls touching it and information about unexplored directions, i.e., such in which there is no connection to another waypoint or a wall. By employing a breadth-first graph search algorithm, the agent can compute the shortest path to a particular object or a position.

The *CWG* graph is constructed by the agent so that in each step it determines whether its current absolute position corresponds to some known waypoint. If that is not the case, it turns around in 60° steps and by checking its distance sensor, it determines the nearest obstacle. Subsequently, the newly added waypoint is incorporated into the *CWG*

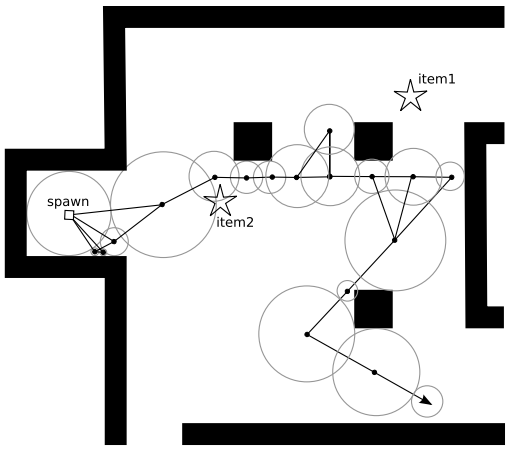


Figure 7.3: *CWG* map representation of the environment used by *Jazzbot*. The graph is a result of a sample walk through the room.

by connecting it to all the other waypoints with which it overlaps. All the perceived objects, together with all the directions in which the agent can see a wall are stored within the node.

Jazzbot's *goal base* is again an *AnsProlog** logic program representing its current goals and their interdependencies. Goals can be either of a declarative, *goals-to-be* or performative nature, *goals-to-do*, or *tasks*. In the *Jazzbot* implementation, each *goal-to-do* activates one or more *tasks*, which in turn trigger, one or more corresponding behaviours of the agent. The relationship between the two is encoded as an *AnsProlog** logic program, as listed in the excerpt from the *Jazzbot*'s belief base implementation in Listing 7.4.

On the ground of holding certain beliefs, the agent is also allowed to adopt new or drop old goals, which are either satisfied, irrelevant, or subjectively recognized as impossible to achieve. The agent thus implements *goal commitment strategies*. While Chapter 6 summarized the overall design methodology resulting from *Jazzbot* development, we discuss concrete details about the *Jazzbot* design in the joint paper with Köster (2008). Furthermore, Chapter 8 provides details on the implementation of the *Jazzbot*'s KR modules.

7.2 Urbibot

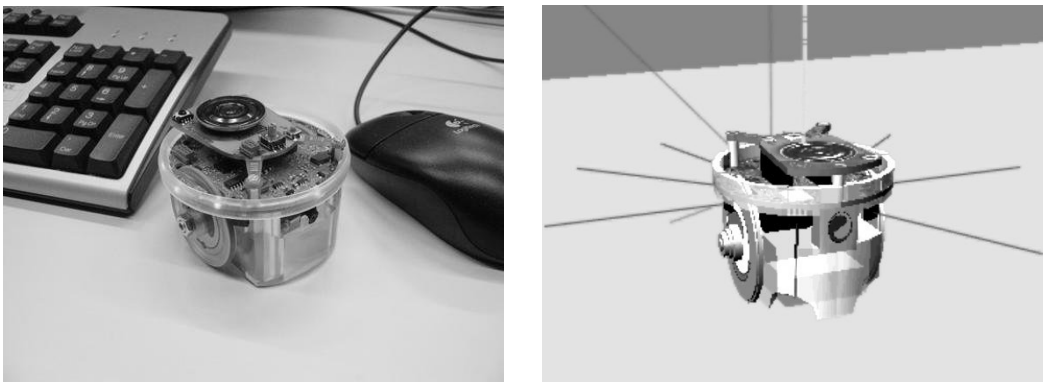
Urbibot is the second case-study, developed as a step towards applications of the *BSM* framework in the mobile robotics domain. It is an agent program steering a small, two-wheeled mobile robot in an environment provided by the physical robotic simulator. *Urbibot* explores a maze, where it searches for red poles and then tries to bump into each of them. At the same time it tries to avoid patrols policing the environment. Upon

Listing 7.3 *Jazzbot*'s belief base implementation in *AnsProlog**.

```
% Initially the bot does not hold the box %
% The bot can later hold other objects as well %
¬hold(box(42)).

% Reasoning about the health status %
alive :− health(X), X > 0.
dead :− health(X), X <= 0.
attacked :− health(X), X <= 90.
wounded :− health(X), X <= 50.

% Reasoning about friendliness of other players %
friend(Id) :− see(player(Id)), not attacked, player(Id).
enemy(Id) :− see(player(Id)), not friend(Id), player(Id).

player(1..5).
```

Listing 7.4 *Jazzbot*'s goal base implementation in *AnsProlog**.

```
% Initially the bot has two maintenance goals and %
% a single achievement goal. %
maintain(happy).
maintain(survive).
achieve(get(box(42))).

% Subgoals of the goal maintain(happy) %
task(communicate) :− maintain(happy).
task(wander) :− maintain(happy).

% Subgoals of the goal maintain(survive) %
task(wander) :− maintain(survive).
task(safety) :− maintain(survive).
task(energy) :− maintain(survive).

% Subgoals of the goal achieve(get(Object)) %
task(search(X)) :− achieve(get(X)), not achieve(get(medikit)), item(X).
task(pick(X)) :− achieve(get(X)), not achieve(get(medikit)), item(X).
task(deliver(X)) :− achieve(get(X)), not achieve(get(medikit)), item(X).

% Specialized subgoals of the goal achieve(get(medikit)) %
task(search(medikit)) :− achieve(get(medikit)).
task(pick(medikit)) :− achieve(get(medikit)).

% Ressurect after being killed %
task(reborn) :− achieve(reborn).

% Definition of items %
item(medikit).
item(X) :− box(X).

box(1..50).
```



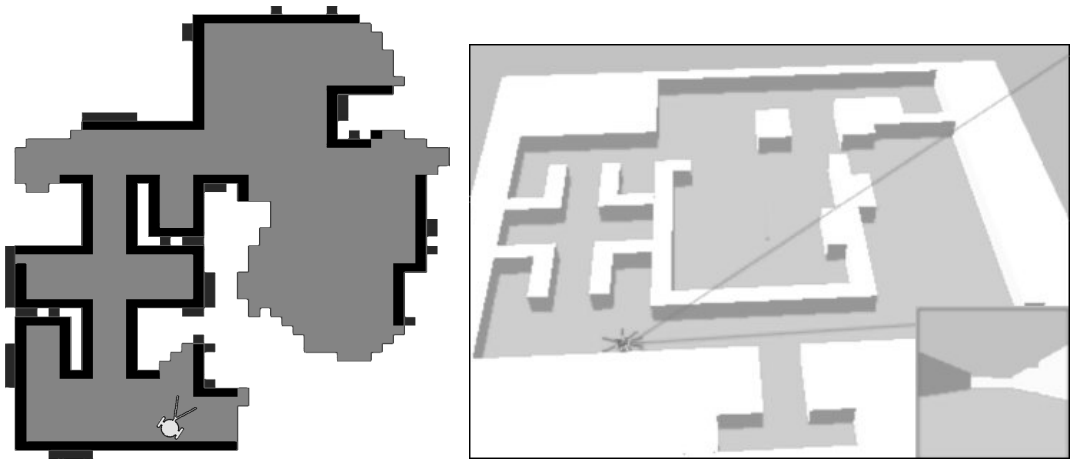
(a) (b)

Figure 7.4: Left, photograph of *e-puck* in life-size (Mondada et al., 2009) (re-printed with courtesy of authors). Right, a screenshot of the *Urbibot* in *Webots* simulator.

encounter with such a patrol, the robot runs away to finally return to the previously interrupted activity when safe again. *Urbibot* is embodied as an *e-Puck* (Mondada et al., 2009; EPFL, 2006), a small educational two-wheeled mobile robot simulated in *Webots*, a 3D physics simulator developed by Cyberbotics Inc. (2009) (also cf. (Michel, 1998)). The robot is steered using *URBI*, a highly flexible and modular robotic programming platform based on event-based programming model. *URBI* platform is developed by Gostai (2009a). The main motivation for using *URBI* is the direct portability of the developed agent program from simulator to the real robot.

Similarly to *Jazzbot*, the overall agent design is derived from the modular BDI architecture introduced in Chapter 3. It also some parts of the code developed for *Jazzbot*. In turn, except for using *JzASP* KR module to represent agent's beliefs about itself, *Urbibot* features similar agent architecture as the one depicted in the Figure 7.2 for the *Jazzbot* agent. *Urbibot*'s beliefs comprise exclusively information about the map. The interface to the simulator environment is provided by the *JzUrbi* KR module (cf. Section 8.4).

As already noted above, *Urbibot*'s behaviour is similar to that of *Jazzbot* agent. However instead of controlling the agent's body with rather discrete commands, such as move forward or turn left, *URBI* allows a more sophisticated control by directly accessing the robot's actuators through the *URBI* programming language primitives (Gostai, 2009c,b). In the particular case of the *e-Puck* robot, these are only its two wheels. The robot also features a mounted camera, a directional distance sensor and an additional GPS sensor, our sole customization of the original *e-Puck* robot model. In the *JzRuby* module, the robot analyzes the camera image stream and by joining it with the output of the distance and GPS sensors it constructs a 2D map of the environment. Figure 7.5a depicts the grid representation of the simulated environment as the *Urbibot* stores it in its *JzRuby*



(a) (b)

Figure 7.5: Left, representation of the environment grid map as the *Urbibot* represents it. On the right, a screenshot of the *Urbibot* exploring the maze. The lower right corner provides the current snapshot of the camera perception.

belief base. Black cells represent walls the robot recognizes, grey area is free space in which it can move and finally the white cells represent areas about which the robot has no information. The figure also depicts the robot's camera view and orientation angle.

Upon encountering a patrol robot, the bot calculates an approximation of the space the patrol robot can see, and subsequently tries to navigate out of this area. To plan paths between two points, *Urbibot* uses A^* algorithm. Again, the details on the implementation of the *Urbibot*'s KR modules can be found later in Chapter 8. Figures 7.4b and 7.5b depict screenshots of the *Urbibot* agent acting in the maze environment. Furthermore, demonstration videos and source code are provided in the corresponding section of the *Jazzyk* project website (Novák, 2009a).

7.3 AgentContest team

The final application developed in the context of my research is the *AgentContest team*, our non-competing entry to the *Multi-Agent Programming Contest 2009*.

Multi-Agent Programming Contest (Behrens et al., 2009b) is a tournament of multi-agent systems in a simple simulated environment. It is *an attempt to stimulate research in the area of multi-agent system* (MAS) development and programming, with the aim to *identify key problems of programming MASs* (Behrens et al., 2009b). Furthermore, the ambition of the project is to *develop benchmark problem instances serving as milestones for testing multi-agent programming languages, platforms and tools.* We published the

reports on the past editions of the competition in a sequel of papers (Dastani et al., 2005a, 2006, 2008a,b).

In the 2009 edition, participants should implement a team of agents playing cowboys herding cows. The agents navigate in a grid game environment in which there are herds of cows, which should be "pushed" into corrals. The team scores a point for each cow which is forced into the team's corral during the game. Apart from cows and cowboy agents, the environment contains also trees, obstacles, and fences, which can be opened by pushing an associated button. The game is designed in such a way that the team of agents is forced to cooperate in order to score points. The cooperation is enforced by the behaviour of cows, which tend to group in compact herds and are repelled by a cowboy standing in a vicinity. I.e., they perform behaviours, also know as flocking and dispersion. Thus, to move a group of cows in a certain direction, a team effort is required. Similarly, in order to open a fence gate, the team has to decide which member will push the button and the rest has to coordinate in order to push the herd through the gate. Figure 7.6 depicts an example of the game visualization.

Development of *AgentContest team* is a step towards evaluating the more advanced techniques for programming with the *BSM* framework, such as the goal oriented programming code patterns presented in Chapter 6, in a multi-agent setting. I.e., unlike in *Jazzbot* and *Urbibot* studies, the stress in this project is on communication and coordination among agents, rather than on their reactivity and pro-activity.

The single agent architecture for individual *AgentContest team* members remains the same as in the previous two case-studies. I.e., the agents feature a belief base written mostly in *Ruby* and a goal base implemented using technique similar to that used for the *Jazzbot* agent (cf. Listing 7.4). However, unlike the previous two projects, the interface to the environment comprises a union of two modules, *JzMASSim* and *JzOAAComm*, described in Section 8.5 and Section 8.6 respectively. *JzMASSim* connector plug-in provides an interface to the game environment simulated by the tournament server *MASSim* (Behrens et al., 2008, 2009a). *JzOAAComm* KR module provides communication facilities for the multi-agent system by exploiting the communication middleware platform *SRI Open Agent Architecture* (Cheyer and Martin, 2001; SRI International, 2007). Figure 7.7 shows a screenshot of the OAA agent monitor, with a team of communicating agents.

At the time of writing this thesis, the MAS implementation is still in development, therefore I provide here only a rough overview of the team's functionality. Similarly to the *Urbibot* agent, the *AgentContest team* members use A^* path planning algorithm within the belief base KR module providing representation of the environment topology. Since the environment provides only incomplete information about the grid, the agents share the information about their perceptions of their surroundings by broadcasting it upon local modification. On the command of a leader agent, the team coordinates in order to organize itself into group formations for pushing cow herds towards the desired direction. When needed, the team reorganizes the formation in order to change the

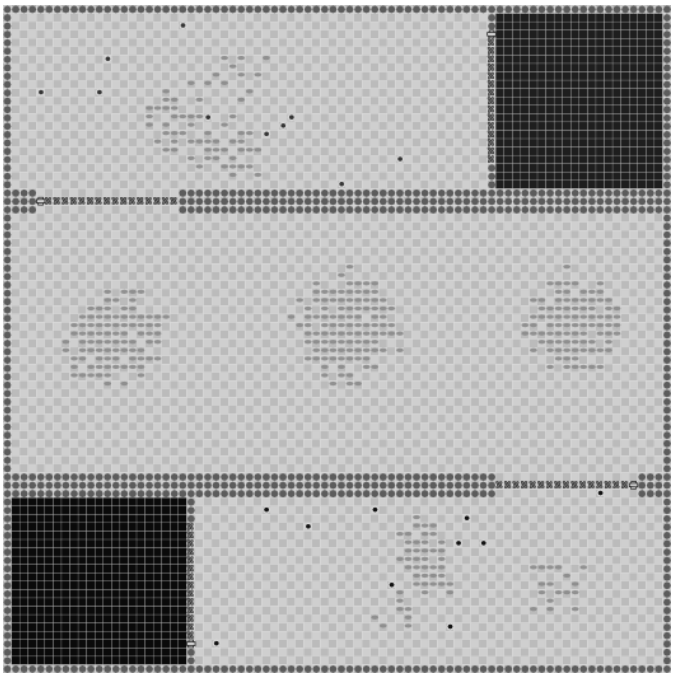


Figure 7.6: Example of the of the AgentContest 2009 scenario simulation environment grid. The squares depict corrals of teams, the gray ovals represent cows and black circles are two teams of cowboys (agents). Finally, the crosses between the blocks of tress (grey circles) represent fences which can be opened by standing on the button cell on sides of the fences.

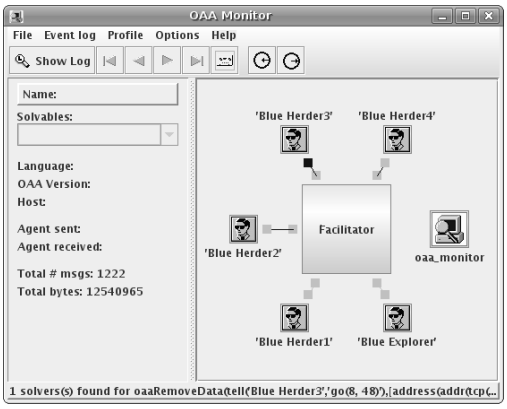


Figure 7.7: Screenshot of the OAA agent monitor depicting five connected agents communicating via the OAA Facilitator agent.

direction in which cows are being pushed, or it sends a team member to open a fence standing in the way. In order not to act in a counter-productive manner, agents avoid the cow herd while moving around to arrange themselves into a formation, or while going to open a fence. To maintain the scalability of the approach, the team members invoke the A^* path planning algorithm only when necessary, i.e., when a need to move from one point to another arises. Otherwise, they move between the extracted path plan nodes, crossroads where the agent should change its general walk direction, in a greedy manner.

7.4 Summary

Above, I briefly presented the three demonstration applications, which we developed to put the programming framework of *Behavioural State Machines* to a test, as well as a vehicle for driving and motivating the theoretical side of the research on methodology and pragmatics of *Jazzyk*. The chapter is based on the joint paper (Köster et al., 2009) and parts of (Novák and Köster, 2008), together with information from master's theses I co-supervised (Fuhrmann, forthcoming, 2009; Dereń, forthcoming, 2009).

Jazzbot virtual agent served as a proof-of-concept and demonstration of feasibility of doing advanced programming with mental attitudes in *Jazzyk*. One of the results of the project was the proposal of the naïve methodology for the *BSM* framework and first attempts to design reusable code patterns.

Urbibot allowed us to further experiment with the proposed code patterns and is meant as a platform on top of which, we can further experiment with cognitive robotics. One of the important requirements of the project was portability of the used technology and of the resulting agent program to real-world physical robots.

Finally the development of the *AgentContest team* should serve as the first attempt towards further experimentation with multi-agent and multi-robotic systems with the emphasis on individual cognitive agents. One of the results of the project is the proposal for a next-generation communication middleware for open heterogeneous multi-agent systems discussed later in Chapter 14.

One of the important by-products of the attempts to develop agent systems described above is the proof-of-feasibility for applications of *non-monotonic reasoning* (*NMR*) in BDI style cognitive agents. We showed that *NMR* can be used efficiently, without being the bottleneck in a very dynamic agent application. We achieved this by limiting the usage of logic programming techniques, *Answer Set Programming* (*ASP*), to tasks it is efficient in. Instead of using *ASP* for complete control of the robot, we use it solely for reasoning about static aspects of the world, such as relationships between objects and phenomena in the environment, agent's beliefs about itself etc. The reasoning about topology of the environment, such as object positions, distances between objects, or sizes, etc., is left to an object oriented language which provides better facilities for encoding algorithms over such information.

Chapter 8

Implemented Jazzyk modules

The core concept of the framework of *Behavioural State Machines* and the programming language *Jazzyk* is that of a *Knowledge Representation module*. While the language itself allows writing agent programs in terms of specification of interactions between modules, the actual reasoning, complex computations and interaction with an environment happens within the KR modules of an agent.

The notion of a KR module is an abstraction, encapsulating arbitrary knowledge bases of the agent. I adopt here a rather liberal view on what kind of technology can be considered a knowledge representation technology, rather similar to that of Davis et al. (1993). I.e., any technology, which allows a system designer and in turn the agent as well, to capture, express and efficiently manipulate the information required w.r.t. the application domain in focus.

In general, from the point of view of the *Jazzyk* language interpreter, a KR module is treated as a blackbox, providing only a generic interface (cf. Definition 2.1). The representation language, as well as the concrete semantics behind the knowledge manipulation operators, is hidden within the module itself. Due to this simplicity and generality, the notion of a KR module not only allows encapsulation of knowledge representation approaches in the traditional sense, but also any other information storage with the interface for *query* and *update*. I.e., not only those using some sort of symbolic logic based representation of objects and phenomena of the environment, but also other, rather non-traditional ones, such as e.g., relational and object databases, virtual machines, solvers of interpreted programming languages, or even artificial neural networks.

Various knowledge representation approaches come in a form of a programming language with an interpreter or a solver. On the other hand, agents interact with their environments by querying their sensors and controlling their effectors. The KR module generic *query*/*update* interface is able to accommodate both flavours of KR modules and thus make them accessible to *Jazzyk* agent programs. In line with behavioural roboticists, a KR module interfacing an agent program with an external environment can be seen as knowledge base representing itself, i.e., the environment is its own best representation.

This chapter discusses several KR modules, *Jazzyk* plug-ins, enabling development of the case studies described in Chapter 7. I describe six KR modules of the two flavours.

Firstly, *JzASP* (Section 8.1) and *JzRuby* (Section 8.2) provide interfaces to logic and object oriented programming engines respectively. Secondly, *JzNexuiz* (Section 8.3), *JzUrbi* (Section 8.4) and *JzMASSim* (Section 8.5) provide interfaces to various environments, agents of case studies in Chapter 7 act in. Finally, I discuss a KR module, which rather falls to the second category, but is useful in application independent settings. *JzOAAComm* (Section 8.6) interfaces agent programs with an inter-agent communication middleware, thus enabling communication and cooperation among agents.

8.1 Answer Set Programming KR module

Answer Set Programming (*ASP*) is a declarative logic programming framework. It stems from *stable model semantics* for logic programs, proposed by Gelfond and Lifschitz (1988). To enable logic based knowledge bases with non-monotonic reasoning for *Jazzyk* agents, we developed *JzASP* KR module. In the following, I briefly summarize *AnsProlog** syntax and semantics. Subsequently, I describe the *JzASP* KR module, facilitating *ASP* in *Jazzyk* agent programs, itself.

8.1.1 Answer Set Programming

According to supported features, such as default negation, integrity constraints, etc., *AnsProlog* comes in several flavours. Here, I consider the most general form coined *AnsProlog** (Baral, 2003), sometimes also referred to as *A-Prolog*.

An *AnsProlog** program P is a set of disjunctive *rules* of the form

$$a_1 \vee \cdots \vee a_n \ :- \ b_1, \ldots, b_k, \text{ not } b_{k+1}, \ldots, \text{ not } b_m. \tag{8.1}$$

$n \geq 0$, $m \geq k \geq 0$, $n + m \geq 0$ and $a_1, \ldots, a_n, b_1, \ldots, b_m$ are atoms. An *atom* is an expression of the form $p(t_1, \ldots, t_l)$, where p is a predicate symbol of arity $\alpha(p) = l \geq 0$ and each t_i is either a variable or a constant. A *literal* is an atom a or its negation not a. The *head* of a rule r, of the form 8.1, is the set $H(r) = \{a_1, \ldots, a_n\}$ and the *body* of r is the set $B(r) = \{b_1, \ldots, b_k, \text{not } b_{k+1}, \ldots, \text{not } b_m\}$. Furthermore, we denote $B^+(r) = \{b_1, \ldots, b_k\}$ and $B^-(r) = \{b_{k+1}, \ldots, b_m\}$. We also denote r as $H(r) : -B(r)$.

Let U_P be the *Herbrand Universe* of P, i.e., the set of all constants appearing in P. B_P denotes the *Herbrand Base* of P, i.e., the set of all literals constructable from predicate symbols in P and elements of U_P. We say that a program P is *ground*, iff it contains no variables in literals. A ground program, denoted *Ground*(P), can be obtained from the original *AnsProlog** program P by applying to each rule of P the set of all possible substitutions from the variables of P to elements of the *Herbrand Universe* U_P.

We say that a ground rule r is *satisfied* by the interpretation I, a set of ground atoms, iff $H(r) \cap I \neq \emptyset$ whenever $B^+(r) \subseteq I$ and $B^-(r) \cap I = \emptyset$. *I satisfies* the program P, iff each rule $r \in P$ is satisfied by I. For a non-ground *AnsProlog** program P, we say that I satisfies P, iff I satisfies each rule $r \in$ *Ground*(P).

Listing 8.1 Example of *JzASP* usage in *Jazzyk*. Adapted from the *Jazzbot* case-study.

```
declare module brain as asp

notify brain on initialize [{
        % Integrity constraints %
        :- peaceful, attacked.
        :- friend(Person), enemy(Person), person(Person).

        % Basic inference rules for bot's situation %
        peaceful :- health(X), X>50, not attacked.
        attacked :- health(X), X<=50.

        % Reasoning about peer agents %
        friend(F) :- not enemy(F), person(F).
}]

when sure_believes brain(Person) [{friend(Person)}] and poss_believes brain [{attacked}]
then add brain(Person) [{enemy(Person).}]
```

Finally, we say that an interpretation M is an *answer set* of P, iff it is the minimal interpretation (w.r.t. the set inclusion $\subseteq$) and satisfies the *Gelfond-Lifschitz reduct* of P (Gelfond and Lifschitz, 1988) defined as

$$P^M = \{H(r) : -B^+(r) \mid I \cap B^-(r) = \emptyset \wedge r \in Ground(P)\}. \tag{8.2}$$

In other words, an interpretation is an answer set of a program, iff it is the least model of the corresponding *Gelfond-Lifschitz* reduct. $AS(P)$ denotes the set of all answer sets of P.

We also say that a ground atom a is a *credulous* (also *possible* or *brave*) consequence of P, iff there exists an answer set (at least one) $A \in AS(P)$, s.t. $a \in A$. Complementary, we say that a is a *cautious* (also *certain* or *skeptical*) consequence of P, iff for each $A \in AS(P)$ we have $a \in A$.

8.1.2 JzASP

JzASP KR module (Köster, 2008) facilitates non-monotonic reasoning capabilities of *Answer Set Programming* to *Jazzyk* agent programs. It integrates an ASP solver *Smodels* by Syrjänen and Niemelä (2001), with accompanying logic program grounding tool *lparse* by Syrjänen (1998). The stored knowledge base of the module consists of a logic program in the *AnsProlog** language, introduced above, with some additional features provided by *lparse* and *Smodels*.

The knowledge base of the module can be accessed by two query operators surebelieves and possbelieves, corresponding to the credulous and cautious consequence relations respectively. The stored logic program can be updated by two complementary update methods add and del corresponding to a fact, rule, assertion and retraction. Internally, the *JzASP*

module processes the program by passing it to the *lparse* library and subsequently lets the *Smodels* solver to compute the program's answer sets. The actual query is answered by meta-reasoning over the program's answer sets.

Before a query formula is processed by the module, all the free variables occurring in it are substituted by their actual valuations in *Jazzyk* interpreter. Subsequently, the query method attempts to match the remaining free variables to terms from the computed answer sets. The variable substitution treatment for update formulae is similar to that in processing queries. Listing 8.1 shows an example of *Jazzyk* code using the *JzASP* KR module. Further examples can be found in previous chapters.

In the current incarnation, the *JzASP* KR module implements only a naïve LP update mechanism, based on plain assertion and retraction of facts or rules, with the same semantics as in *Prolog* (cf. e.g., (Shapiro and Sterling, 1994)). I.e., the formula is either added or deleted from the knowledge base, without implementing some repair mechanisms for the case, the asserted formula would cause inconsistency of the knowledge base. It is actually responsibility of the *Jazzyk* agent programmer to ensure that the updates during the agent's lifecycle won't violate the consistency of the knowledge base. In the future, a more sophisticated mechanism based on a kind of bounded *Dynamic Logic Programming* (Leite, 2003) could be implemented.

8.2 Ruby KR module

In the course of development of the case studies discussed in Chapter 7, *AnsProlog** turned out to be a very efficient tool for encoding parts of agent's knowledge bases. However, logic programming also turns out not to be a very practical tool for describing and reasoning about topologies of 3D environments or reasoning about distances between object, arithmetics, or even camera image classification. To facilitate such types of knowledge representation tasks, we implemented *JzRuby* KR module. It integrates an object-oriented scripting programming language *Ruby* and thus makes the interpreter available to *Jazzyk* agent programmers. The following two subsections briefly describe the language and the implemented module.

8.2.1 Ruby language

Ruby is a general purpose object-oriented programming language first invented by Matsumoto (2009) (for a more comprehensive overview, cf. e.g., the book by Flanagan and Matsumoto (2008)). Additionally, it supports also functional style of programming and its main features include dynamic typing, reflection and automatic memory management.

In the last years, *Ruby* has reached a certain level of maturity and broader acceptance. *Ruby* standard library includes support for the basic software development toolbox, such as arithmetics, collection of containers such as arrays, vectors, sets, etc., exception handling, string manipulation routines, full networking stack support, IO support, file

Listing 8.2 Example of *JzRuby* usage in *Jazzyk*. Informally extends the previous Example 8.1.

```
declare module map as ruby
declare module brain as asp
...

notify map on initialize [{
        # Initialize and load the code package
        require 'rubygems'
        load('jazzbotmap.rb')

        class JazzbotMap
                public
                def initialize
                        @graph = RGL::DirectedAdjacencyGraph.new
                        ...
                end
        ...
        end
        ...

        jbmap = JazzbotMap.new
}]

when poss_believes brain(Person) [{see(Person)}] then {
        when not query map(Person) [{jbmap.currCell.contains($Person)}]
        then update map(Person) [{jbmap.currCell.addObject($Person)}]
}
```

manipulation, (de-)compression routines, etc. Other notable packages developed by the *Ruby* community include complex libraries such as e.g., graph representation structures and manipulation routines, support for semantic web tasks, or XML handling.

Finally, the syntax of Ruby is flexible, simple and supports code modularization. It is also fairly familiar to programmers coming from other object-oriented languages, such as e.g., *C++* or *Java*.

8.2.2 JzRuby

Since *Ruby* is an interpreted scripting language, it is relatively simple to access the language interpreter in run-time and integrate it as a library into a 3rd party program. This feature enabled integration of *Ruby* interpreter into a *Jazzyk* KR module *JzRuby*.

The *JzRuby* plug-in (Fuhrmann, forthcoming, 2009) features a simple query/update interface, allowing evaluation of arbitrary *Ruby* expressions. It consists of a single query operator `query` and a single update operator `update`. The functionality of the *Ruby* KR module resembles the interactive mode of the *Ruby* interpreter, in which the user enters an arbitrary programming language expression on the command line and the interactive interpreter executes it and returns its value. Thus, the KR module is initialized with a

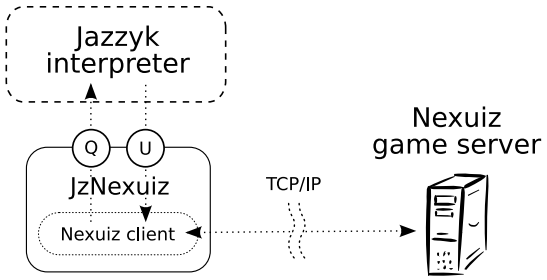


Figure 8.1: *JzNexuiz* implementation scheme.

regular *Ruby* program, possibly including external modules and libraries. Subsequently, each query/update invocation takes a formula, again a regular *Ruby* program, and passes it to the *Ruby* interpreter for evaluation. In the case of a query invocations the return value is captured and evaluated, 0 amounts to $\bot$ and any non-zero value yields $\top$. Update operator invocations are plainly executed in a synchronous manner, without evaluating the return value.

Before execution of a formula in the *Ruby* interpreter, query/update formulae variables are not directly substituted. Instead, they are declared as global name-space variables and substituted values are assigned to them. In turn, programmer refers in query/update formulae to global variables prepended by the dollar sign `$`. Listing 8.2 briefly shows an example of *Jazzyk* code using *JzRuby* KR module.

8.3 Nexuiz KR module

To enable development of the *Jazzbot* project, the first environment interface KR module developed for *Jazzyk* programming language was *JzNexuiz*. The KR module allows to connect to the *Nexuiz* game engine, which provides interface for interaction with a simulated 3D environment.

8.3.1 Nexuiz

Nexuiz is a free open-source first-person shooter game developed by *AlienTrap Software* group (AlienTrap Community, 2009) and published at the Nexuiz game website (Nexuiz Team, 2007). Nexuiz is based on a modified *DarkPlaces Quake* game engine by Hale (2009). A player in the game moves around a 3D space, usually a large multi-store building. According to the game mode, the task is either to kill other enemy players by shooting at them, defend a basis, or to capture the flag of the opponent team. The game provides a fully rendered graphical view to the simulated world including various visual and aural effects, such as explosions, etc.

Listing 8.3 Example of a code using *JzNexuiz* KR module.

```
declare module body as nexuiz

notify body on initialize [{include('config.cfg')}]

when sense body [{sonar}] then {
        act body [{turn left 90}] ;
        act body [{turn right 90}] }
else {
        act body [{move forward 1}]
}
```

The game software consists of two main components. The game simulation server and the client. While the server runs the game simulation and provides multi-player platform for interaction and communication, the client merely receives perceptions of the virtual avatar in the game and sends player's atomic commands, key strokes, to the server. The server subsequently updates the state of the simulated environment accordingly.

8.3.2 JzNexuiz

JzNexuiz module (Mainzer, 2008) integrates the *Nexuiz* game client and uses it to connect to the game server, as depicted in Figure 8.1. Technically, the module connects over TCP/IP with a *Nexuiz* server. The plug-in integrates a stripped down and customized *Nexuiz* client source code. In turn, the bot's actions are implemented as the corresponding key strokes performed by a human player.

The intended consequence of the technical setup described above, is that an agent using *JzNexuiz* module is a pure client-side bot. That means that in order to faithfully mimic the human player style environment for the bot, the sensory interface is designed so that it provides only a (strict) subset of the information of that a human game player can access. For instance, a *Jazzyk* bot with *JzNexuiz* module can only check the scene in front of it using the directional sonar sensor. The rendering of the whole scene is inaccessible to it, so only a single object can be seen at a time. Similarly to a human player, the bot can reach only to the local information about its environment. Information about objects which it cannot see, or are located behind the walls of the space it stands in, are inaccessible to it.

The *JzNexuiz* KR module implements the game client functionality and facilitates the bot's interaction with the game server. The list of sensors agents can exploit includes the body health sensor, liquid sensor, ground sensor, gps device, 3D compass device, sonar, map ID information, eye sensor and ear sensor listen. Similarly, the list of actuators contains basic capabilities move, turn, jump, use, attack and say. Each query formula starts with the name of the accessed virtual sensor device, followed by the corresponding arguments being either constants or variables facilitating retrieval of information from the environment.

Similarly, the update formulae consist of the action to be executed by the avatar followed by a list of arguments specifying the command parameters. The truth value of a query formula evaluation depends on the sensory input retrieved from the environment. In the case the query evaluates to true ($\top$), the additional information about e.g., the distance of an obstacle or the reading of the body health sensor is stored in the provided free variables. Listing 8.3 provides a sample program demonstrating the use of the *JzNexuiz* KR module.

8.4 URBI plug-in

Urbibot agent, described in Chapter 7, is an agent steering a small mobile robot. To enable development of *Urbibot*, as well as to provide a general purpose interface to robotic hardware, we implemented the *JzUrbi* plug-in. It utilizes the special purpose robot programming language *URBI*.

8.4.1 Urbi

URBI, Universal *Real-Time Behavior Interface* developed by Gostai (2009a), is a software platform for development of applications for robotics. It is based on *URBI*, an event-driven programming language supporting parallelization. Rather than supporting general purpose programming features, *URBI*'s aim is to enable efficient creation of programs from components called *UObjects*. Thus, *URBI* allows "gluing" of heterogeneous components, representing parts of the robot hardware body, into a functional complex. Instead of synchronization constructs, *URBI* supports mixing modes for variables shared among threads of executions. I.e., in the case a variable represents the speed of a servo engine and two threads set it to two different values at the same time, the actual resulting speed is calculated as a mix of the two, e.g., an arithmetic average.

URBI platform is a portable cross-platform tool, featuring the familiar *C++* style syntax. It is divided into two components, the *URBI* server and the client. While the server interprets commands directly on the *UObject* components, representing hardware parts of the robot, the client is merely a thin telnet client which allows remote execution of the *URBI* program.

8.4.2 JzUrbi

In order to interface *Jazzyk* programs with a robotic body, the *JzUrbi* KR module (Fuhrmann, forthcoming, 2009) integrates the *URBI* programming language client side interpreter. Similarly to the *JzNexuiz* module, it connects over TCP/IP to an *URBI* server on the simulator's side or with the *URBI* robot controller that controls the robot's body.

The single query method query provides the agent program with the sensor information from the body. *Jazzyk* variables are treated the same way as in *JzRuby*, i.e., as global variables of the underlying *URBI* interpreter. Similarly, the single update method update simply sends the provided update formula, an *URBI* program, to the *URBI* server.

To steer the robot's hardware components, an agent connects over the *JzUrbi* module with the *URBI* server, running either in the *Webots* simulator by Cyberbotics Inc. (2009) or on the actual robot controller (on-board or off-board). In the particular case of the *Urbibot*, the sensory input accessed the following sensors: camera, distance sensor, GPS, touch-sensor and a light-sensor. Together with the update interface steering the *Urbibot*'s two wheels, these two methods provide the basic interface for the robot control.

8.5 MASSim client plug-in

The last implemented *Jazzyk* KR module interfacing agent programs with the environment for the third case-study *AgentContest 2009 team* presented in Chapter 7 is the *JzMASSim* module.

8.5.1 MASSim

In the last years, in the *Computational Intelligence Group* of *Clausthal University of Technology*, we implemented and ran the infrastructure for the *Multi-Agent Programming Contest* (Dastani et al., 2005a, 2006, 2008a,b; Behrens et al., 2009b). As already noted in Chapter 7, *AgentContest* is a tournament of multi-agent systems in a simple simulated environment. The tournament is executed remotely. I.e., the game simulation runs on the tournament server called *MASSim*, while the individual agents connect to it via Internet by utilizing the standard TCP protocol. In each step of the game, the server sends to the connected agents information about their individual local perceptions, i.e., an XML message describing the contents of cells in the relevant small cut-out of the overall grid. Agents then have limited time to deliver an answer, again a XML message, describing their individual actions in the environment. Failing to act, amounts to no action. Furthermore, agents can test the connectivity to the game server by sending ping messages. In the case of a network connection disruption, agents can reconnect.

8.5.2 JzMASSim

The communication protocol between the *MASSim* server and the connected agents is based on XML standard (W3C, 2008a) and follows a request-response scheme. Thus, even when the actual contents of the exchanged XML messages change from edition to edition, the overall protocol stays the same. This allowed development of a general, on a concrete game scenario independent, connector to the *MASSim* server infrastructure (Behrens et al., 2008), *JzMASSim* (Yi, 2009).

Listing 8.4 Example of a *Jazzyk* program utilizing *JzMASSim* KR module.

```
declare module client as MASSClient
declare module brain as ruby

/* Initialization with the connection data: server IP/port and agent username/password */
notify mclient on initialize [{
        ip=127.0.0.1, port=12300,
        usr=botagent1, pwd=mypassword
}]

notify parser on initialize [{
        /* Ruby program for deciding the next action. */
}]

when isStarted client [{ }] then {
        /* either retrieve the next message from the mailbox */
        when getMsg mclient (Msg) [{ }] then {
                /* incorporate the new perceptions and delete the queue head */
                update brain(Msg) [{ map.addPerception($Msg) }] ,
                delMsg client [{ }]
        } ;
        /* ... or act */
        when query brain (Action) [{ $Action = me.nextAction }] then sendMsg client (Action) [{ }]
}
```

JzMASSim plug-in basically implements a mailbox for the XML messages exchanged with the *MASSim* server. The mailbox is asynchronous, i.e., the mailbox collects messages in a queue as they come from the *MASSim* server and upon a query from the *Jazzyk* agent program a message can be retrieved. The KR module provides two query operators getInboxSize and getMsg retrieving the current size of the mailbox queue and the message in the queue head. The retrieved message is filled into a provided free variable, therefore query formulae can be arbitrary, i.e., they do not bear any semantics. The query/update semantics is provided by the operators themselves.

To act in the simulated environment, agents are supposed to send the *MASSim* server a well-formed XML message describing the action, the agent intends to execute in the next simulation step. The update operator sendMsg provides this functionality. The message to be sent to the server is supposed to be contained in the argument variable again. Additionally, the module provides an update operator delMsg which deletes the message in the head of the mailbox queue.

Note that in order to keep the module general and reusable across several *AgentContest* editions, the *JzMASSim* module cannot deal with the semantics of the delivered or sent XML messages directly. Instead, the agent programmer must use another KR module, which provides XML parsing capabilities to decode, analyze and encode the XML messages. As a suitable option, the *JzRuby* KR module can be used. Listing 8.4 shows an example of a small *Jazzyk* program utilizing *JzMASSim* KR module.

Listing 8.5 Example of a *Jazzyk* program utilizing *JzOAAComm* module.

```
declare module comm as oaacomm

notify comm on initialize [{
        oaa_params('192.168.0.3', '3378'),
        register('Agent007')
}]

notify comm on finalize [{oaa_disconnect}]

when query comm (Msg) [{get_message}]
        then update comm [{tell('Agend42', 'I've got a message!')}]
        else update comm [{broadcast('Send me a message please.')}}
```

8.6 OAA connector KR module

The above described *Jazzyk* KR modules primarily support development of single-agent systems. Communication and coordination between agents lie on the way towards multi-agent systems development. Multi-agent communication middleware platforms provide support for inter-agent communication. To support development of the *AgentContest team*, we exploit the *Open Agent Architecture* (*OAA*) by SRI International (2007) and described also in detail by Cheyer and Martin (2001).

8.6.1 Open Agent Architecture

OAA is "*a framework for integrating a community of heterogeneous software agents in a distributed environment*" (SRI International, 2007). The core component of the *OAA* platform is the *Facilitator*, an agent whose capabilities involve distributing and coordinating tasks within the multi-agent system. Its tasks include maintaining a yellow pages registry of connected agents and their capabilities, called *solvables*, as well as routing peer-2-peer messages, broadcasts and multi-casts betwen distributed agents. Furthermore, given a complex task, solving of which requires decomposition into a plan of single agent capability invocations, it can plan and facilitate the execution of the task. Subsequently it collects the resulting partial answers into a compound solution to the complex request. Agents connected to the *OAA* platform communicate by exchanging messages encoded in *OAA*'s proprietary *Interagent Communication Language* (*ICL*). *ICL*'s syntax resembles that of *Prolog* (Shapiro and Sterling, 1994) and thus is easily accessible to casual MAS programmers.

8.6.2 JzOAAComm

Module *JzOAAComm* (Dereń, forthcoming, 2009) interfaces *Jazzyk* agent programs with the *OAA Facilitator* agent. Similarly to the *JzURBI* and *JzMASSim* plug-ins, it wraps

a client side connector which communicates over TCP/IP network stack with the sever-side *OAA Facilitator* agent. The query/update interface consists of two operators query and update respectively. The module is initialized with the configuration for connecting to the *Facilitator*. *Jazzyk* agent is supposed to provide an IP address and port of the node where the *Facilitator* is listening. The agent can also register its unique identifier by means of the register/1 predicate and the list of registered capabilities, solvables, of the agent by means of the solvables/n predicate. Solvables are simple *ICL* terms.

The query and update formulae take the form inspired by *ICL* terms, i.e., requests and commands are encoded as predicate symbol with zero or more arguments. The available query requests include get_message/0, get_community/0, can_solve/1 and registered_solvables/1. These respectively extract a message from the mailbox queue, extract the list of agents registered with the *Facilitator* agent, extract the list of agents, which can solve the solvable provided as the argument and finally extract the list of solvables registered with the *Facilitator*. The update formulae can be built from the following commands broadcast/1, multicast/2 and tell/2. The first takes as an argument only the message to be broadcasted to the community, while the latter two also require specification of recipients. In the case of the multicast command, the argument is a solvable and the message will be sent to all the agents which registered it. Finally, the simple peer-2-peer message send requires specification of an identifier of the agent on the receiving side of the communication channel.

Unlike the *JzMASSim* module, *JzOAAComm* does not maintain a mailbox for the *Jazzyk* agent locally. Instead, the agents' mailboxes are managed centrally by the *Facilitator* agent and the *JzOAAComm* module simply queries it to extract the message at the head of the queue. Listing 8.5 lists a short program showing usage of *JzOAAComm* KR module for communication among agents in a small multi-agent systems.

8.7 Summary

The presented chapter briefly describes six *Jazzyk* plug-ins, KR modules, which were used in development of the case-studies described earlier in Chapter 7. The first two facilitate exploitation of two proper knowledge representation technologies, logic programming and object-oriented programming languages. The subsequent three provide interfaces to various environments agents can be embodied in. The last plug-in facilitates inter-agent communication in a multi-agent system consisting of heterogeneous collaborating agents.

The work presented here is a result of several diploma theses (Köster, 2008; Fuhrmann, forthcoming, 2009; Dereń, forthcoming, 2009; Mainzer, 2008) and a student project (Yi, 2009) written under my co-supervision in the *Computational Intelligence Group* at *Clausthal University of Technology*. The KR modules were implemented in *C++* programming language, on top of the *Jazzyk SDK*, *Jazzyk* software development kit for KR module plug-ins. Appendix B presents the *Jazzyk SDK* package in more detail.

Chapter 9

Probabilistic Behavioural State Machines

Situated cognitive agents, such as mobile service robots, operate in rich, unstructured, dynamically changing and not completely observable environments. Since various phenomena of real world environments are not completely specifiable, as well as because of limited, noisy, or even malfunctioning sensors and actuators, such agents must operate with *incomplete information*.

On the other hand, similarly to mainstream software engineering, *robustness* and *elaboration tolerance* are some of the desired properties for cognitive agent programs. Embodied agent is supposed to operate reasonably well, also in conditions previously unforeseen by the designer. It should degrade gracefully in the face of partial failures and unexpected circumstances (robustness). At the same time, the program should be concise, easily maintainable and extensible (elaboration tolerance).

Agent programs in the reactive planning paradigm are specifications of partial plans for an agent, about how to deal with various situations and events occurring in the environment. On one side, the inherent incomplete information stems from the level of knowledge representation granularity chosen at the agent's design phase, i.e., it is impossible to model the full detail of reality in an embodied agent. On the other, as engineers, we are striving for robust and easily maintainable programs, so these should be readable, well structured and at the same time expressive enough for the application in consideration. The tension between the two sets of requirements yields a trade-off of *intentional underspecification* of resulting agent programs.

Most BDI inspired agent-oriented programming languages on both sides of the spectrum, between theoretically founded to pragmatic ones (I provide an overview in Chapter 1), facilitate encoding of underspecified, non-deterministic programs. Any given situation or an event can at the same time trigger multiple behaviours, which themselves can be non-deterministic, i.e, can include alternative branches.

A precise and exclusive qualitative specification of behaviour triggering conditions is often impossible due to the, at the design time chosen and fixed, level of knowledge representation granularity. This renders the qualitative condition description a rather coarse grained means for steering agent's life-cycle. In such contexts, a quantitative heuristics steering the language interpreter's choices becomes a powerful tool for encoding developer's informal knowledge or intuitions about agent's run-time evolutions. For

example, it might be appropriate to execute some applicable behaviours more often than others, or some of them might intuitively perform better than other behaviours in the same context, and therefore should be preferably selected.

This chapter presents *Probabilistic Behavioural State Machines* (*P-BSM*), a *probabilistic extension* of the BSM framework. The core idea is straightforward. Language rules (mst's) coupling a triggering condition with an applicable behaviour are extended with labels denoting the probability, with which the interpreter's selection mechanism should choose the particular behaviour in a given context. One of the elegant implications of the extension is that subprograms with labelled rules can be seen as specifications of probability distributions over actions applicable in a given context. This allows steering agent's focus of deliberation on a certain subbehaviour with only minor changes to the original agent program. I call this technique *adjustable deliberation*.

In the following, I first introduce the probabilistic extension of the original *BSM* framework and subsequently, I also present *Jazzyk(P)*, the corresponding extension of the *Jazzyk* programming language.

9.1 Probabilistic Behavioural State Machines

In the plain *BSM* framework, the syntactic construct of a mental state transformer encodes a transition function over the space of mental states of a *BSM* (cf. Definition 2.10). Hence, an execution of a compound non-deterministic choice mst amounts to a non-deterministic selection of one of its components and its subsequent application to the current mental state of the agent. In order to enable finer grained control over this selection process, the core idea behind the *P-BSM* framework is the extension of the *BSM* formalism with specifications of probability distributions over components of choice mst's.

The *P-BSM* formalism, below, heavily builds on associativeness of *BSM* composition operators of non-deterministic choice and sequence (cf. Remark 2.13). We also informally say that an mst τ *occurs* in a mst τ' iff τ' can be constructed from the set of all mst's $\mathfrak{T}$, by using composition operators as defined by Definition 2.4.

Definition 9.1 (Probabilistic BSM). A *Probabilistic Behavioural State Machine* (*P-BSM*) $\mathcal{A}_p$ is a tuple $\mathcal{A}_p = (\mathcal{M}_1, \ldots, \mathcal{M}_n, \mathcal{P}, \Pi)$, where $\mathcal{A} = (\mathcal{M}_1, \ldots, \mathcal{M}_n, \mathcal{P})$ is a *BSM* and $\Pi : \tau \mapsto P_\tau$ is a function assigning to each non-deterministic choice mst of the form $\tau = \tau_1 | \cdots | \tau_k \in \mathcal{P}$ occurring in $\mathcal{P}$ a discrete probability distribution function $P_\tau{:}\tau_i \mapsto [0, 1]$, s.t. $\sum_{i=1}^{k} P_\tau(\tau_i) = 1$.

W.l.o.g. it is safe to assume that each mst occurring in the agent program $\mathcal{P}$ can be uniquely identified, e.g., by its position in the agent program.

The probability distribution function P_τ assigns to each component of a non-deterministic choice mst $\tau = \tau_1 | \tau_2 | \cdots | \tau_k$ a probability of its selection by a *BSM* interpreter.

Note that because of the unique identification of mst's in an agent program $\mathcal{P}$, the function Π assigns two distinct discrete probability distributions P_{τ_1} and P_{τ_2} to choice mst's τ_1, τ_2, even when they share the syntactic form but occur as distinct components of $\mathcal{P}$. To distinguish from the *BSM* formalism, we call mst's occurring in a *P-BSM probabilistic mental state transformers*. *BSM* mst's as defined in Definition 2.4 will be called *plain*.

Similarly to plain mst's, the semantic counterpart of a probabilistic mst is a probabilistic update. A *probabilistic update* of a *P-BSM* $\mathcal{A}_p = (\mathcal{M}_1, \ldots, \mathcal{M}_n, \mathcal{P}, \Pi)$ is a tuple $p{:}\rho$, where $p \in \mathbb{R}$, s.t. $p \in [0, 1]$, is a probability and $\rho = (\oslash, \psi)$ is an update from the *BSM* $\mathcal{A} = (\mathcal{M}_1, \ldots, \mathcal{M}_n, \mathcal{P})$. In turn, the semantics of a probabilistic mental state transformer in a state σ, formally defined by the $yields_p$ predicate below, is the probabilistic update set it yields in that mental state.

Definition 9.2 ($yields_p$ calculus). A probabilistic mental state transformer τ yields a probabilistic update $p{:}\rho$ in a state σ, iff $yields_p(\tau, \sigma, p{:}\rho)$ is derivable in the following calculus:

$$\frac{\top}{yields_p(\mathbf{skip},\sigma,1{:}\mathbf{skip})} \qquad \frac{\top}{yields_p(\oslash\psi,\sigma,1{:}(\oslash,\psi))} \qquad \text{(primitive)}$$

$$\frac{yields_p(\tau,\sigma,p{:}\rho), \sigma \models \phi}{yields_p(\phi \longrightarrow \tau,\sigma,p{:}\rho)} \qquad \frac{yields_p(\tau,\sigma,\theta,p{:}\rho), \sigma \not\models \phi}{yields_p(\phi \longrightarrow \tau_p,\sigma,1{:}\mathbf{skip})} \qquad \text{(conditional)}$$

$$\frac{\tau{=}\tau_1|\cdots|\tau_k, \Pi(\tau){=}P_\tau, \forall 1{\leq}i{\leq}k{:}\ yields_p(\tau_i,\sigma,p_i{:}\rho_i)}{\forall 1{\leq}i{\leq}k{:}\ yields_p(\tau,\sigma,P_\tau(\tau_i){\cdot}p_i{:}\rho_i)} \qquad \text{(choice)}$$

$$\frac{\tau{=}\tau_1{\circ}\cdots{\circ}\tau_k, \forall 1{\leq}i{\leq}k{:}\ yields_p(\tau_i,\sigma_i,p_i{:}\rho_i) \wedge \sigma_{i+1}{=}\sigma_i \bigoplus \rho_i}{yields(\tau,\sigma_1,\prod_{i=1}^{k} p_i{:}\rho_1 \bullet \cdots \bullet \rho_k)} \qquad \text{(sequence)}$$

The modification of the plain *BSM yields* calculus, introduced in Definition 2.9, for primitive and conditional mst's is rather straightforward. A plain primitive mst yields the associated primitive update for which there's no probability of execution specified. A conditional mst yields probabilistic updates of its right hand side if the left hand side query condition is satisfied. It amounts to a **skip** mst otherwise. The function Π associates a discrete probability distribution function with each non-deterministic choice mst and thus modifies the probability of application of the probabilistic updates yielded by its components accordingly. Finally, similarly to the plain *yields* calculus, a sequence of probabilistic mst's yields sequences of updates of its components, however the joint application probability equals to the conditional probability of selecting the particular sequence of updates. The following example illustrates the sequence rule of the probabilistic $yields_p$ calculus.

Example 9.3. Consider the mst $(0.3{:}\tau_1 \,|\, 0.7{:}\tau_2) \circ (0.6{:}\tau_3 \,|\, 0.4{:}\tau_4)$. Let's assume that for each of the component mst's τ_i, we have $yields_p(\tau_i, \sigma, p_i{:}\rho_i)$ in a state σ. The plain *yields*

calculus yields the following sequences of updates $\rho_1 \bullet \rho_3$, $\rho_1 \bullet \rho_4$, $\rho_2 \bullet \rho_3$ and $\rho_2 \bullet \rho_4$. The probability of selection of each of them, however, equals to the conditional probability of choosing an update from the second component of the sequence, provided that the choice from the first one was already made. I.e., the probabilistic $yields_p$ calculus results in the following sequences of probabilistic updates $0.18{:}(\rho_1 \bullet \rho_3)$, $0.12{:}(\rho_1 \bullet \rho_4)$, $0.42{:}(\rho_2 \bullet \rho_3)$ and $0.28{:}(\rho_2 \bullet \rho_4)$.

The corresponding adaptation of the mst denotational semantics from Definition 2.10 straightforwardly follows.

Definition 9.4 (probabilistic mst denotational semantics). Let $\mathcal{A}_p = (\mathcal{M}_1, \ldots, \mathcal{M}_n, \mathcal{P}, \Pi)$ be a *P-BSM*. A probabilistic mental state transformer τ encodes a transition function $\mathfrak{fp}_\tau : \sigma \mapsto \{p : \rho | yields_p(\tau, \sigma, p : \rho)\}$ over the space of mental states $\sigma \in \mathcal{S}$, where $\sigma = \langle \sigma_1, \ldots, \sigma_n \rangle$ and $\mathcal{S} = S_1 \times \cdots \times S_n$ is the space of the agent's mental states.

According to the Definition 9.1, each mst occurring in a *P-BSM* agent program can be uniquely identified. Consequently, also each probabilistic update yielded by the program can be uniquely identified by the mst it corresponds to. The consequence is that w.l.o.g. we can assume that even when two probabilistic updates $p_1{:}\rho_1$, $p_2{:}\rho_2$, yielded by the agent program $\mathcal{P}$ in a state σ, share their syntactic form, i.e. $p_1 = p_2$ and ρ_1, ρ_2 encode the same plain *BSM* update, they both independently occur in the probabilistic update set $\mathfrak{fp}(\sigma)$.

The following lemma shows that the semantics of probabilistic mst's embodied by the $yields_p$ calculus can be understood as *an encoding of a probability distribution* or *a probabilistic policy* over updates yielded by the underlying plain mst. Moreover, it also implies that composition of probabilistic mst's maintains their nature as probability distributions.

Lemma 9.5. *Let $\mathcal{A}_p = (\mathcal{M}_1, \ldots, \mathcal{M}_n, \mathcal{P}, \Pi)$ be a* P-BSM. *For every mental state transformer τ occurring in $\mathcal{P}$ and a mental state σ of $\mathcal{A}_p$, we have*

$$\sum_{p:\rho \in \mathfrak{fp}_\tau(\sigma)} p = 1 \tag{9.1}$$

Proof of Lemma 9.5. The proof follows by induction on nesting depth of mst's. The nesting depth of an mst is the maximal number of steps required to derive $yields_p(\tau, \sigma, p{:}\rho)$ in the $yields_p$ calculus for all σ from $\mathcal{A}_p$ and all $p{:}\rho$ yielded by τ.

depth $= 1$**:** Equation 9.1 is trivially satisfied for primitive updates from $\mathcal{A}_p$ of the form **skip** and $\oslash\psi$.

depth $= 2$**:** Let's assume $\tau_1, \ldots, \tau_k$ are primitive mst's yielding $1{:}\rho_1, \ldots, 1{:}\rho_k$ in a state σ respectively, and ϕ be a query formula. We recognize three cases

conditional: In the case $\sigma \models \phi$, we have $yields_p(\phi \longrightarrow \tau_1, \sigma, 1{:}\rho_1)$. Similarly for $\sigma \not\models \phi$, we have $yields_p(\phi \longrightarrow \tau_1, \sigma, 1{:}\mathbf{skip})$, hence Equation 9.1 is satisfied in both cases.

choice: According to Definition 9.2 for each $1 \leq i \leq k$ we have

$$yields_p(\tau_1 | \cdots | \tau_k, \sigma, P_{\tau_1|\cdots|\tau_k}(\tau_i){:}\rho_i)$$

where $\Pi(\tau_1 | \cdots | \tau_k) = P_{\tau_1|\cdots|\tau_k}$. Since $P_{\tau_1|\cdots|\tau_k}$ is a discrete probability distribution (cf. Definition 9.1) over the elements $\tau_1, \ldots, \tau_k$, we have

$$\sum_{1 \leq i \leq k} P_{\tau_1|\cdots|\tau_k}(\tau_i) = 1$$

hence Equation 9.1 is satisfied as well.

sequence: For the sequence mst, we have $yields_p(\tau_1 \circ \cdots \circ \tau_k, \sigma, 1{:}(\rho_1 \bullet \cdots \bullet \rho_k))$, so Equation 9.1 is trivially satisfied again.

depth $= n$**:** Assume that Equation 9.1 holds for mst's of nesting depth $n-1$, we show it holds also for mst's of depth n. Again, we assume that ϕ is a query formula of $\mathcal{A}_p$ and $\tau_1, \ldots, \tau_k$ are compound mst's of maximal nesting depth $n-1$ yielding sets of updates $\mathfrak{fp}_{\tau_1}(\sigma), \ldots, \mathfrak{fp}_{\tau_k}(\sigma)$ in a mental state σ respectively. Similarly to the previous step, we recognise three cases:

conditional: According to the derivability of ϕ w.r.t. σ, for the conditional mst $\phi \longrightarrow \tau_1$ we have either $\mathfrak{fp}_{\phi \longrightarrow \tau_1}(\sigma) = \mathfrak{fp}_{\tau_1}(\sigma)$ or $\mathfrak{fp}_{\phi \longrightarrow \tau_1}(\sigma) = \{1{:}\mathbf{skip}\}$. For the latter case, Equation 9.1 is trivially satisfied and since τ_1 is of maximal nesting depth $n-1$, we have $\sum_{p:\rho \in \mathfrak{fp}_{\phi \longrightarrow \tau_1}(\sigma)} p = \sum_{p:\rho \in \mathfrak{fp}_{\tau_1}(\sigma)} p = 1$ as well.

choice: Let $P_{\tau_1|\cdots|\tau_k}$ be the probability distribution function assigned to the choice mst $\tau_1 | \cdots | \tau_k$ by the function Π. We have

$$\mathfrak{fp}_{\tau_1|\cdots|\tau_k}(\sigma) = \left\{ p{:}\rho \middle| \exists 0 \leq i \leq k : yields_p(\tau_i, \sigma, p_i{:}\rho) \wedge p = P_{\tau_1|\cdots|\tau_k}(\tau_i) \cdot p_i \right\}$$

Subsequently,

$$\sum_{p:\rho \in \mathfrak{fp}_{\tau_1|\cdots|\tau_k}(\sigma)} p = \sum_{0 \leq i \leq k} \left(P_{\tau_1|\cdots|\tau_k}(\tau_i) \cdot \sum_{p:\rho \in \mathfrak{fp}_{\tau_i}(\sigma)} p \right)$$

However, because of the induction assumption that the Equation 9.1 holds for mst's τ_i with maximal nesting depth $n-1$, for all i $\sum_{p:\rho \in \mathfrak{fp}_{\tau_i}(\sigma)} p = 1$, and since $P_{\tau_1|\cdots|\tau_k}$ is a discrete probability distribution function, we finally arrive to

$$\sum_{p:\rho \in \mathfrak{fp}_{\tau_1|\cdots|\tau_k}(\sigma)} p = \sum_{0 \leq i \leq k} P_{\tau_1|\cdots|\tau_k}(\tau_i) = 1$$

sequence: For the sequence mst $\tau_1 \circ \cdots \circ \tau_k$, we have

$$\mathfrak{fp}_{\tau_1 \circ \cdots \circ \tau_k}(\sigma) = \left\{ \prod_{i=1}^{k} p_i{:}(\rho_1 \bullet \cdots \bullet \rho_k) | \forall 1 \leq i \leq k : yields_p(\tau_i, \sigma, p_i{:}\rho_i) \right\}$$

and subsequently,

$$\sum_{p:\rho \in \mathfrak{fp}_{\tau_1 \circ \cdots \circ \tau_k}(\sigma)} p = \sum_{\prod_{i=1}^{k} p_i:(\rho_1 \bullet \cdots \bullet \rho_k) \in \mathfrak{fp}_{\tau_1 \circ \cdots \circ \tau_k}(\sigma)} \prod_{i=1}^{k} p_i \tag{9.2}$$

Observe that if we fix the update sequence suffix $\rho_2 \bullet \cdots \bullet \rho_k$, the sum 9.2 can be rewritten as

$$\left(\sum_{p_1:\rho_1 \in \mathfrak{fp}_{\tau_1}(\sigma)} p_1 \right) \cdot \left(\sum_{\prod_{i=2}^{k} p_i:(\rho_2 \bullet \cdots \bullet \rho_k) \in \mathfrak{fp}_{\tau_2 \circ \cdots \circ \tau_k}(\sigma)} \prod_{i=2}^{k} p_i \right)$$

Finally, by reformulation of the sum of products 9.2 as a product of sums and by applying the induction assumption for the mst's $\tau_1, \ldots, \tau_k$ of nesting depth $n-1$, we arrive to

$$\prod_{i=1}^{k} \sum_{p:\rho \in \mathfrak{fp}_{\tau_i}(\sigma)} p = \prod_{i=1}^{k} 1 = 1$$

Hence, the Equation 9.1 is satisfied.

□

Finally, the operational semantics of a *P-BSM* agent is defined as a set of traces in the induced transition system enabled by the *P-BSM* agent program.

Definition 9.6 (P-BSM operational semantics). A *P-BSM* $\mathcal{A}_p = (\mathcal{M}_1, \ldots, \mathcal{M}_n, \mathcal{P}, \Pi)$ can make a step from state σ to a state σ' with probability p, iff $\sigma' = \sigma \bigoplus \rho$, s.t. $p{:}\rho \in \mathfrak{fp}_\tau(\sigma)$. We also say that with probability p, $\mathcal{A}_p$ induces a, possibly compound, transition $\sigma \xrightarrow{p:\rho} \sigma'$.

A possibly infinite sequence of states $\omega = \sigma_1, \ldots, \sigma_i, \ldots$ is a run of *P-BSM* $\mathcal{A}_p$, iff for each $i \geq 1$, $\mathcal{A}$ induces the transition $\sigma_i \xrightarrow{p_i:\rho_i} \sigma_{i+1}$ with probability p_i.

Let $pref(\omega)$ denote the set of all finite prefixes of a possibly infinite computation run ω and $|.|$ the length of a finite run. $P(\omega) = \prod_{i=1}^{|\omega|} p_i$ is then the probability of the finite run ω.

The semantics of an agent system characterized by a *P-BSM* $\mathcal{A}_p$, is a set of all runs ω of $\mathcal{A}_p$, s.t., all of their finite prefixes $\omega' \in pref(\omega)$ have probability $P(\omega') > 0$.

Informally, the semantics of an agent system is a set of runs involving only transitions induced by updates with a non-zero selection probability. Additionally, we require an admissible *P-BSM* interpreter to fulfill the following specialisation of the weak fairness condition for the plain *BSM* framework (cf. Definition 2.15), for all the induced runs.

Condition 9.7 (P-BSM weak fairness condition). Let ω be a possibly infinite computation run of a *P-BSM* $\mathcal{A}_p$. Let also $freq_{p:\rho}(\omega')$ be the number of transitions induced by the update $p{:}\rho$ along a finite prefix of $\omega' \in pref(\omega)$.

We say that ω is *weakly fair w.r.t.* $\mathcal{A}_p$ iff for all updates $p{:}\rho$ we have that if from some point on $p{:}\rho$ is always yielded in states along ω, 1) it also occurs on ω infinitely often, and 2) for the sequence of finite prefixes of ω ordered according to their length holds

$$\liminf_{\substack{|\omega'|\to\infty \\ \omega'\in pref(\omega)}} \frac{freq_{p:\rho}(\omega')}{|\omega'|} \geq p$$

Similarly to the plain *BSM* weak fairness condition, Condition 9.7 embodies a minimal requirement on admissible *P-BSM* interpreters. It admits only *P-BSM* interpreters which honor the intended probabilistic semantics of the non-deterministic choice selection of the $yields_p$ calculus. The first part of the requirement is a consequence of the plain *BSM* weak fairness condition, while the second states that in sufficiently long computation runs, the frequency of occurrence of an always yielded probabilistic update corresponds to its selection probability in each single step.

9.2 Jazzyk(P)

Jazzyk is a programming language instantiating the plain *BSM* theoretical framework introduced in Chapter 5. This section informally describes its extension *Jazzyk(P)*, an instantiation of the framework of *Probabilistic Behavioural State Machines* introduced above.

Jazzyk(P) syntax differs from that of *Jazzyk* only in specification of probability distributions over choice mst's. *Jazzyk(P)* allows for explicit labellings of choice mst members by their individual application probabilities. Consider the following labelled choice mst $p_1{:}\tau_1$; $p_2{:}\tau_2$; $p_3{:}\tau_3$; $p_4{:}\tau_4$ in the *Jazzyk(P)* notation. Each $p_i \in [0,1]$ denotes the probability of selection of mst τ_i by the interpreter. Furthermore, to ensure that the labelling denotes a probability distribution over the choice mst, *Jazzyk(P)* parser requires that $\sum_{i=1}^{k} p_i = 1$ for every choice mst $p_1{:}\tau_1 ; \ldots ; p_k{:}\tau_k$ occurring in the considered agent program. Similarly to *Jazzyk*, during the program interpretation phase, *Jazzyk(P)* interpreter proceeds in a top-down manner subsequently considering nested mst's from the main agent program, finally down to primitive update formulae. When the original *Jazzyk* interpreter faces a selection from a non-deterministic choice mst, it randomly

Listing 9.1 Example of *Jazzyk(P)* syntax. Adapted from the *Jazzbot* project. The operators subscripted with $\mathcal{M}$, correspond to *Jazzbot*'s KR module handling the topological map of the environment.

```
when ⊨_B [{ threatened }] then {
        /* ***Emergency modus operandi*** */

        /* Detect the enemy's position */
    0.7 : when ⊨_B [{ attacker(Id) }] and ⊨_E [{ eye see Id player Pos }]
    then ⊘_M [{ positions[Id] = Pos }] ;

    /* Check the camera sensor */
    0.2 : when ⊨_E [{ eye see Id Type Pos }] then {
        ⊕_B [{ see(Id, Type) }] ,
        ⊘_M [{ objects[Pos].addIfNotPresent(Id) }]
    }

    /* Check the body health sensor */
    when ⊨_E [{ body health X }] then ⊕_B [{ health(X). }]
} else {
    /* ***Normal mode of perception*** */

    /* Check the body health sensor */
    when ⊨_E [{ body health X }] then ⊕_B [{ health(X). }] ;

    /* Check the camera sensor */
    when ⊨_E [{ eye see Id Type Pos }] then {
        ⊕_B [{ see(Id, Type) }] ,
        ⊘_M [{ positions[Id] = Pos }]
    }
}
```

selects one of them assuming a discrete uniform probability distribution. I.e., the probability of selecting from a choice mst with k members is $\frac{1}{k}$ for each of them. The extended interpreter *Jazzyk(P)* honors the specified selection probabilities. It generates a random number $p \in [0, 1]$ and selects τ_s, s.t. $\sum_{i=1}^{s-1} p_i \leq p \leq \sum_{i=1}^{s} p_i$.

For convenience, *Jazzyk(P)* enables use of incomplete labellings. An *incompletely labelled* non-deterministic choice mst is one, containing at least one member mst without an explicit probability specification such as $p_1{:}\tau_1 \,;\, p_2{:}\tau_2 \,;\, \tau_3 \,;\, \tau_4$. In such a case, the *Jazzyk(P)* parser automatically completes the distribution by uniformly dividing the remaining probability range to unlabelled mst's. I.e., provided an incompletely labelled choice mst with k members, out of which $m < k$ are labelled, $p_1{:}\tau_1 \,; \ldots ; \, p_m{:}\tau_m \,;$ $\tau_{m+1} \,; \cdots \,; \, \tau_k$, it assigns the residual probability p to the remaining mst's $\tau_{m+1}, \ldots, \tau_k$. The probability p is then calculated as follows

$$p = \frac{1 - \sum_{i=1}^{m} p_i}{k - m} \tag{9.3}$$

Listing 9.1 provides an example of a *Jazzyk(P)* code snippet adapted from the *Jazzbot*

Listing 9.2 Example of focusing bot's attention during emergency situations rewritten with reusable macros.

```
when ⊨B [{ threatened }] then {
    /* ***Emergency modus operandi*** */
    0.7 : DETECT_ENEMY_POSITION ;
    0.2 : SENSE_CAMERA ;
          SENSE_HEALTH
} else {
    /* ***Normal mode of perception*** */
    SENSE_HEALTH ;
    SENSE_CAMERA
}
```

project described in Chapter 7. The code chunk implements an example perception mst of the agent. In the normal mode of operation, the bot in a single step queries either its camera or its body health status sensor with the same probability of selection for each of them, i.e., 0.5. However, in the case of emergency, the bot focuses more on escaping the attacker, therefore, in order to retrieve the attacker's position, it queries the camera sensor more often (selection probability $p = 0.7$) than sensing objects around it ($p = 0.2$). Checking it's own body health is of the least importance ($p = 0.1$), however not completely negligible.

In an implemented program, however, the Listing 9.1 would be rewritten using the macro facility of the *Jazzyk* interpreter and reduced to a more concise code shown in Listing 9.2.

9.3 Adjustable deliberation

The proposed extension allows finer grained steering of the interpreter's non-deterministic selection mechanism and has applications across several niches of methodologies for rule-based agent-oriented programming languages. The first experiments in the context of the *Jazzbot* application have shown that labelling probabilistic mst's is a useful means to contextually focus agent's perception (cf. Listing 9.2). Similarly, the labelling technique is useful in contexts, when it is necessary to execute certain behaviours with a certain given frequency. For example, approximately in about every 5th step broadcast a ping message to peers of the agent's team. Finally, the technique can be used when the agent developer has an informal intuition that preferring more frequent execution of certain behaviours over others might suffice, or even perform better in a given context. For instance, when we want to prefer cheaper, but less efficient behaviours over resource intensive, but rather powerful. Of course writing such programs makes sense only when a rigorous analysis of the situation is impossible or undesirable and at the same time a suboptimal performance of the agent system is acceptable.

An instance of the latter technique for modifying the main control cycle of the agent

program is what I coin *adjustable deliberation*. Consider the following plain *Jazzyk* code implementing *Jazzbot*'s main control cycle as listed in Listing 7.1:

```
PERCEIVE ; HANDLE_GOALS ; ACT
```

The macros PERCEIVE, HANDLE_GOALS and ACT encode behaviours for perception, similar to that in the Listing 9.1, goal commitment strategy implementation and action selection respectively. In the case of emergency, it might be useful to slightly neglect deliberation about agent's goals, in favour of an intensive environment observation and quick reaction selection. The following reformulation of the agent's control cycle demonstrates the simple program modification:

```
when ⊨_B [{ emergency }] then { PERCEIVE ; HANDLE_GOALS ; ACT }
                          else { 0.4 : PERCEIVE ; HANDLE_GOALS ; 0.4 : ACT }
```

9.4 Summary

The presented chapter is based on my recent paper (Novák, 2009b). Its main contribution is introduction of *Probabilistic Behavioural State Machines* framework, with the associated agent programming language *Jazzyk(P)*. *P-BSM*, and in turn *Jazzyk(P)*, allow labelling of alternatives in non-deterministic choice mental state transformers. Thus, they provide a means for specification of a probability distribution over the set of enabled transitions for the next step in agent's life-cycle. In turn, the informal reading of a *P-BSM* choice mst's can be "a specification of the probability, with which the next transition will be chosen from the function denoted by the particular member mst". In other words, *a probability of applying the member mst function to the current mental state*.

Finally, its worth to note that the introduced language *Jazzyk(P)* is a proper extension of the plain *Jazzyk*. As I note in Subsection 5.2.1, *Jazzyk* assumes uniform probability distribution over components of non-deterministic choice mst's. On the other hand, plain mst's can be seen as incompletely labelled probabilistic mst's. The corresponding residual probabilities are computed by the *Jazzyk(P)* interpreter according to Equation 9.3, i.e., *Jazzyk(P)* assumes uniform distribution over the components of the choice mst. In consequence, w.l.o.g. *Jazzyk* interpreter can treat every plain agent program as a probabilistic one. This relation, however, does not necessarily hold between *BSM* and *P-BSM* frameworks. The *BSM* abstract interpreter (cf. Section 2.3) does not have to satisfy the weak fairness condition, but must not necessarily assume uniform distribution on yielded updates.

Chapter 10

Embedding GOAL in Behavioural State Machines

The *BSM* framework is a relatively new contribution to the field of agent programming languages. While it clearly aims at pragmatic issues in programming cognitive agents with mental attitudes, by providing a solid and crisp semantics, it falls into the category of theoretically founded experimental agent programming frameworks (for an overview, cf. Chapter 1). While most of these languages facilitate programming agent systems with mental attitudes, at the same time, they sometimes significantly differ in some aspects of the underlying theory and background ideas. The task to compare them and establish the relative expressive power thus becomes crucial to the advancement of the research field.

On the other hand, implementation and rapid prototyping of experimental agent-oriented programming languages from scratch in e.g., *Java*, is a large, non-trivial and error-prone effort. Moreover, a disadvantage of such an effort is that it is difficult to ascertain that such an implementation is faithful to the agent language semantics. Therefore, it would be useful to have an *intermediate language* that provides *a core instruction set* of more high-level programming constructs than e.g., *Java* provides, and that could be used to compile agent programs into. I.e., considering only plain programming languages *per se*, there is a need for a core agent programming language which

1. is *minimal* w.r.t. the set of language constructs, yet
2. facilitates *semantics preserving translations* from other agent-oriented programming languages into directly executable programs, and thus
3. provides *enough expressive power*, so that it enables a natural link to formal methods enabling reasoning about agents in terms of *mental attitudes*.

Mainstream languages, such as *Java*, do not facilitate *ex post* reasoning about developed systems in terms of mental attitudes. The main reason is that even though they provide a concise general-purpose instruction set, translating agent programs to them is not semantics preserving. The resulting programs are executed in terms of imperative or

object-oriented instruction flows, rather than in terms of interactions between agent's mental attitudes.

Unlike the *BSM* framework, the language *GOAL*, by Hindriks (2001), provides first-class agent-oriented notions such as beliefs and goals. In this chapter, I will show how agent programs written in *GOAL* can be translated in a semantics-preserving manner to *BSM* agent programs. As it turns out, the *BSM* agent programming framework provides an interesting option for compiling agent programs into. While the requirement of the language minimality stated above is not approached directly, the translation demonstrates that the *BSM* framework does indeed provide a sufficient architecture and a set of primitives, needed to emulate agent languages, such as *GOAL*. The main contribution of the chapter is a formal proof, showing that it is relatively easy to compile *GOAL* agents into *BSM* programs. It also demonstrates the usefulness of *BSM,* and in turn also *Jazzyk*, as a target language of an agent program compiler. In the following, after a brief overview of the *GOAL* language, I introduce the translation function $\mathfrak{C}$ from *GOAL* to *BSM* agent programs. For simplicity, in the transformation, I consider only ground programs, i.e., those without variables.

10.1 GOAL

The agent programming language *GOAL*, *Goal-Oriented Agent Language*, was first introduced by Hindriks (2001) and later described and improved in (de Boer et al., 2007) and (Hindriks, 2009). It incorporates declarative notions of beliefs and goals, and a mechanism for action selection based on these notions. That is, *GOAL* agents derive their choice of action from their beliefs and goals.

Definition 10.1 (GOAL agent). A *GOAL* agent $\mathcal{A}$ represented as a tuple $\mathcal{A} = \langle \Sigma, \Gamma, \Pi, A \rangle$, consists of four sections:

1. a set of beliefs Σ, collectively called the *belief base*,
2. a set of goals Γ, called the *goal base*,
3. a *program section* Π, which consists of a set of *action rules*, and
4. an *action specification section* A that consists of a specification of the pre- and postconditions of actions of the agent.

For illustration, Listing 10.1 lists an example of a simplified *GOAL* agent that manipulates blocks on a table.

10.1.1 Beliefs and Goals

The beliefs and goals of a *GOAL* agent are drawn from a *KR language* such as *Prolog* (Shapiro and Sterling, 1994). In the following, however, I abstract from particularities

Listing 10.1 Example of a *GOAL* agent program manipulating blocks on the table in a blocksworld environment.

```
:main: blocksWorld {
        //*** Initializations omitted ***/
        :beliefs{...}

        :goals{...}

        :program{
                if bel(on_table([B|S]), clear(B), block(C), clear(C)), goal(on_table([C,B|S]))
                then move(C,B).

                if goal(on(B,A)), bel(on_table([C|S]), clear(C), member(B,S))
                then move(C,table).
        }

        :actionspec{
                move(X,Y) {
                        :pre{ clear(X), clear(Y), on(X,Z), not(on(X,Y)) }
                        :post{ not(on(X,Z)), on(X,Y) }
                }
        }
}
```

of a specific KR language and consider only a template of the KR technology used in *GOAL*.

Definition 10.2 (KR Technology). A *KR technology* is a triple $\langle \mathcal{L}, \mathcal{Q}, \mathcal{U} \rangle$, where:

- $\mathcal{L}$ is some logical language, with a typical element $\phi \in \mathcal{L}$,
- $\mathcal{Q}$ is a set of query operators $\models \in \mathcal{Q}$ such that $\models \subseteq 2^{\mathcal{L}} \times \mathcal{L}$,
- $\mathcal{U}$ is a set of update operators $\oslash \in \mathcal{U}$ of type : $2^{\mathcal{L}} \times \mathcal{L} \rightarrow 2^{\mathcal{L}}$.

The definition above is quite abstract and only specifies the types of operators which are associated with a knowledge representation language. It allows for a wide range of KR technologies that fit introduced KR schema. The only assumption made, is that the special symbol $\bot$ is part of the KR language $\mathcal{L}$. Intuitively, it is interpreted as *falsum*, i.e., when $\bot$ can be derived from a set of sentences this set is said to be *inconsistent*. The definition is inspired by the discussion on the foundations of knowledge representations by Davis et al. (1993). Apart from minor differences, it corresponds to the notion of a KR module in the *BSM* framework (cf. Definition 2.1).

W.l.o.g we can assume that *GOAL* is defined over a specific schema of a KR technology $\mathcal{K}_0 = \langle \mathcal{L}, \{\models\}, \{\oplus, \ominus\} \rangle$, where $\models$ is an entailment relation on $\mathcal{L}$, $\oplus$ is a revision operator and $\ominus$ is a contraction operator. $\models$ is used to verify that a sentence follows from a particular set of sentences. $\oplus$ adds a new sentence to a the set and finally, $\ominus$ removes

(contracts) a sentence from a set of sentences. Both $\oplus$ as well as $\ominus$ are assumed to yield consistent sets of sentences, i.e., for arbitrary theory T and ϕ, we have that $T \oplus \phi \not\models \bot$ and $T \ominus \phi \not\models \bot$. From now on, I use the label $\mathcal{K}_0$ to refer to arbitrary KR technologies following this scheme and used by GOAL agents.

The belief base Σ and the goal base Γ of a *GOAL* agent are defined as subsets of sentences, a theory, from the KR language $\mathcal{L}$. Together the belief and the goal base of an agent make up a *mental state* m of a GOAL agent.

Definition 10.3 (GOAL agent mental state). $m = \langle \Sigma, \Gamma \rangle$ is said to be a *GOAL* agent mental state, when

1. the belief base Σ is consistent, i.e., $\Sigma \not\models \bot$,
2. the individual goals $\gamma \in \Gamma$ are consistent, i.e., $\{\gamma\} \not\models \bot$ for each γ.
3. additionally, the agent does not believe it achieved its goals, i.e., for all $\gamma \in \Gamma$, we have $\Sigma \not\models \gamma$.

10.1.2 Action Selection and Specification

A *GOAL* agent chooses an action by means of a rule-based action selection mechanism. A program section in a *GOAL* agent consists of *action rules* of the form

$$\textbf{if } \psi \textbf{ then } \mathtt{a}.$$

These action rules define a mapping from states to actions, together specifying a non-deterministic policy or course of action. The condition of an action rule, typically denoted by ψ, is called a *mental state condition*. It determines the states in which the action `a` may be executed. Mental state conditions are Boolean combinations of basic formulae $\textbf{bel}(\phi)$ or $\textbf{goal}(\phi)$, with $\phi \in \mathcal{L}$. For example, $\neg\textbf{bel}(\phi_0) \wedge \textbf{goal}(\phi_0 \wedge \phi_1)$ is a mental state condition.

Definition 10.4 (mental state condition semantics). Given a mental state $m = \langle \Sigma, \Gamma \rangle$, the semantics of a mental state condition, is defined by the following four clauses:

$$\begin{array}{lll} m \models_g \textbf{bel}(\phi) & \text{iff} & \Sigma \models \phi, \\ m \models_g \textbf{goal}(\phi) & \text{iff} & \text{there is a } \gamma \in \Gamma \text{ s.t. } \{\gamma\} \models \phi, \\ m \models_g \neg\psi & \text{iff} & m \not\models_g \psi, \\ m \models_g \psi_1 \wedge \psi_2 & \text{iff} & m \models_g \psi_1 \text{ and } m \models_g \psi_2. \end{array}$$

Actions are specified in *GOAL* using a STRIPS-like specification.

The action specification section consists of specifications of the form

$$\textbf{action}\{\ \textbf{:pre}\{\phi\}\ \ \textbf{:post}\{\phi'\}\ \}. \tag{10.1}$$

Such a specification of an action **action** consists of a precondition ϕ and a postcondition ϕ'. An action is *enabled* whenever the agent believes the precondition to be true. Upon its execution, the agent updates its beliefs by the postcondition ϕ'. Thereby, indirectly, it can update also its goals. In line with STRIPS-style action specifications, the postcondition ϕ' of an action consists of two parts $\phi' = \phi_d \wedge \phi_a$. ϕ_d denotes a list of negative literals (negated facts), also called the *delete list*, and ϕ_a stands for a conjunction of positive literals (facts), also called the *add list*. Each action is assumed to match with exactly one corresponding action specification.

10.1.3 Semantics of a GOAL Agent

The semantics of a *GOAL* agent execution is defined in terms of Plotkin-style transition semantics (Plotkin, 1981). It is sufficient to present here the semantics for executing a single action by a *GOAL* agent. In Section 10.2, I show how this semantics can be *implemented* by means of a *BSM*.

Definition 10.5 (action semantics). Let $m = \langle \Sigma, \Gamma \rangle$ be a mental state, **if** ψ **then** `a` be an action rule, and `a` {**:pre**$\{\phi\}$ **:post**$\{\phi_a \wedge \phi_d\}$ } be a corresponding action specification of a *GOAL* agent. The following semantic rule is used to derive that action `a` can be executed

$$\frac{m \models \psi, \Sigma \models \phi}{m \xrightarrow{\texttt{a}} m'}$$

$\Sigma' = (\Sigma \ominus \phi_d) \oplus \phi_a$ and $m' = \langle \Sigma', \Gamma \setminus \{\gamma \in \Gamma | \Sigma' \models \gamma\} \rangle$. We also say that $m \xrightarrow{\texttt{a}} m'$ is a *possible computation step* in m.

Besides user specified actions, *GOAL* has two built-in actions **adopt** and **drop**, modifying agent's goal base. The following axioms define their semantics

$$\begin{aligned} \langle \Sigma, \Gamma \rangle &\xrightarrow{\mathbf{adopt}(\phi)} \langle \Sigma, \Gamma \cup \{\phi\} \rangle \\ \langle \Sigma, \Gamma \rangle &\xrightarrow{\mathbf{drop}(\phi)} \langle \Sigma, \Gamma \setminus \{\gamma \in \Gamma \mid \{\gamma\} \models \phi\} \rangle \end{aligned} \tag{10.2}$$

The **drop** action features a non-trivial semantics. Besides removing the goal formula ϕ from the goal base Γ, if it is present there, it also retracts all the goals, which can indirectly cause the agent to further pursue the goal ϕ.

10.2 Compiling a GOAL agent into a BSM

This section shows that *GOAL* agents can be implemented as, or compiled into, *Behavioural State Machines*. The compiler is abstractly represented by a function $\mathfrak{C}$ that translates, compiles, a *GOAL* agent into a *BSM*. The main result is a proof that for every

GOAL agent $\mathcal{A} = \langle \Sigma, \Gamma, \Pi, \mathcal{A} \rangle$, there is a *BSM* $\mathfrak{C}(\mathcal{A}) = (\mathcal{M}_1, \ldots, \mathcal{M}_n, \tau)$ implementing that *GOAL* agent. In fact, I will show that a *BSM* $\mathfrak{C}(\mathcal{A}) = (\mathcal{M}_\Sigma, \mathcal{M}_\Gamma, \tau)$ with precisely two KR modules, $\mathcal{M}_\Sigma$ corresponding to the belief base Σ and $\mathcal{M}_\Gamma$ realizing the goal base Γ, implements the original *GOAL* agent $\mathcal{A}$.

To show the transformation, I will first define the KR modules $\mathcal{M}_\Sigma$ and $\mathcal{M}_\Gamma$ of the *BSM*. Subsequently, I will show how to construct the *BSM* agent program τ that implements the action rules in the program section Π and action specifications A of the *GOAL* agent. Finally, I will prove the equivalence of the *GOAL* agent with its *BSM* counterpart $\mathfrak{C}(\mathcal{A})$, by showing that both are able to generate the same computation runs.

10.2.1 Knowledge bases

GOAL knowledge bases take the form of KR technologies $\mathcal{K}_0 = \langle \mathcal{L}, \{\models\}, \{\oplus, \ominus\} \rangle$. Because of the close similarity with the notion of KR module, a *GOAL* belief base can be straightforwardly mapped onto a *BSM* KR module, which implements a query corresponding to evaluation of a mental state condition $\mathbf{bel}(\phi)$ on Σ, as well as provides updates corresponding to performing an action in *GOAL*. The *GOAL* belief base Σ can be simply mapped onto a *BSM* module $\mathcal{M}_\Sigma$ of the form

$$\mathcal{M}_\Sigma = \mathfrak{C}_{\mathbf{bb}}(\Sigma) = \langle \Sigma, \mathcal{L}, \{\models\}, \{\oplus, \ominus\} \rangle \tag{10.3}$$

While the underlying KR technology is implicitly assumed in a *GOAL* agent, this assumption is made explicit in the corresponding *BSM* KR module.

The translation of the goal base of a *GOAL* agent into a *BSM* KR module is less straightforward. A KR module that corresponds to the goal base needs to be able to appropriately implement 1) the evaluation of a mental state condition $\mathbf{goal}(\phi)$ on the goal base, as well as 2) the execution of updates on it. Furthermore, the goals that have been achieved by recent belief updates are considered satisfied and must, in turn, be retracted. Because the query operator **goal** has a somewhat non-standard semantics (cf. Definition 10.4), we need to define a non-standard KR technology associated with the KR module implementing the goal base. The following mapping of a goal base Γ onto the module $\mathcal{M}_\Gamma$ provides the needed translation

$$\mathcal{M}_\Gamma = \mathfrak{C}_{\mathbf{gb}}(\Gamma) = \langle \Gamma, \mathcal{L}, \{\models_{\mathbf{goal}}\}, \{\oplus_{\mathbf{adopt}}, \ominus_{\mathbf{drop}}, \ominus_{\mathbf{achieved}}\} \rangle, \tag{10.4}$$

where

- $\Gamma \models_{\mathbf{goal}} \phi$ iff there is a $\gamma \in \Gamma$ such that $\{\gamma\} \models \phi$,
- $\Gamma \oplus_{\mathbf{adopt}} \phi = \Gamma \cup \{\phi\}$,
- $\Gamma \ominus_{\mathbf{drop}} \phi = \Gamma \setminus \{\gamma \in \Gamma \mid \{\gamma\} \models \phi\}$,
- $\Gamma \ominus_{\mathbf{achieved}} \phi = \Gamma \setminus \{\phi\}$.

$\models_{\mathbf{goal}}$ is used to implement $\mathbf{goal}(\phi)$, $\oplus_{\mathbf{adopt}}$ implements **adopt**, $\ominus_{\mathbf{drop}}$ is used to implement **drop**. Finally, $\ominus_{\mathbf{achieved}}$ implements the goal update mechanism to remove achieved goals. Note that $\ominus_{\mathbf{drop}}$ operator cannot be used for removing a goal formula from the goal base. To remove an agent's goal, the goal update mechanism of *GOAL* (cf. Definition 10.4) requires a simple set operator such as $\ominus_{\mathbf{achieved}}$.

10.2.2 Action rules

Using the KR technology translations, it is now possible to translate mental state conditions ψ used in *GOAL* action rules of the form **if** ψ **then** a. As noted above, $\mathfrak{C}(\mathbf{bel}(\phi))$ can be mapped onto the *BSM* query $\models \phi$. Similarly, we can define $\mathfrak{C}(\mathbf{goal}(\phi)) = (\models_{\mathbf{goal}} \phi)$. Boolean combinations of mental state conditions are translated into Boolean combinations of compound *BSM* queries.

In the case, when a is either **adopt** or **drop** action, its translation into a *BSM* mental state transformer is rather straightforward. Since both $\mathbf{adopt}(\phi)$ and $\mathbf{drop}(\phi)$ are always enabled, they are simply mapped to corresponding primitive update operators as follows

$$\begin{aligned} \mathfrak{C}(\mathbf{adopt}(\phi)) &= \oplus_{\mathbf{adopt}}\phi \\ \mathfrak{C}(\mathbf{drop}(\phi)) &= \ominus_{\mathbf{drop}}\phi \end{aligned} \tag{10.5}$$

Compilation of user defined actions, i.e., actions specified in the action specification section A, into a *BSM* depends on the action specification A of the original *GOAL* agent. Such actions are mapped to conditional mst's of the form $\varphi \longrightarrow \tau$. The preconditions of an action are mapped to the query part φ of the *mst*. Similarly, the effects of that action, expressed by a postcondition in *GOAL*, are translated into a sequential mst τ. Assuming that a is a *GOAL* action with the corresponding action specification a {**:pre**$\{\phi\}$ **:post**$\{\phi_d \wedge \phi_a\}$, we define

$$\mathfrak{C}(\mathtt{a}) = (\models \phi \longrightarrow \ominus\phi_d \circ \oplus\phi_a) \tag{10.6}$$

Note that the *BSM* operators $\models$, $\oplus$, and $\ominus$ are associated with the KR module $\mathcal{M}_\Sigma$, implementing the belief base of the *GOAL* agent, i.e., the precondition ϕ is evaluated w.r.t. the set of agent's beliefs. In line with Definition 10.5, the postcondition $\phi_d \wedge \phi_a$ is used to update that belief base.

Combining the translations of mental state conditions and actions yields a translation of action rules in the program section of a *GOAL* agent. It is also convenient to introduce a translation of a complete program section Π. Note that the order of translation is unimportant. The *GOAL* program section translation follows.

$$\begin{aligned} \mathfrak{C}(\mathbf{if}\ \psi\ \mathbf{then}\ \mathtt{a}) &= \mathfrak{C}(\psi) \longrightarrow \mathfrak{C}(\mathtt{a}) \\ \mathfrak{C}(\emptyset) &= \mathbf{skip} \\ \mathfrak{C}(\Pi) &= \mathfrak{C}(r) \,|\, \mathfrak{C}(\Pi \setminus \{r\}) \text{ , if } r \in \Pi \end{aligned} \tag{10.7}$$

Finally, we need to ensure that the resulting *BSM* implements the *blind commitment strategy* of the *GOAL* language. I.e., a goal is removed, whenever it is believed to be completely achieved. To this end, it is convenient to introduce the notion of a *possibly adopted goal*. ϕ is said to be a *possibly adopted goal*, whenever it is possible that the agent may come to adopt ϕ as a goal. I.e., whenever it is already present in the goal base, or there is an action rule of the form **if** ψ **then adopt**(ϕ) in Π. Thus, the set of possibly adopted goals $\mathcal{P}_{\mathcal{A}}$ of a *GOAL* agent $\mathcal{A} = \langle \Sigma, \Gamma, \Pi, A \rangle$ can be defined by

$$\mathcal{P}_{\mathcal{A}} = \Gamma \cup \{\phi \mid \textbf{if } \psi \textbf{ then adopt}(\phi) \in \Pi\}. \tag{10.8}$$

We can model the part of the resulting *BSM*, responsible for the implementation of the *GOAL*'s blind commitment strategy by a sequential mst that consists of a sequence of conditional removals of goals from $\mathcal{P}_{\mathcal{A}}$. In the case $\phi \in \mathcal{P}_{\mathcal{A}}$ is believed to be achieved, ϕ must removed from the goal base.

$$\begin{aligned} \mathfrak{C}_{\mathbf{bcs}}(\emptyset) &= \textbf{skip} \\ \mathfrak{C}_{\mathbf{bcs}}(\mathcal{P}_{\mathcal{A}}) &= (\models \phi \longrightarrow \ominus_{\mathbf{achieved}}\phi) \circ \mathfrak{C}_{\mathbf{bcs}}(\mathcal{P}_{\mathcal{A}} \setminus \{\phi\}) \text{ , if } \phi \in \mathcal{P}_{\mathcal{A}} \end{aligned} \tag{10.9}$$

Finally, putting all the pieces of the puzzle together, we arrive to translation of *GOAL* agent into a *BSM*.

Definition 10.6 (translation of *GOAL* agent to *BSM*). The compilation of a *GOAL* agent $\langle \Sigma, \Gamma, \Pi, A \rangle$ into a *BSM* is defined as

$$\mathfrak{C}(\langle \Sigma, \Gamma, \Pi, A \rangle) = (\mathcal{M}_{\Sigma}, \mathcal{M}_{\Gamma}, \mathfrak{C}(\Pi) \circ \mathfrak{C}_{\mathbf{bcs}}(\mathcal{P}_{\mathcal{A}})).$$

Listing 10.2 shows the *GOAL* program section from Listing 10.1 translated into a corresponding mental state transformer.

10.2.3 Correctness of the Translation Function $\mathfrak{C}$

The main effort in proving that the compilation of a *GOAL* agent $\mathcal{A} = \langle \Sigma, \Gamma, \Pi, A \rangle$ into the *BSM* $\mathfrak{C}(\mathcal{A}) = (\mathcal{M}_{\Sigma}, \mathcal{M}_{\Gamma}, \mathfrak{C}(\Pi) \circ \mathfrak{C}_{\mathbf{bcs}}(\mathcal{P}_{\mathcal{A}}))$ is correct, consists of showing that the action rules Π of the *GOAL* agent generate the same sequences of mental states as the mental state transformer $\mathfrak{C}(\Pi) \circ \mathfrak{C}_{\mathbf{bcs}}(\mathcal{P}_{\mathcal{A}})$. In order to prove this, it is convenient to first show some useful properties of the translation. Lemma 10.7 shows that $\mathfrak{C}_{\mathbf{bcs}}(\mathcal{P}_{\mathcal{A}})$ implements the goal update mechanism of *GOAL*. Subsequently, Lemma 10.8 provides the relation of *GOAL* mental states resulting from action execution to the application of updates to *BSM* mental states. Finally, Lemma 10.9 shows that the evaluation of mental state conditions in *GOAL* corresponds to the evaluation of their translations in *BSM*. The proofs are rather straightforward and follow from definitions above.

Lemma 10.7, below, shows that a *BSM* state after removing goals that are believed to be achieved is also a proper *GOAL* mental state. Furthermore, it shows that the mst $\mathfrak{C}_{\mathbf{bcs}}(\mathcal{P}_{\mathcal{A}})$ implements this goal update mechanism.

Listing 10.2 Translation of the *GOAL* program from Listing 10.1 to a corresponding *Jazzyk* program.

```
/*** Modules initialization omitted ***/
{
        /*** 𝔈(Π) ***/
        when ⊨ [{on_table([B|S]), clear(B), block(C), clear(C)}] and ⊨goal [{on_table([C,B|S])}] then {
                when ⊨ [{clear(C), clear(B), on(C,Z), not(on(C,B))}]
                then ⊕ [{not(on(C,Z)), on(C,B)}]
        } ;
        when ⊨goal [{on(B,A)}] and ⊨ [{on_table([C|S]), clear(C), member(B,S)}] then {
                when ⊨ [{clear(C), clear(table), on(C,Z), not(on(C,table))}]
                then ⊕ [{not(on(C,Z)), on(C,table)}]
        }
} ,
{
        /*** 𝔈bcs(P_A)) ***/
        when ⊨ [{on(b,a), on(a,table)}] then ⊖goal [{on(b,a), on(a, table)}] ,
        when ⊨ [{on_table([a,b])}] then ⊖goal [{on_table([a,b])}] ,
        when ⊨ [{on_table([b])}] then ⊖goal [{on_table([b])}]
}
```

Lemma 10.7. *Let $m = \langle \Sigma, \Gamma \rangle$ be a BSM state, such that $\Sigma \not\models \bot$. Let also $\Gamma \subseteq \mathcal{P}_{\mathcal{A}}$ and ρ be an update $\ominus_{\mathbf{achieved}}\gamma_1 \bullet \ldots \bullet \ominus_{\mathbf{achieved}}\gamma_n$. Then yields$(\mathfrak{C}_{\mathbf{bcs}}(\mathcal{P}_{\mathcal{A}}), m, \rho)$ if and only if*

1. *$\langle \Sigma, \Gamma \bigoplus \rho \rangle$ is a GOAL mental state according to Definition 10.3, and*
2. *there is no Γ', s.t., $\Gamma \bigoplus \rho \subseteq \Gamma' \subset \Gamma$, where $\langle \Sigma, \Gamma' \rangle$ is a GOAL mental state.*

Proof. Both directions follow immediately from the construction of $\mathfrak{C}_{bcs}(\mathcal{P}_{\mathcal{A}})$ (Equation 10.9) and $\mathcal{P}_{\mathcal{A}}$ (Equation 10.8). We have that $\gamma_1, \ldots, \gamma_n$ are *all* the goals satisfied w.r.t. the current agent's beliefs, i.e., $\forall 1 \leq i \leq n : \Sigma \models \gamma_i$. After removing them from Γ, the resulting mental state $\langle \Sigma, \Gamma \bigoplus \rho \rangle$ satisfies the Definition 10.3, hence point 1 holds. Similarly, from the construction of $\mathfrak{C}_{bcs}(\mathcal{P}_{\mathcal{A}})$ we have that no unsatisfied goal (w.r.t. Σ) was removed from Γ by application of ρ. I.e., $\Gamma \bigoplus \rho$ is the result of a *minimal change* on Γ, so that it becomes a proper *GOAL* mental state. Hence, also point 2 holds. □

The following Lemma 10.8 furthermore proves that the *GOAL* states resulting from executing an action can also be obtained by applying updates of a particular structure. This allows relating *GOAL* actions to *BSM* updates.

Lemma 10.8. *Let $\mathcal{A} = \langle \Sigma, \Gamma, \Pi, A \rangle$ be a GOAL agent and $\mathfrak{C}(\mathcal{A}) = (\mathcal{M}_\Sigma, \mathcal{M}_\Gamma, \tau)$ its BSM compilation. Also let* a *be a user defined action of GOAL agent $\mathcal{A}$, with action specification* a {**:pre**$\{\phi\}$ **:post**$\{\phi_a \wedge \phi_d\}$}*. Then*

1. $m \xrightarrow{\mathtt{a}} m' \iff \exists n \geq 0 : m' = m \bigoplus (\ominus\phi_d \bullet \oplus\phi_a \bullet \ominus_{\mathbf{achieved}}\gamma_1 \bullet \ldots \bullet \ominus_{\mathbf{achieved}}\gamma_n)$,

2. $m \xrightarrow{\mathbf{drop}(\phi)} m' \iff m' = m \bigoplus(\ominus_{\mathbf{drop}}\phi)$,
3. $m \xrightarrow{\mathbf{adopt}(\phi)} m' \iff m' = m \bigoplus(\oplus_{\mathbf{adopt}}\phi)$, *and finally*
4. *if* $yields(\tau, m, \rho)$, *then* ρ *is either of the form* $\ominus\phi_d \bullet \oplus\phi_a \bullet \ominus_{\mathbf{achieved}}\gamma_1 \bullet \ldots \bullet \ominus_{\mathbf{achieved}}\gamma_n$ *for some* $n \geq 0$, *or one of the* $\ominus_{\mathbf{drop}}\phi$, *or* $\oplus_{\mathbf{adopt}}\phi$.

Proof sketch.

1. Follows from the *GOAL* action semantics in Definition 10.5, the construction of $\mathfrak{C}(\mathtt{a})$ and Lemma 10.7.

2-3. Again, both follow straightforwardly from the definition of **adopt** and **drop** semantics (Equation 10.2), the constructions of $\mathfrak{C}(\mathbf{adopt}\phi)$ and $\mathfrak{C}(\mathbf{drop}\phi)$ (Equation 10.5).

4. Follows from the construction of $\mathfrak{C}(\mathcal{A})$ in Definition 10.6 and the fact that since **adopt**, nor **drop** (resp. $\oplus_{\mathbf{adopt}}$ and $\ominus_{\mathbf{drop}}$) do not touch the belief base Σ, none of the conditional mst's in $\mathfrak{C}_{bcs}(\mathcal{P}_{\mathcal{A}})$ can yield an mst.

□

Finally, Lemma 10.9 relates the evaluation of *GOAL* mental state conditions to the evaluation of their *BSM* translations in the same state.

Lemma 10.9. *Let* ψ *be a mental state condition. Then the following holds*

$$m \models_g \psi \iff m \models_{\mathbf{goal}} \mathfrak{C}(\psi)$$

Proof. Immediately follows from Definition 10.4 of mental state condition semantics and the construction of $\mathcal{M}_\Gamma$ (Equation 10.4). □

Finally, Theorem 10.10 shows that the updates generated by the translation of a *GOAL* agent to a *BSM* produce the same mental states as the execution of actions by that *GOAL* agent. That shows that the *BSM* implements the *GOAL* agent,the main result of this chapter.

Theorem 10.10 (correctness of the GOAL-2-BSM compilation). *Let* $\mathcal{A} = \langle \Sigma, \Gamma, \Pi, A\rangle$ *be a GOAL agent being in mental state* $m = \langle \Sigma, \Gamma\rangle$. *Let also* $\mathfrak{C}(\mathcal{A}) = (\mathcal{M}_\Sigma, \mathcal{M}_\Gamma, \tau)$ *be its corresponding BSM translation. Then for all* ρ, *we have*

$$\exists \mathtt{a} : m \xrightarrow{\mathtt{a}} m \bigoplus \rho \quad \iff \quad yields(\tau, m, \rho).$$

Proof. Informally, to show the left to right direction ($\Longrightarrow$), we have to show that if a *GOAL* action $\mathtt{a}$ is enabled in a mental state m, there exists an update ρ such that 1) the state resulting from performing $\mathtt{a}$ is $m \bigoplus \rho$, and 2) ρ is yielded by τ in this state. Note that the expression on the left hand side denotes both, a *BSM*, as well as a *GOAL* transition. From Lemma 10.8 we have that such a ρ exists and bears the form

1. $\rho = \ominus\phi_d \bullet \oplus\phi_a \bullet \ominus_{\mathbf{achieved}}\gamma_1 \bullet \ldots \bullet \ominus_{\mathbf{achieved}}\gamma_n$ for user specified actions $\mathtt{a}$,
2. $\rho = \ominus_{\mathbf{drop}}\phi$ if $\mathtt{a} = \mathbf{drop}(\phi)$, and
3. $\rho = \oplus_{\mathbf{adopt}}\phi$ if $\mathtt{a} = \mathbf{adopt}(\phi)$.

Suppose now that $m \xrightarrow{\mathtt{a}} m \bigoplus \rho$ and $\mathtt{a}$ is a user defined action (the other cases dealing with $\mathtt{a} = \mathbf{drop}(\phi)$ and $\mathtt{a} = \mathbf{adopt}(\phi)$ are similar). This means there is an action rule **if** ψ **then** $\mathtt{a}$, a precondition ϕ and a postcondition $\phi_d \wedge \phi_a$ associated with action $\mathtt{a}$. Moreover, we assume that $m \models_g \psi$ and $\Sigma \models \phi$. It remains to show that the corresponding update ρ is also yielded by τ. By construction, we must have that

$$\tau = (...|(\mathfrak{C}(\psi) \longrightarrow (\models \phi \longrightarrow \ominus\phi_d \circ \oplus\phi_a))|...) \circ \mathfrak{C}_{\mathbf{bcs}}(\mathcal{P}_{\mathcal{A}})$$

Since we have $m \models_g \psi$ and $\Sigma \models \phi$, using Lemma 10.9, it is immediate that we have $yields(\mathfrak{C}(\psi) \longrightarrow (\models \phi \longrightarrow \ominus\phi_d \circ \oplus\phi_a), m, \ominus\phi_d \bullet \oplus\phi_a)$. Finally, from Lemma 10.7, we have that $yields(\mathfrak{C}_{\mathbf{bcs}}(\mathcal{P}_{\mathcal{A}}), m \bigoplus(\ominus\phi_d \bullet \oplus\phi_a), \{\ominus_{\mathbf{achieved}}\gamma_1 \bullet \ldots \bullet \ominus_{\mathbf{achieved}}\gamma_n)$ and by applying sequential composition on the resulting updates we are done.

($\Longleftarrow$) For the other direction, we have to prove that the updates performed by $\mathfrak{C}(\mathcal{A})$ correspond to enabled actions of the *GOAL* agent $\mathcal{A}$. So suppose that $yields(\tau, m, \rho)$, and ρ is of the form $\ominus\phi_d \bullet \oplus\phi_a \bullet \ominus_{\mathbf{achieved}}\gamma_1 \bullet \ldots \bullet \ominus_{\mathbf{achieved}}\gamma_n$ (using Lemma 10.8, point 4; the other cases with $\rho = \ominus_{\mathbf{drop}}\phi$ and $\rho = \oplus_{\mathbf{adopt}}\phi$ are again similar). From the construction of $\mathfrak{C}$ it follows that we must have $yields(\mathfrak{C}(\psi) \longrightarrow (\models \phi \longrightarrow \ominus\phi_d \circ \oplus\phi_a) \circ \mathfrak{C}_{\mathbf{bcs}}(\mathcal{P}_{\mathcal{A}}), m, \rho)$. From the rule for conditional mst in the *yields* calculus (cf. Definition 2.9) it follows that $m \models_j \mathfrak{C}(\psi)$ and $m \models_j (\models \phi)$. By Lemma 10.9 we then have $m \models_g \psi$ and $\Sigma \models \phi$. It must also be the case that there exists an action rule **if** ψ **then** $\mathtt{a}$ with action specification $\mathtt{a}$ {**:pre**{ϕ} **:post**{$\phi_a \wedge \phi_d$}, such that $m \xrightarrow{\mathtt{a}} m \bigoplus(\ominus\phi_d \bullet \oplus\phi_a \bullet \ominus_{\mathbf{achieved}}\gamma'_1 \bullet \ldots \bullet \ominus_{\mathbf{achieved}}\gamma'_m)$ (cf. Lemma 10.8, point 1). It remains to show that $\ominus_{\mathbf{achieved}}\gamma_1 \bullet \ldots \bullet \ominus_{\mathbf{achieved}}\gamma_n$ is equal to $\ominus_{\mathbf{achieved}}\gamma'_1 \bullet \ldots \bullet \ominus_{\mathbf{achieved}}\gamma'_m$. This, however, follows immediately from Lemma 10.7. □

10.3 Summary

This chapter presents results, I published earlier in a joint paper with Hindriks (Hindriks and Novák, 2008). Above, I showed that any *GOAL* agent can be compiled into a *Behavioural State Machine* and thus to *Jazzyk* agent program. More precisely, I proved that every possible computation step of a *GOAL* agent can be emulated by the *BSM* resulting from compilation of the *GOAL* agent to a *BSM*.

An elegant consequence of the presented translation is that the compilation procedure is *compositional*. I.e., any modification or extension of the *GOAL* agent's belief base, goal base, the program or the action specification sections only *locally* affects the translated knowledge bases of the resulting *BSM*, as well as the compiled agent.

As mentioned above, *BSM* does not commit to any particular view on the KR modules. This flexibility allowed us to implement the goal base of a *GOAL* agent by means of explicit emulation of the goal update mechanism. On the other hand, it is relatively easy to note that the introduced compilation function provides a means to translate *GOAL* agents into *BSM*s, but the relationship does not hold not *vice versa*.

Conclusion

... where I discuss the broader context and implications of the introduced theoretical and engineering results, give a future work outlook beyond single agent systems and in which I finally conclude the body of this dissertation.

Chapter 11

Part I: Theoretical foundations

The framework of *Behavioural State Machines*, in the form presented in Chapter 2, is a culmination of a sequel of papers (Novák and Dix, 2006, 2007; Novák, 2008a,c; Novák and Jamroga, 2009), presented on various conferences and workshops. The *BSM* framework was originally devised as a generalization of agent programming languages, such as *3APL* by Hindriks et al. (1999) and *AgentSpeak(L)* by Rao (1996) so that it could enable accommodation of heterogeneous knowledge representation technologies in a single agent system. Gradually, however, it became clear that in order to achieve the desired level of generality and interoperability or KR technologies, I have to tackle the more fundamental issues of agent-oriented programming first. Thus, *BSM* came to life, as a programming framework of its own, with development mainly driven by the case-studies presented in Chapter 7. Below, I discuss distinctive features of the *BSM* framework w.r.t. the state-of-the-art BDI agent programming frameworks and related theoretical work. I discuss the most important contributions of the *BSM* framework to the state of the art of cognitive agent programming. I.e., its support for what I call *vertical* and *horizontal modularity*, support for implementing *flexible model of agent reasoning* and finally its tight relationship to a *logic for reasoning* about *BSM* programs.

11.1 Vertical modularity

To facilitate implicit interactions between agent's knowledge bases, such as beliefs and a goal base, the mainstream theoretically founded agent programming languages are strongly bound to a single knowledge representation technology employed across the knowledge bases. In all the cases it is a variant of the first-order logic. For instance, consider a language object, a goal formula ϕ, denoting an agent's desire to bring about a certain state described by ϕ in a goal base $\mathcal{G}$. In the case the agent believes that it already reached that state, i.e., the belief base $\mathcal{B}$ entails the goal ϕ, $\mathcal{B} \models \phi$, the goal should be dropped from $\mathcal{G}$. To enable entailment $\models$ of a goal formula from the belief base $\mathcal{B}$, straightforwardly, the easy solution can to implement both the goal base, as well as the belief base with the operator $\models$ as first-order logic entities.

Unlike that family of agent programming languages, the approach taken by the *BSM* framework is rather liberal and provides support for *vertical modularity*, i.e., modularity

w.r.t. the underlying KR technologies. One of the consequences of this liberal approach is however a loss of implicit purposes ascribed to knowledge bases. The syntax of the language is also agnostic of mental attitudes. I.e., the *BSM* framework does not provide explicit first-class concepts such as a *belief* or *goal* and the agent program only encodes relationships between various formulae of agent's knowledge bases. It is agent designer's responsibility to ascribe cognitive purposes, mental attitudes, to the knowledge bases and KR language objects stored in them, as well as to encode their interrelationships in the *BSM* program explicitly. While on one hand moving this burden on agent designer's shoulders, on the other it enables a high degree of flexibility in implementation of various models of agent reasoning, such e.g., *I-System* rationality axioms, described in Section 3.1.

Because of the liberal approach taken by the BSM framework, it could also be argued[1] that because of the lack of first-class constructs for agent's mental attitudes and their interdependencies, the *BSM* framework is too abstract, oversimplified and thus it does not qualify as a proper agent-oriented programming language. In Chapter 3, I argue that actually the lack of a fixed set of language constructs for mental attitudes and their interdependencies allows programmers to implement BDI-style cognitive agents in a more flexible and modular way. To demonstrate that it is indeed the case, later in the dissertation, I describe applications of the *BSM* framework in programming cognitive agents, which make use of the framework's flexibility and modularity.

By exploiting the plug-in architecture of the *BSM* framework, the modular BDI architecture introduced in Chapter 3 provides a template, a *BSM* behavioural template, for development of BDI inspired cognitive agents. It facilitates the use of heterogeneous knowledge representation technologies in agent's knowledge bases and by the same mechanism it also enables a straightforward integration with legacy software, 3rd party software, or external environments. Thus agent's deliberation abilities reside in the KR modules, while its behaviours are encoded as a mental state transformer, a *BSM* agent program.

One of the precursors and sources of inspiration of the modular BDI architecture is *IMPACT*, an agent platform by Subrahmanian et al. (2000). Similarly to the *BSM* framework, *IMPACT* allows encapsulation of interfaces to various 3rd party or legacy systems into logic programming style wrappers, which were later used by *IMPACT* agents. Unlike *BSM*, however, the semantics of *IMPACT* programs was based on logic programming and also parts of agent's knowledge base had to be written as a logic program. Instantiation of *IMPACT* as a *BDI* style agent system with knowledge bases storing and maintaining agent's mental attitudes was thus not as straightforward as for *BSM*. In turn, *IMPACT* does not support a straightforward implementation and adaptation of the agent reasoning model according to the specific application domain requirements. Rather, agent's actions for the next step are selected directly on the

[1] And indeed, it was many times pointed out on various occasions.

ground of its beliefs, information from the external modules.

According to my knowledge, the modular BDI architecture (Novák and Dix, 2006), was probably the first proposal for modularizing the BDI architecture with heterogeneous underlying KR technologies. In the recent work with Dastani et al. (2008d), we formally study translation techniques between heterogeneous KR technologies used in such a modularized BDI architecture. To facilitate integration of heterogeneous KR technologies, the approach taken there aims at preservation of the implicit agent reasoning model of the underlying language. E.g., unlike in the modular BDI architecture, the language interpreter itself drops a goal, whenever its belief counterpart becomes derivable from the belief base. I introduce a technique emulating that mechanism in the *GOAL* language in Chapter 10.

11.2 Horizontal modularity

In its first incarnation, the modular BDI architecture (Novák and Dix, 2006), the *BSM* framework did not provide means for structuring of agent programs, such as nesting of mental state transformers. Similarly to languages, such as *3APL* or *AgentSpeak(L)*, the agent program was encoded as a plain set of context aware production rules. The flat program structure, however, does not scale up well with a growing number of rules. This very pragmatic problem led to introduction of the nested structure of agent programs. The *BSM* framework introduces the construct of a *mental state transformer*, as a means for encapsulating meaningful subprograms. The mst construct is inspired by the notion of a *GOAL* mental state transformer by Hindriks (2001). However, *GOAL* mst's are rather constrained to atomic updates of agent's mental state, while *BSM*'s mst can denote a complex function over agent agent's knowledge bases. In turn, the functional semantics of mental state transformers provides a powerful abstraction for hierarchical decomposition of agent programs.

The nested structure and semantics of mst's was inspired by Gurevich's *Abstract State Machines*[2] (*ASM*) (Börger and Stärk, 2003), which a general purpose model of computation over heterogeneous variable domains. In this sense, *BSM* could be informally seen as a particular instantiation of the *ASM* framework, providing *structured reactive control* over heterogeneous knowledge bases. In result, mental state transformers denote functions, *reactive policies*, over the space of mental states.

The notion of *mental state transformer* as an encapsulated subprogram, a function, provides a basis for modularizing agent programs. In Chapter 5, I describe a concrete implementation of the *BSM* framework as the programming language *Jazzyk*. It integrates a macro preprocessor, which in turn enables writing agent programs in terms of reusable named subprograms with encapsulated functionality. Even though the idea of purely syntactical macros expanded troughout a program is straightforward and rather

[2]Formerly known as *Evolving Algebras* invented by Gurevich (1994).

trivial in itself, in practice, it has important consequences. It enables easy scaling of agent programs written with the *BSM* framework. Furthermore, it also facilitates natural management of large number of program statements, rules, in a manner programmers know from mainstream structured imperative programming languages, such as *Pascal*, *C*, or alike.

Recently, there were several attempts to introduce modules into theoretically founded an agent programming language were made in the field. Probably incomplete list includes the efforts by Dastani et al. (2004) and by van Riemsdijk et al. (2006b) for *3APL*, by Hindriks (2007) for *GOAL*, Dastani et al. (2008e) for *2APL*, or the papers (Madden and Logan, 2009; Hübner et al., 2006b,a) for *AgentSpeak(L)/Jason*. Unlike the straightforward, purely syntactic, technique implemented for the *BSM* framework, these approaches to modularization of an agent language are heavily based on introducing new semantic constructs and in turn adapting the original language semantics. In the case of the BSM framework, this is not necessary. In result, the hierarchical structuring of *BSM* agent programs, together with the powerful abstraction of mental state transformers facilitating functional encapsulation provides a high level of scalability and thus also programming convenience.

11.3 Agent reasoning model

Agent programming languages such as *GOAL*, *3APL*, or *AgentSpeak(L)* provide a fixed implicit agent reasoning model embedded in the semantics of the language. A programmer then focuses only on encoding agent's functional behaviour, while it is the language interpreter, which takes care for interdependencies between agent's mental attitudes behind the scenes. As already mentioned in Section 11.1, the *BSM* framework, and thus in turn also the modular BDI architecture introduced in Chapter 3, does not definitely fix the set of agent's mental attitudes and the axiomatic system of their relationships and interdependencies. As a consequence, it does not implicitly provide mechanisms managing agent's mental attitudes behind the scenes, directly in the semantics of the language. Instead, the programmer is free to define a number and type of individual knowledge bases according to a particular-application domain-specific requirements and subsequently encode such interrelationships among knowledge bases in an explicit manner. While on one hand this requires more effort on the side of the agent designer, on the other the *BSM* framework allows for application-domain-specific adaptation of the agent reasoning model on the language level. As in the case of introducing the modularization support to agent languages, such arbitrary adaptations would yield changes in the programming language semantics and in turn a modified interpreter. An attempt to leverage this issue in *AgentSpeak(L)* was recently made by Winikoff (2005a), resulting in implementation of a meta-interpreter for the programming language. The *BSM* framework supports such adaptations as a first-class concept. In this sense, the *BSM*

framework can be also seen as a *meta language* for specifying and encoding the agent reasoning model tailored for the specific system in development.

11.4 Logic for programming agents

Since its inception, the problem of development of a formal approach to cognitive agent programming is connected with the issue of *logic for reasoning about the resulting programs*. In the past, often it was the logic for agent systems, which was proposed first and only subsequently a suitable programming framework or a language followed. For instance, this was the case for the BDI logic (Rao and Georgeff, 1991) and the *AgentSpeak(L)* (Rao, 1996) language. However, a tight formal relationship between a programming framework and a reasoning framework for agent behaviours is not always established in such a way that it enables practical use of the logic as a formal specification language.

To provide the basis for a formal specification language, as well as to enable reasoning about *BSM* agent programs, in Chapter 4, I introduced *DCTL** logic tailored for the *BSM* framework. Together with the proposal for program annotations, allowing translation of logic-agnostic *BSM* mental state transformers into the temporal logic, it enables extraction of semantics characterization from agent programs and their subsequent verification. Later, in Chapter 6, I used this tight relationship for development of practical design patterns, supporting and simplifying programming with the *BSM* framework. Moreover, I showed that *DCTL** logic allows formalization of the notion of an *achievement goal*, similar to *persistent relativized goal*, one of the central notions of Cohen and Levesque's BDI style logic (1990).

Several logics for agent systems with mental attitudes were proposed. As already mentioned, *AgentSpeak(L)* and in turn the *Jason* framework are inspired by Rao and Georgeff's BDI logic (1991). Additionally, Bordini et al. (2003) provide a formal approach to model checking *AgentSpeak(F)*, a restricted version of *AgentSpeak(L)* w.r.t. *LTL* specifications. *ConGolog* by Levesque et al. (1997) is an agent programming language based on the *situation calculus* (Levesque et al., 1998). In turn, the results for the situation calculus can be used almost directly for reasoning about *ConGolog* programs. de Boer et al. (2007) propose a verification framework for the *GOAL* agent programming language developed first by Hindriks (2001). Similarly, Dennis and Farwer (2008) propose a verification framework for the language *Gwendolen*. van Riemsdijk et al. (2004; 2006a) proposes a version of dynamic logic for reasoning about a restricted version of *3APL* language. Finally, *Concurrent MetateM* described by Fisher (1993; 1994) and Fisher and Hepple (2009) is a language based on the direct execution of temporal formulae. Thus, it directly facilitates both, an *a priori* program specification and verification, as well program execution in the same language.

Because of their bond to an imperative language, pragmatically oriented programming

frameworks, such as *JACK* or *Jadex*, provide only limited support for formal approaches to reasoning about programs written with them. On the other hand exactly because of that relationship, they provide a high level of comfort and familiarity to system developers.

The *BSM* framework tries to strike a balance between the two worlds. It provides a clear semantics, enabling a tight bond with a formal logic for reasoning about its programs and at the same time it strives to provide similar level of flexibility and software engineering convenience as the pragmatic approaches do.

Chapter 12

Part II: Software engineering issues

The second part of this dissertation deals with pragmatic issues of using the framework of *Behavioural State Machines*. It stems mainly from parts of earlier papers (Novák, 2008c) and (Novák and Jamroga, 2009). While the *BSM* framework sets the theoretical scaffolding for a particular way of thinking about design of cognitive agents, the programming language *Jazzyk* provides concrete tools for using it in practice. In this section, I discuss the main contributions of the proposed approach to designing extensible agent-oriented programming languages. I also provide an overview of the broader context and the state of the art in the field.

12.1 The programming language

Jazzyk is an instantiation of the framework of *Behavioural State Machines* in a concrete implemented programming language. While the introduced *Jazzyk* interpreter provides a reference *BSM* implementation, the requirements imposed on admissible *BSM* interpreters in Chapter 2 still leave space for slightly different solutions to particularities, such as the specifics of shared domains, variable substitutions mechanism, or non-deterministic choice. As an example of an admissible language adaptation, Chapter 9 provides an extension of the implementation of the non-deterministic choice mechanism.

Similarly to the other state-of-the-art agent-oriented programming languages, such as the already discussed *Jason/AgentSpeak(L)*, *3APL*, *2APL*, or *GOAL*, *Jazzyk* is an experimental language. While on one side, it is highly desirable to evaluate and use it for larger case-studies and demonstration applications, in the presented work, I focused on enabling experimentation with the core language features and the design style it promotes. Thus, it does not provide a high level of comfort for developers in terms of accompanying tools such as an IDE or a debugger. In the context of the state-of-the-art theoretically founded languages, *Jazzyk*'s syntax is comparatively simple and highly readable, though perhaps a bit "talkative".

The main contributions of *Jazzyk* to the state-of-the-art approach to designing an agent-oriented programming language lie in its two most distinctive features. Firstly, it introduces *hierarchical structuring* of source code of agent programs, and secondly, it

relies on *syntactic macro facilities* for handling modularity and reusability of subprograms. These enable experiments geared towards development of the novel approach to design of *extensible* agent-oriented programming languages.

12.2 Extensible language constructs

It can be argued that *Jazzyk* should not be regarded as a proper agent-oriented programming language. Indeed, it does not come with a set of first-class constructs for programming systems in terms of agent's mental attitudes. Instead, *Jazzyk* provides a modular approach to development of *reactive* systems, while providing a lightweight toolbox (macros) for extending it in a domain- or an application-specific manner.

To illustrate the versatility of the language, in Chapter 6 I gradually developed code patterns, finally implementing an achievement and a maintenance goal. The effort represents the first step towards a more extensive *library* of programming language constructs. I follow the implicit view on intentional stance of promoted by Cohen and Levesque (1990), rather than the one by Bratman (1999) and Rao and Georgeff (1991). I.e., only agent's beliefs and goals are treated as first class objects in the programming framework, while the intentional stance of an agent is a property of an agent program execution towards achieving a goal.

The mainstream approach to design both flavours of agent-oriented programming frameworks, i.e., those theoretically founded as well as those engineering ones overlaying *Java* (cf. Chapter 1), is to first identify and choose a set of agent-oriented features and subsequently implement them in explicit language constructs. One of the negative consequences of such an approach is rigidity of the resulting programming framework. In order to extend such languages with new features, be it towards pragmatic engineering (e.g. attempts to introduce *modules* into agent-oriented programming languages, (van Riemsdijk et al., 2006b; Dastani et al., 2008e; Madden and Logan, 2009; Hindriks, 2007)) or in order to extend the programming with mental attitudes itself (cf. e.g., (van Riemsdijk et al., 2008; Hindriks and van Riemsdijk, 2009; Sardiña et al., 2006; Hindriks et al., 2008)), the language designer must modify the language semantics and subsequently the language interpreter. *Jazzyk* promotes semantic extensions as specific KR module operators and language extensions as code templates or patterns.

As for the formal treatment of design patterns, implementing useful agent-oriented concepts in agent programming languages, only little state-of-the-art work exists. On the one hand, in the family of theoretically founded approaches, Bordini et al. (2007) discuss using code templates for implementation of various goal handling strategies and plan patterns in *Jason*. However, unlike the approach presented in Chapter 6, the use of reusable code patterns is not promoted as a way to extend the programming language itself. On the other hand, engineering approaches, such as *JACK* (Winikoff, 2005b) and *Jadex* by Pokahr et al. (2005), natively allow for exploiting modular *Java*

style of programming and thus also allowing for various customizations by application programmers. This provides a basis for potential use of various design patterns for the underlying language. However, up to my knowledge, customization and extension of these frameworks with formally characterized agent-oriented patterns was not considered yet.

The proposed approach to agent-oriented programming language customization provides a basis for bridging the gap between the programming languages and methodologies and agent modelling frameworks. One of the main results of the analytical stage of single agent systems in methodologies such as *Tropos* (Giorgini et al., 2004) or *MaSE* (DeLoach, 2004), are agent's goals, or tasks, associated with agent's roles. Such methodologies are not coupled to a particular agent architecture and the details of the agent design are left to a particular platform. The presented notion of *commitment oriented programming* paves the way to a hierarchical decomposition of agent system specification in terms of various types of commitments, such as e.g., goals, subgoals, plans, tasks, behaviours, etc. In turn, *BSM* code patterns allow system designers to develop custom sets of high level language constructs fitting the notions of the considered agent-oriented software engineering methodology. In order to explore the connection between the two worlds, one of the avenues of future research would be gradual development of an extensive library of design patterns for handling aspects of various agent-oriented notions, such as obligations, protocols, or roles, etc.

While I do my best to show the usefulness of the presented approach to designing extensible agent-oriented languages, the proposal still needs to be thoroughly explored and validated in many various-scale experiments and case-studies, beyond those presented in Chapter 7.

Chapter 13

Part III: Evaluation, extensions and beyond

The final part of this dissertation discusses experimental work and evaluation of the previously introduced theoretical and engineering results. It is mainly based on the results published in (Hindriks and Novák, 2008; Novák and Köster, 2008; Köster et al., 2009; Novák, 2009b). In the following, I discuss mainly related work, as well as some future outlooks for further evaluation and directions of development of the technologies, based on the framework of *Behavioural State Machines*.

13.1 Experimental work

The three case-studies, described in this paper, served me most importantly as a vehicle to nurture and pragmatically drive the theoretical research towards a methodology for using the *BSM* framework, and in turn the agent-oriented programming language *Jazzyk*. As an important side effect, I also collected a body of experiences with programming BDI-inspired virtual cognitive agents for computer games and simulated environments.

Since in the long run, the background motivation was the development of autonomous robots, in both cases of *Jazzbot* and *Urbibot*, the virtual agents had to be running autonomously and independently from the simulator of the environment. This choice had a strong impact on the design of the agents w.r.t. the action execution model and the model of perception. In both described applications, the agents are remotely connecting to a simulated environment, in which they execute actions in an asynchronous manner. I.e., they can only indirectly observe the effects (success/failure) of their actions through later perceptions.

As far as the model of perception is concerned, unlike other game bots, *Jazzbot* is a pure client side bot. The amount of information it can perceive is a strict subset of the information provided to the game client used by human players. Hence, the *Jazzbot* agent cannot take advantage of additional information, such as the global topology of the environment or information about objects in distant parts of the environment, which are accessible to the majority of other bots, available for first-person shooter games. In the case of *Urbibot*, the simulator provides only perceptions accessible to the embodiments of robot's sensors. In particular, these are most importantly a camera, a directional distance sensor and global positioning, hence the available information is, similarly to

Jazzbot, only local, incomplete and noisy. To some extent, these constraints apply also to the third case-study in development, the *AgentContest team*. However, in this case, the focus is more on the multi-agent coordination, rather than on development of individual single agent systems.

Because the implemented agents are running independently from the simulation engine and execute their actions in an asynchronous manner, their efficiency is only loosely coupled to the simulation platform speed. In our experiments, the speed of agent's reactions was reasonable w.r.t. task the bots were supposed to execute. However, especially in the case of *Jazzbot*, due to deficiencies on the side of sensors, such as a missing camera rendering the complete scene the bot can see, *Jazzbot* in its present incarnation cannot match the reaction speed of advanced human players in a peer-to-peer match.

The *BSM* framework assumes that the mental state of an agent, including its environment, changes only between the single executions of the deliberation cycle. Therefore, in order to implement agile agents, which act in their environments reasonably quickly w.r.t. the speed of change of the environment, the query and update operators should be computable procedure invocations and shouldn't take too long.

Since, the agents store their internal state in application-domain-specific KR modules, the control model of the *BSM* framework results in agents which can instantly change the focus of their attention w.r.t. the observed changes of the state of the environment. The goal-orientedness of agent's purposeful behaviours emerges from the coupling between behaviour triggers and agent's attitudes modeled in its components. This turned out to be of a particular advantage when a quick reaction to interruptions, such as an encounter with an enemy agent or a patrol, was needed. On the other hand, because of the open plug-in architecture of the *BSM* framework, we were able to quickly prototype and experiment with various approaches to knowledge representation and reasoning, as well as various models of interaction with the environment.

The presented research follows the spirit of Laird and van Lent's argument (2001), that approaches for programming intelligent agents should be tested in realistic and sophisticated environments of modern computer games. The *Jazzbot* project thus walks in footsteps of their *SOAR QuakeBot* (Laird, 2001).

Another relevant project, *Gamebots* by Adobbati et al. (2001), provides a general purpose interface to a first-person shooter game *Unreal Tournament* produced by Epic Games, Inc. (2004). *Gamebots*' approach is however server side. Its virtual agents are provided with much more information than a human player has, what was not in the spirit of our aim to emulate mobile robots in virtual environments. We did not pick the *Gamebots* framework for our project because it is specific to the commercially available game *Unreal Tournament* and since 2002, the *Gamebots* project does not seem to be actively maintained.

Recently, an interesting new interface to various games, such as *Unreal Tournament* game engine, called *Pogamut 3*, was developed by Gemrot et al. (2009). It provides a flexible and rather abstract interface to game simulations. Even though the interface it

opens to third party developers is again server-side, because of its flexibility, it might be interesting to explore possibilities of using it in future experiments with *Jazzbot*-like agent systems. An interesting avenue of possible future work would be using the rich 3D game simulation engine for experimentation with *serious games*, i.e., specialist training simulations. To this end, multi-agent scenarios would need to exploit the powers of an open communication infrastructure, similar to that I describe later in Chapter 14. A new specialized *Pogamut* plug-in for *Jazzyk* interpreter would have to be developed as well.

Traditionally, agent-oriented programming languages, such as *GOAL* or *3APL*, use mainly Prolog for representing beliefs of cognitive agents. To enable easier comparison and porting of applications from such languages, I plan to develop *Jazzyk* plug-ins, KR modules, interfacing *Prolog* interpreter, (e.g., *SWI Prolog* engine (Wielemaker, 2003)), as well as one for a *LISP* dialect *Scheme* (Sussman and Steele Jr., 1998).

Our work on the *Jazzbot* and *Urbibot* case-studies is novel in the sense that it seems to be the first efficient application of non-monotonic reasoning framework of *ASP* in a highly dynamic domain of simulated robotics or a first-person shooter computer game. To my knowledge the first attempt by Padovani and Provetti (2004) uses *Answer Set programming* for planning and action selection in the context of the *Quake 3 Arena* game (id Software, Inc., 1999). However, authors note that their bot could not recalculate its plans rapidly enough, since each stable model re-computation required up to 7 seconds in a standard setup. Thus, in comparison to *Jazzbot* or *Urbibot*, their agent was capable to react to events occurring in the environment only to a lesser extent. This was probably due to the fact, that both the action selection, as well as planning was completely written in *ASP*. *Jazzbot* and *Urbibot* use logic programming only for reasoning about static aspects of the world and goals. The action selection is left to *Jazzyk*, a language designed for that task.

Similarly, to my knowledge, the transparent integration of various KR technologies such as the logic programming for representing agent's beliefs and goals and the object-oriented language for storing topological information, together with a generic reactive control model of the agent program in *Jazzyk* is rather unique. In consequence, the *BSM* framework shows a lot of potential for further experimentation towards exploiting synergies of various AI technologies in cognitive agent systems. Especially the attractive domain of virtual agents and autonomous non-player characters for computer games is an interesting direction for future research. Yet, more experimentation is needed to explore the limits and deficiencies of the proposed approach in such a domain.

13.2 Probabilistic approaches to agent-oriented development

The underlying semantic model of the *Behavioural State Machines* framework is a labelled transition system. In consequence, the underlying semantic model of the *P-BSM*

framework is a discrete probabilistic labelled transition system, i.e., a structure similar to a *Markov chain* (Markov, 1906). This similarity suggest a relationship of the *P-BSM* underlying semantic structure to various types of *Markov models* (cf. e.g., (Meyn and Tweedie, 1993)). However, here I do not further explore this relationship. As of now, the *P-BSM* framework should be seen as a pragmatic minor, yet quite a powerful, extension of the practically tested *BSM* framework.

In the field of agent-oriented programming languages, recently Hindriks et al. (2008) introduced an extension of the *GOAL* language, where a quantitative numeric value is associated with execution of an action leading from a mental state m to another mental state m'. I.e., a triple of a precondition ϕ (partially describing m), an action a and a post-condition ψ (describing m') is labelled with a utility value $U(\phi, a, \psi)$. Subsequently, in each deliberation cycle, the interpreter selects the action with the highest expected future utility w.r.t. agent's goals.

The approach of Hindriks et al. focuses on estimating aggregate utility values of bounded future evolutions of the agent system. I.e., it evaluates possible future courses of evolution, plans, the agent can consider, and subsequently chooses an action advancing the system evolution along the best path. The *P-BSM*, on the other hand, is concerned only with selection of the next action, resulting from the bottom-up propagation of probabilistic choices through the nested structure, a decision tree, of the agent program. While the approach of Hindriks et al. can be seen as a step towards look-ahead like reactive planning, *P-BSM* remains a purely reactive approach to programming cognitive agents. Informally, except for the nested structuring of agent programs, the probabilistic choice of the *P-BSM* framework could be emulated by the approach of Hindriks et al. with the look-ahead planning bound of the length one. Even though the underlying philosophy of the two approaches differs, they both share the view that

> *while underspecification of agent programs is in general inevitable, in situations when a suboptimal performance is tolerable, providing the agent program interpreter with a heuristics for steering its choices can lead to rapid development of more efficient and robust agent systems.*

Note that the core idea behind the *P-BSM* extension, i.e., specification of the probability of choosing a branch of a non-deterministic computation by a label, is straightforwardly applicable to other agent programming languages, especially those rule-based.

One possible avenue for a future work on using the probabilistic extension of the original *BSM* framework and its further development is the domain of *behaviour learning*. I.e., instead of hard-encoding the execution frequencies high level behaviours composed of basic capabilities, these can be learned by a mass evaluation in simulated scenarios and adjusted by a hill climbing type of algorithm[1]. In this context, because of its flexible hierarchical structure, the *P-BSM* framework allows even for recombinations of

[1] The idea stems from a discussion with Lin Padgham at *AAMAS 2009* conference in Budapest, Hungary.

the lower level behaviours in the course of learning, which could lead to a kind of *genetic programming* approach to building cognitive agent systems.

13.3 Comparison with related frameworks

The task to compare various agent-oriented software engineering approaches and their associated tools and technologies is of great importance for the field theorists. Programming frameworks, platforms and languages are, however, engineering tools in the first place. Thus, a sound study of relations of such languages necessarily involves not only comparison of their expressive powers, but also evaluation of their respective forms and quality of their implementations. It not of much value to modern programmers to know that programming in a low level language such as e.g., *assembly language*, is the most expressive form of writing code for a CPU, when the engineering processes connected with usage of that language leads to prolonged software development, difficult maintenance and low portability of the code, to name just few desirable properties imposed on a programming language. A programming language has also to come with high level constructs supporting abstractions appropriate for the programming paradigm, as well as the application domain the language should support. It is easy to see that such a rigorous and objective comparison of different languages becomes inherently biased by subjective points of view of the evaluators. Moreover, the quality of implementation of the tools associated with the approach influences the results of the comparison as well.

13.3.1 Theoretically founded agent programming languages

Even though perhaps problematic, studies of relationships between programming languages purely geared towards their respective expressivity still make a lot of sense. They help us to gain a deeper insight into the question about which problems actually lie in the core of the task to build intelligent agents of various types. Examples of such studies are the bi-simulation results shown by Hindriks (2001) for the couples *3APL* and *AgentSpeak(L)* and *3APL* and *ConGolog* (Giacomo et al., 2000).

While on one side, Chapter 7 presents a practical approach to evaluate abilities and merit of the introduced programming framework, in Chapter 10 I also made an attempt to theoretically study the relationship with a state-of-the-art agent-oriented programming language *GOAL*. The result shows that the *BSM* framework, and in turn also *Jazzyk*, has at least the expressive power of *GOAL*. In Theorem 10.10, I show that there is an efficient translation for every *GOAL* agent into a *BSM* agent program. This result is actually not very surprising, because the *BSM* framework is a rather generic programming system. Yet, the difficulties, which arose in the course of proving the result, are symptomatic and indicate the differences, or perhaps even limitations, of the abilities of the BSM framework w.r.t. the family of logic-based agent-oriented programming languages, such as e.g., *Jason* or *3APL*. Note, the $\mathfrak{C}_{\mathbf{bcs}}(\mathcal{P}_{\mathcal{A}})$ component of the translation

(cf. Subsection 10.2.2). In order to clean the goal base from the goals which became satisfied by taking an action, a belief base update, from the $\mathfrak{C}(\Pi)$ component, it is necessary to enumerate all the goals which the agent could possibly hold at any timepoint of its lifecycle.

Most of the state-of-the-art agent programming languages involve similar mechanisms, either associated with their goal bases, or the base of intentions. In the case this set of goals is infinite, it wouldn't be easy, if possible at all, to express the goal handling mechanism by means provided by the *BSM* framework. This is mainly due to the fact that firstly, the *BSM* framework's architecture allows only for execution of a finite number of atomic updates per deliberation cycle, and secondly, each atomic update is yielded by a concrete primitive update formula in the program. Thus it is not easily possible to express statements implying unbounded, or even infinite internal atomic updates or checks. In practice however, agent programs written with such languages are finite and often the semantic rules requiring that the elements of agent's knowledge bases should be dropped *immediately* upon certain change are not of critical importance to application developers. Often, it is not critically required that whenever a goal is satisfied it must be immediately dropped, but rather it should be dropped sometime in the near future, similarly to goal handling defined by the ACHIEVE design pattern in Definition 6.10. Thus, the aforementioned limitation of the *BSM* framework can be relatively easily overcame by using techniques such as those presented in the compilation of *GOAL* programs into *Jazzyk*, or by designing appropriate code patterns.

13.3.2 Common semantic basis for agent programming languages

The implementation strategy used to identify specific semantic features of the *GOAL* language and to emulate these explicitly in *Jazzyk* also raises the question, whether features of other agent programming languages can be compiled to *Jazzyk* in a similar way. Although I do not extensively argue for this, I believe that a similar approach can also be applied to other rule-based agent programming languages. In particular, the following translation methodology could be applied to compile agent programs into *BSM*s:

1. compile the underlying knowledge base(s) into equivalent *BSM* KR module(s),

2. compile the (action, planning, etc.) rules of the agent program into *BSM* mental state transformers using the operators of the KR module(s), and finally

3. implement any specific semantic features of the language by an appropriate compound *BSM mst* and "append" it to the one constructed in the previous step.

Since the *BSM* framework also features a much simpler conceptual scheme than higher level agent languages, it provides a promising basis for an intermediate language into

which agent programs can be compiled and interpreted. In that sense, the *BSM* framework can be seen as a meta-language, which enables tailoring of the architecture and the agent deliberation cycle to emulate even variations of other languages. Thus the *BSM* framework can be seen as a minimalistic tool for rapid prototyping of new agent-oriented features in programming languages based on the reactive planning paradigm. Thus, one of the possible avenues for a future work could be further research towards compilation of other languages into *Jazzyk* and implementation of respective efficient translators. If the approach would prove successful, *Jazzyk* could be seen as a kind of a low level assembly language for agent-oriented programming.

Relevant in this context, Winikoff (2005a) introduces a very simple meta-interpreter for a variant of the *AgentSpeak* language. The meta-interpreter makes the *AgentSpeak* deliberation cycle explicit, so that whenever a new feature extending the original language is proposed, its implementation can be rapidly prototyped and tested using the meta-interpreter. The technique used for showing correctness of the meta-interpreter very much resembles the one introduced in Chapter 10 for showing the relationship between *GOAL* and *Jazzyk*.

To tackle the problem of establishing a common semantic basis for various BDI-style agent-oriented programming languages, an interesting alternative framework was recently presented by Dennis et al. (2007). To my knowledge, this is the only proposal to date in this direction. The resulting solution, is based on the idea of constructing a specialized rule-based language, incorporating each and every semantic feature of a variety of available high-level agent languages. Thus, their approach does not provide a similar minimalistic implementation strategy, as the one promoted and illustrated in Chapter 10. The solution instance presented in this dissertation is based on the idea to provide a concise set of simple high-level concepts, a common core. Furthermore, by making the agent's deliberation cycle explicit in the language, this in turn enables compilation of a variety of agent programs into the core instruction set. This strategy is explicitly aimed at *reducing* a set of high-level agent programming constructs to a *simpler*, more basic set of concepts.

13.4 Broader context

To touch on the broader context the work presented in this thesis fits into, below, I provide a brief and rather shallow overview of related work, which also seems to be relevant to the framework of *Behavioural State Machines*. I do not give any deeper insights into the suspected relationships. Such studies provide another set of possible directions for future research towards establishing the expressivity and limits of the *BSM* framework.

Except for the vertical modularity (cf. Section 11.1), the core feature of the *BSM* framework is the hierarchical structuring of agent programs. In a way, *Jazzyk*'s top-

down process of searching for an applicable primitive update, yielded by the program (cf. Chapter 5), resembles application of decision trees (e.g., (Russell and Norvig, 2002, Chapter 18)). A decision tree reaches its decision by performing a sequence of tests, structured in a hierarchical manner. Internal nodes correspond to tests and leaves of the tree yield primitive actions, in this case *BSM* updates. Decision tress can be seen as vehicles for learning processes and can be constructed as a result of a learning process. This resemblance again reinforces the speculation on feasibility of behaviour learning, I previously discussed mentioned above in Section 13.2.

It also seems that there is a resemblance of the *BSM* framework to the idea behind *universal plans*, by Schoppers (1987). A universal plan is a compact hierarchical, decision-tree style structure, representing every possible course of action an agent can take. Similarly to the *BSM* framework, actions of an agent are chosen by a iterated top-down decision process, which traverses the universal plan. Its nodes correspond to sensory conditions and leaves to atomic actions. Is is noteworthy to mention that later it was found that *universal plans* amount to *policies* similar to those employed in *Markov Decision Processes* (Markov, 1906) (also e.g., (Meyn and Tweedie, 1993)). Ginsberg (1989) argues that the universal plan synthesis is way too complex.[2] In the *BSM* framework, I focus, however, on a human centered approach to writing agent programs. Moreover, the domains I consider in the case-studies, described in Chapter 7, are also reasonably small. To establish a deeper insight into the relationship between universal planning and the ideas behind the *BSM* framework remains a future research task.

Together with Dix (2007)[3] I show that for a variant of a nested hierarchical structure of interactions rules, not including the sequence operator, *BSM*-style agent programs can be *unfolded* into a set of plain condition-action rules of the form

$$\phi \longrightarrow \oslash\psi$$

ϕ is a possibly compound *BSM* query and $\oslash\psi$ is a single primitive update mst. A set of such rules defines a *policy* over the mental state space.

Supporting the intuition of a close relationship between universal plans and the *BSM* framework. This observation also correlates with the fact that the core of inspiration for the *BSM* framework were *reactive planning* approaches, such as *AgentSpeak(L)* by Rao (1996) or the original *Procedural Reasoning System* (*PRS*) by Georgeff and Lansky (1987).

[2] For a more thorough discussion on the controversy around universal plans, consult the bibliographical and historical notes in (Russell and Norvig, 2002, Chapter 12).

[3] The paper is not discussed deeply in this thesis. I perceive the main results therein as a kind of a dead-end avenue towards the final form of the *BSM* framework.

13.5 Application domains

To conclude the discussion of the dissertation results, I finally elaborate on application domains, in which I see a potential for exploiting the strengths of the *BSM* framework and the *Jazzyk* language. Yet, the real capabilities of the *BSM* framework in these, is to be established by further case-studies and experiments.

Chapter 7 already sketched some application domains in which we tried to evaluate the *BSM* framework's capabilities. One of the prominent problems in computer gaming industry and digital entertainment is creation of smart non-player characters. I.e., virtual characters, avatars and entities, which behave in a a non-trivial, intelligent manner. Nowadays, this task is approached by programming the reactive behaviours in the form of standard finite state machines (FSM). Encoding behaviours as FSMs is however rather limited, as the number of states the agent can be in must be fixed and the transitions must be hard-encoded in the design time. Because of these limitations, the resulting agents are rather predictable and in turn not believable enough.

The *BSM* framework offers an interesting alternative. The number of states, a *BSM* agent can be in, is possibly infinite and determined solely by the product of states its knowledge bases can be in. Queries can be seen as specifications of state space partitions. Moreover, the transition specifications are not definite as the state resulting from a transition is determined by the knowledge base semantics itself upon an update by an update formula. Finally, agent program and subprograms are structured in hierarchical and reusable manner what speeds up the development process and allows for rapid prototyping.

As the gaming industry is moving towards games with large number of characters, one of the challenges in the digital entertainment domain is *scalability*. It is yet to be evaluated how efficient execution of agents using *Jazzyk* interpreter actually is.

The small footprint of the *Jazzyk* interpreter and its relative versatility make it suitable for applications in non-critical embedded domains, where rich sensing and sophisticated action selection is necessary. In this context, an interesting application domain is entertainment mobile robotics. The *Urbibot* study (cf. Chapter 7) was already a step in this direction. In fact, *Jazzyk* interpreter was developed with portability to embedded platforms in mind. The module *JzUrbi* (cf. Chapter 8) provides an interface to a highly portable low-level language for robot's hardware control. Similarly, applications of the framework in the domain of ambient intelligent, such as e.g., intelligent building control, seems to be a promissing potential application domain.

Chapter 14

Towards open multi-agent systems

14.1 Epilogue

It was a long day for Bronja. Finally she arrived back home. She opens the door of her penthouse apartment, opens curtains to enjoy the beautiful view over the evening city for half a minute. She kicks her shoes off, immerses into a big, soft leather armchair and switches on the TV. The BBNN news network shows footages from some rescue operation. Bronja does not want to see those things. Not tonight. She's tired. Then she sees it! The Bratislava airport terminal building covered in smoke!

Twenty minutes after noon, an earthquake struck the area between the cities of Bratislava, Vienna and Budapest. The hardest hit took Bratislava's city airport. The new stylish airport terminal building partly collapsed and is still burning. Airport's underground kerosene tanks were also partly damaged and the situation threatens to escalate. Noon is a busy time at Bratislava airport and many travellers and airport personel members are buried or trapped alive in parts of the collapsed building.

In the following minutes and hours, the police, army and firefighter brigades are busy. The relief operation runs full speed. A team of army reconnaissance mobile robots was deployed at the site. These are however not primarily designed for rescue operations and lack functionality, such as the ability to remove heavy debris. Vienna firefighters quickly come to help and provide two of their ÖHH-78 models, autonomous heavy-duty tracked robots featuring a forklift device. The police also deployed a small fleet of Skylar UAVs, autonomous mini-helicopter drones, to support the operation and provide the overall picture of the disaster site.

The group of robots quickly coordinates, forms two teams and already after two hours of autonomous operation they are able to help to extract a group of elderly tourists from the terminal building. They were trapped in a staircase. The earthquake struck as they were moving from the gate to the bus, supposed to take them to the aircraft to Bucharest.

Bronja watches the news with trembling in her legs. In the morning, it

could be her! She made it to the departure gate at the airport right before the BBWings staff wanted to close it. Fortunately, thanks to the airport robotic assistant Ape, they knew she comes and were waiting for her. Hadn't she make it, it could be her whom the robot rescuers would have to save.

14.2 Outlook

This dissertation presented theoretical, as well as technological results stemming from the proposal of the framework of *Behavioural State Machines* in Chapter 2. The *BSM* framework is my proposed solution to the problem of programming single agent systems. In its focus stands the problem of programming embodied reactive deliberating agents. I.e., such which base their decisions on reasoning about their internal model of the world and at the same time are able to quickly react to changes of the context, but not forget about the goals they were pursuing before the interruption.

A natural next step for the future research would be the one towards development of multi-agent systems exploiting such technologies. Here, a completely new realm of issues opens, most prominently those of inter-agent communication and coordination.

Visions, such as that of multi-robot teams, promoted in the introductory example of this chapter, require integrative technologies for development of, possibly large, *open multi-agent systems* (MAS). I.e., those, comprising heterogeneous cooperative agents. In such scenarios, a number of agents interacts to fulfill complex tasks without human intervention or seamlessly support human everyday activities. The agents can be embedded in consumer electronic devices, small robots, or desktop computers, etc. Thus, possible application areas for such a technology are numerous, from multi-robotics, to ubiquitous computing, or ambient intelligence, to name just a few.

One of the main aspects of open systems is *inter-agent coordination*, i.e., *communication*. One of the background motifs behind the liberal attitude taken by the *BSM* framework is the recognition that *different application domains require different knowledge representation technologies*. In a similar liberal fashion, it is relatively straightforward to also recognize that

> *different multi-agent applications, require different different communication and coordination technologies.*

By the few constraints the *BSM* framework imposes on the internal architecture of agent systems, it promotes development of heterogeneous agents exploiting different underlying technologies. In order to enable such agents to cooperate, there arises a need for a communication platform supporting development of *heterogeneous and open multi-agent systems*. Moreover, to enable communication in such MASs comprising heterogeneous and independent agents, the middleware platform should impose as few requirements on the architecture and implementation of the individual agents as possible.

To conclude this dissertation, as well as to pave the way towards application of open frameworks, such as *BSM*, I conclude the discussion in this final part of the dissertation by a proposal for a *lightweight communication platform for open multi-agent systems.* It aims at supporting development of open heterogeneous multi-agent systems, such as multi-robot systems or applications of ambient intelligence, discussed above. I also argue, why the state-of-the-art *FIPA* compliant agent platforms are not directly suitable for this task and suggest a set of features of a suitable communication platform.

14.3 Lightweight open communication platform

14.3.1 State of the art

The state-of-the-art MAS platforms[1] either follow widely accepted standards for interoperability of agents, such as *FIPA* (FIPA, 2000–2009c) or *OMG MASIF* (Object Management Group, 2000), or use a proprietary, rather non interoperable approach. *JADE* platform (Bellifemine et al., 2005; Telecom Italia Lab and The JADE Board, 2009) is an epitome of a *FIPA* compliant agent platform, the currently prevailing MAS interoperability standard. While the group of the proprietary platforms is rather large and their main focal points vary, the noteworthy ones with a broad range of application domains include the already discussed *OAA* by SRI International (2007), *Cougaar* by Helsinger et al. (2004) and *RETSINA* by Sycara et al. (2003).

While providing a high degree of interoperability w.r.t. inter-agent communication as well as platform distribution, the *FIPA* complying platforms are not suitable for development of open heterogeneous multi-agent systems. A fully *FIPA* compliant platform implementing the mandatory *AMS service* requires agents to *physically reside on the platform* and it overtakes management of the full agent lifecycle of agents running in it ((FIPA, 2000–2009b), par. 4.2.2). Therefore, regardless of their specific features, all the state-of-the-art *FIPA* compliant agent platforms commit to a single agent programming language, which is almost exclusively *Java*. The standard does not specify any external API for connection of agents, not physically residing on the platform. In turn, these platforms rather promote MASs, which are homogeneous w.r.t. the implementation programming language used.

The benefits of using *FIPA*-compliant platforms is however the strong support for inter-agent communication. Agents in these platforms communicate using *FIPA Agent Communication Language* (*FIPA ACL*), which became a *de facto* standard for agent communication languages. Thus, use of *FIPA ACL* enables access to a plethora of specialized domain specific agent services and gateways implemented for the FIPA interoperable platforms, such as *JADE*.

[1] Although a plethora of MAS platforms was developed, most of them are not maintained any more. I focus here only on those freely available, still maintained platforms.

On the other hand, except for *Cougaar*, most of the proprietary platforms like e.g., *OAA* or *RETSINA* (specifically *RETSINA* Communicator (Shehory and Sycara, 2000)) are not so tightly bound to a specific agent programming language. The downside is however that they do not provide a sufficient interoperability support, similar to that of the platforms following a standard. So even though it is relatively easy for agents running outside the platform to connect and register with it, they can only communicate with agents and services residing on the same platform. Often such platforms natively support *KQML/KIF* agent communication languages. In turn, multi-agent systems developed with such platforms are rather closed w.r.t. the outside world.

14.3.2 The platform

Availability of energy-efficient and affordable small form factor computers opens possibilities for application of the multi-agent metaphor to application domains like ubiquitous computing, ambient intelligence or multi-robot systems. A specialized middleware is necessary to support development of a wide range of embodied open heterogeneous MASs, such as networks of agents controlling household appliances and devices for smart homes or teams of robots.

In particular, it is important on one side to *ensure a wide interoperability of multi-agent systems* using such a middleware solution, while at the same time *provide decoupling of agents from the platform, as well as from a specific agent programming language.* In the following, I propose and argue for a set of features, such an agent communication platform should support.

Agents should be platform independent entities taking care of their own execution and lifecycle.

Interoperability issues the platform has to address and APIs it should provide are twofold:

inter-agent: Agents should be able to communicate in a standard communication language, such as *FIPA ACL*, supporting a variety of message transport protocols, applicable according to a particular application domain.

inter-platform: Agents associated with the platform should not only be able to communicate with agents registered with the same platform, but also with agents and services on other standard (*FIPA*) compliant platforms as well.

Platform services should stem from a widely accepted standard, such as *FIPA*, so that the platform provides the necessary services to the agents, but at the same time does not constrain their autonomy. The essential services should therefore include

directory service with which agents can register their coordinates, properties and capabilities and which should provide a look-up facility, and

message transport service enabling the inter-agent communication on the same, or on other standard-complying platforms or gateways. It should also provide a translation between various message transport protocols.

Technical implementation has to support agent vs. platform decoupling. I.e., it should result in a *lightweight and modular middleware solution* supporting

portability and scalability: The middleware should enable deployment on various hardware and software platforms, with a focus on a wide range of computers from small form factor computers, such as e.g., *Gumstix* Gumstix Inc. (2009), to server solutions,

lightweight APIs: The interfaces, the platform provides, should be *agent programming language agnostic*, i.e., the platform designers should impose as few restrictions on associated agent implementation technology as possible (e.g., provide a plain TCP socket interface), and

robustness: In a heterogeneous open multi-agent systems a platform cannot rely on correctness of behaviours agents associated with it perform. Hence it has to be able to deal even with malicious behaviours in a robust manner.

14.3.3 Summary and related work

The proposal for the lightweight communication platform is based on a position paper, I published as a technical report (Novák, 2008b). The proposed communication infrastructure for *open heterogeneous multi-agent systems* stems from the *FIPA Abstract Architecture* (FIPA, 2000–2009a). However, unlike the *FIPA* platform, not imposing strong constraints on agents it manages. The most prominent issues it relaxes are the strong coupling with a specific agent programming language and the management of agents execution and lifecycle by the platform. In the case of *FIPA*, the nowadays prevailing standard, these constraints are a result of the *FIPA Abstract Architecture* reification, in the form of the *Agent Management Specification* (FIPA, 2000–2009b), in particular *Agent Management Services* (Subsection 4.2 therein) and *Agent Platform* (Section 5 therein) specifications.

Except for the use of a standard ACL, perhaps the closest relative of the proposed agent communication platform is the *CoABS Grid* (Kahn and Cicalese, 2002; Global InfoTek, Inc., 2009) infrastructure, which, however, is not in freely available under open source licensing terms.

The proposed experimental lightweight communication platform is a result of considerations and experiences made in preparatory and development phases of the *AgentContest team* project (cf. Chapter 7). Because we did not make any steps towards implementing the platform in the course of my work towards this dissertation, as a replacement in the *AgentContest team* project we used the above mentioned *SRI*'s *Open Agent Architecture* instead.

Chapter 15

Conclusion

The recurring theme of this thesis is that of marrying deliberation and reactivity in embodied cognitive agent systems. If something is to be taken from this dissertation, than perhaps the motto of the second part of this thesis, which distills its main contribution and the essence of the research leading to it:

How to build cognitive agents by integration of various existing AI technologies?

How to encode action selection mechanism in a concise and elaboration tolerant manner?

And once we have a framework tackling the previous two issues, how should programmers use it?

Let reactivity rule over deliberation!

Behavioural State Machines is a framework for programming reactive programs. Rules of such agent programs take the form *query* $\longrightarrow$ *update*, where each formula, be it a query or an update, takes the form of a complex operation on an agent's knowledge base. This way, it is possible to express functionality of an agent system in terms of *reactive* rules and at the same time facilitate its complex *deliberation*. As a consequence of this setting, it becomes relatively easy to exploit synergies of *heterogeneous knowledge representation approaches* and technologies embedded within agent's knowledge bases.

A programming framework or a language is an engineering tool in the first place. Therefore, I invested a lot of effort to make the programming language also practical and accessible to system developers, its users. I tackled this challenge along two orthogonal vectors: 1) *the language versatility* and 2) *the methodology of the framework*.

To provide programmers with constructs facilitating code modularity, reuse and elaboration tolerance, the language provides hierarchical structuring and integrates a powerful macro preprocessor. In order to demonstrate extensibility of the framework and thus provide clues on how to use it in an efficient and sophisticated manner, I introduce the notion of design patterns implementing agent's commitments towards its mental attitudes. As a result, I arrive to a novel way of programming agent systems relying on such patterns, *commitment oriented programming*.

The field of programming agent and multi-agent systems lies on the edge between theoretical research and engineering. While for the theoretical part it's possible to lay down proofs or rigorous refutations of results, the possibilities for evaluating statements

about practical usefulness of a technology are rather limited. To stand up to the motto of the thesis preface, I do my best to *demonstrate* that the proposed technologies also work in reality. I.e., *there is a way to apply them successfully.* It would certainly be desirable to make a stronger, rather universally quantified, statement about the presented contributions. However, only evaluation on a large number of applications, experiments and case-studies, at best drawn from real world problems, can provide evidence about the real usefullness of the proposal for a broader range of uses and users. I feel that I did my best to make my case.

15.1 Acknowledgements

As I mention in the Preface to the dissertation text, a great deal of the presented results stem from joint works with my colleagues and collaborators.

Koen Hindriks contributed to the elegant form the *yields* calculus in Chapter 2 takes. His influence can be also seen throughout Chapter 10, which is a result of our joint work during my stay in Delft and his later visit in Clausthal. We presented that work at MATES 2008. I am grateful to Koen for never missing an opportunity to challenge me with *constructive* critique and stimulating discussions on all sorts of topics of research, life and fun.

Wojtek Jamroga helped me to clarify my views on the formal background of the *BSM* framework. After all, the elegant notation for the *BSM* operational semantics was invented by Wojtek for our joint paper presented at AAMAS 2009. Chapters 4 and 6 present main results of that joint work. After all, technically, the *DCTL** logic, the crucial enabler of the semantic characterizations for the code patterns, was developed by Wojtek, after his critique of my first, erroneous, attempt presented in Dagstuhl seminar in 2008.

My supervisor Jürgen Dix co-authored the original paper on the modular BDI architecture, published at AAMAS 2006, and helped me to polish my ideas there. Chapter 3 heavily relies on that joint work. Jürgen also helped me to polish my later paper for ProMAS 2007 workshop and we published together with Mehdi Dastani and later Tristan Behrens all the reports on the Multi-Agent Programming Contest, presented in CLIMA IV, CLIMA VII, ProMAS 2007 and ProMAS 2008 and the KI Journal in 2009.

I must acknowledge the great help by diploma students of the *Computational Intelligence Group* at *Clausthal University of Technology.* Bernd Fuhrmann, Michael Köster, David Mainzer, Slawomir Dereń and Weiyu Yi did under my supervision almost all the implementation work towards the three case-studies presented in Chapters 7 and 8.

Bernd implemented *Jazyk*,[1] the first prototype of the interpreter. Even though I later completely rewrote it, Bernd's work allowed me to explore and clarify various technical

[1] "Jazyk" with capital J means "The Language" in Slovak. The double "zz" in *Jazzyk* reflects the second major re-implementation of the original interpreter.

issues. That enabled later rapid development of the the current *Jazzyk* interpreter incarnation. Furthermore, the *JzUrbi* and *JzRuby* plug-ins, as well as the *Urbibot* agent based on them were developed as a part of Bernd's student projects and his diploma thesis.

Michael Köster developed the *JzASP* module, which together with the *JzNexuiz* written by David Mainzer and *JzRuby* by Bernd Fuhrmann enabled the *Jazzbot* demo application. While I made the overall design, the actual implementation of *Jazzbot*, presented in our joint paper for the CogRob 2008 workshop, was also developed by Michael.

To enable the step towards multi-agent systems in *AgentContest team* Slawomir implemented the *JzOAAComm* connector module and Weiyu wrote the *JzMASSim* module. At the time of writing this dissertation, the development of *AgentContest team* by Slawek is about to be finished and will be published as a part of his diploma thesis.

The joint work by Bernd, Michael, David and me, on the *Jazzbot* and *Urbibot* case-studies, resulted in a joint paper, we presented at AGS 2009 workshop.

Finally, I am grateful to Ion Gaztañaga, the main developer of the *Boost.Interprocess* library, for his support and help with the pre-release of his library. *Jazzyk* interpreter couldn't technically work the way it does without his portable package.

Appendices

Appendix A

Implementation of Jazzyk interpreter

A.1 Architecture

Technically, *Jazzyk* interpreter is implemented in *C++* as a standalone command line tool. The KR modules are shared dynamically loaded libraries, installed as standalone packages on a host operating system. When a KR module is loaded, the *Jazzyk* interpreter forks a separate process to host it. The communication between the *Jazzyk* interpreter and a set of the KR module subprocesses is facilitated by an OS specific shared memory subsystem. This allows loading multiple instances of the same KR module implemented in a portable way. Figure A.1 depicts the technical architecture of the *Jazzyk* interpreter.

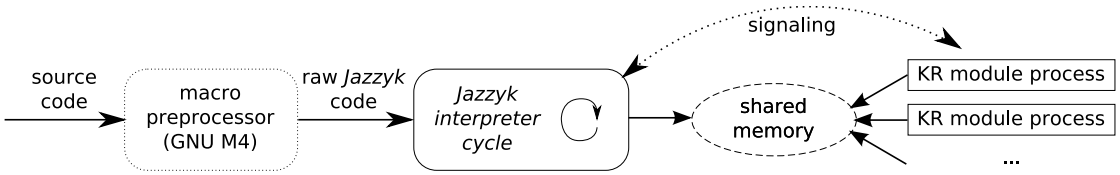


Figure A.1: *Jazzyk* interpreter scheme

Jazzyk interpreter was written with portability in mind. It should be relatively easy to compile, install or port to most POSIX compliant operating systems. As of the time of writing this thesis, *Jazzyk* interpreter was developed and well tested on Linux operating system. It was reported to run also on Mac OS X, yet no support is provided for this system. The interpreter was also successfully experimentally ported to Microsoft Windows XP^{TM}operating system with Cygwin support, however currently no support is provided for this port either. The *Jazzyk* interpreter was published under the open-source GNU GPL v2 license and is hosted at `http://jazzyk.sourceforge.net/`.

A.2 Installation

These are basic instructions on how to install and start to use Jazzyk interpreter. For advanced issues and deeper insights consult the project documentation.

The preferred and currently the only way to install Jazzyk interpreter is to download the source code and compile and install it manually by yourself. To download the *Jazzyk* sources get the "jazzyk-[version].tar.gz" or "jazzyk-[version].tar.bz2" package from the file repository on the project website. To be able to develop or run Jazzyk agent programs, you need to install also Jazzyk KR modules.

Jazzyk is built using standard GNU Autotools chain. It is developed in C++ and compiled with GNU GCC G++ compiler. To compile *Jazzyk* you must have the standard GNU development tool chain installed together with some additional libraries, in particular

- GNU Libtool 2,
- GNU GCC G++ compiler,
- GNU Bison,
- GNU Flex,
- GNU M4, and
- C++ Standard Template Library.

Preferably, use the latest versions of the tools. Even though *Jazzyk* first developed with GNU gcc 4.2.x C++ compiler series, as of version 1.20 it was adapted to compile well with GNU gcc 4.3 series. It should still compile well with older gcc versions, but we are not going to actively support those any more.

The compilation and installation process follows the standard Autotools command chain:

```
$ ./configure
$ make
$ make install
```

The installation command should be invoked with administrator privileges on your system. For advanced options consult the file INSTALL distributed within the installation package, or execute the configure command, with --help option. To be able to execute Jazzyk programs, however, install also KR modules, *Jazzyk* plug-in packages. These can be also downloaded from the project website.

A.3 Jazzyk interpreter manual page

Jazzyk
interpreter of the *Jazzyk* programming language

(*In varietate concordia!*)

Synopsis

jazzyk [*options*] [*file*]

Description

Jazzyk is an experimental, special-purpose programming language for development of knowledge intensive (intelligent) agent systems. *Jazzyk* agents consist of

- a number of knowledge bases, each realized by a separate
- specialized knowledge representation module (plug-in), and an agent program in a form of a set of possibly nested rules of the basic form: `when Query then Update`.

Jazzyk was designed to exploit the power of heterogeneous knowledge representation (KR) technologies in a single agent system. Each such KR technology is encapsulated in a separate *Jazzyk* KR module providing a simple generic interface consisting of a set of query and update operations. Semantics of *Jazzyk* based on *Behavioural State Machines*, an adaptation of computational model of Gurevich's *Abstract State Machines*.

Theory of *Behavioural State Machines*, and in turn also *Jazzyk*, draws a strict distinction between agent's knowledge representational and behavioural aspects. While an agent's deliberation abilities reside in its KR modules, its behaviour are encoded as a Behavioural State Machine.

Options

General options:

–**help** display the help message and exit

–**version** output version information and exit

–**license** display the GNU GPL license information and exit

Compiler options:

-0 [–**stdin**] after reading in program files, read also standard input. This option is useful to execute for example automatically generated programs, or programs preprocessed by a specialized filters.

-I [–**include** ***path***] macro preprocessor include path(s); Provided include paths will be passed to the internal macro preprocessor invocation via **-I** option. By default, the macro preprocessor searches only in the current path (path in which the interpreter was invoked (see *pwd*(1))). The source code can include also files from other then the current directory, but it has to use relative paths to find the included file correctly. Otherwise all the paths where include files reside, should be passed to the interpreter using **-I** options.

–**no-mp** bypass the macro preprocessor; Internal macro preprocessor will not be invoked on the input file(s) and the content will be passed directly to the compiler. This option can be useful in the case the input source code either does not use any higher level macro overlay definitions, or it is already preprocessed by the macro preprocessor.

Interpreter options:

-L [–**libraries** ***path***] add an absolute system path to the location of external plug-ins; Plug-ins are standard shared dynamically loadable libraries and by default the interpreter searches in the standard system paths where libraries are present. Use this option in the case you have KR plug-ins installed in a non-standard location (w.r.t. your OS), or the interpreter has difficulties to find the requested plug-ins.

-o [–**ordered**] pick the first applicable rule to apply from a set transformer; By default, when the interpreter executes a set mental state transformer {<transformer> ; <transformer}, it first randomly shuffles the transformer set and then searches for an applicable rule from the beginning. This way a random applicable rule is chosen from the original set transformer. Using this option disables the random shuffling step and lets the interpreter to choose the first applicable rule of the set transformer w.r.t. the ordering as defined in the source code.

-n [–**cycles** ***num***] perform only *num* interpreter cycles and quit; 0 (default) means endless interpreter cycle loop.

-q [–**no-check-query**] switch off checking sanity of resulting query variable substitutions. Modules are allowed to not substitute all the provided free variables.

Debug options:

-e [–**only-macros**] print the output of macro preprocessor run only; do not compile, or interpret; Using this option amounts to the same effect as if *GNU M4* (*m4*(1))

were executed with the default options described later in this manual and the corresponding input file (or standard input stream).

-E [–**compile**] run preprocessor and compile only, i.e., do not interpret. This option is useful for verifying whether the program is interpretable. This means that it is well-formed according to the syntax rules of the *Jazzyk* language and additional semantical constraints hold: each module referenced either in update, query, or notification expression/statement is also declared in the program (Note: it can be declared also after the first use!)

-p [–**print**] pretty print the program tree structure after compilation stage; This is a convenience feature to check that the compiler understood the structure of the program correctly. Currently it prints only a rough structure of the program. In the future it should print the executable formatted source code of the input program.

Diagnostics

When failing, the interpreter provides informative error messages in the standard compiler format with the location in the source file and an error message. The error output can be parsed and processed by IDEs and text editors like *vim*(1), *emacs*(1) and others. Messages are written to the standard error output.

During the internal invocation of the *GNU M4* (*m4*) macro preprocessor, the interpreter forwards whatever error messages of *m4* as well.

GNU M4 invocation

GNU M4 macro preprocessor is internally invoked with the following default arguments:

```
m4 -s -E -I $(PACKAGELIBDIR) ...
```

Where `$(PACKAGELIBDIR)` stands for the default installation directory where internally used shared macros are placed. When the default installation prefix is used, this should be `/usr/lib/jazzyk`. For more details, see the installation instructions of the *jazzyk* package and help message of its *configure* script.

For more details on the *m4* options semantics see *m4*(1).

Bugs

As of time of writing this manual, no issues and bugs are known to me. For more details see the `Changelog` file. In the case you will spot any bugs, or problems, or you have some enhancement/feature request, do not hesitate and contact the author, or maintainer.

Author

The *Jazzyk* interpreter was written by Peter Novak `peter.novak@tu-clausthal.de` as a result of his research work towards PhD. degree in *Computational Intelligence Group* of *Clausthal University of Technology*, Clausthal, Germany
(`http://cig.in.tu-clausthal.de`).

Peter Novák <`peter.novak@tu-clausthal.de`>, `http://peter.aronde.net/`

See also

GNU M4 *m4*(1).

Copying permissions

Appendix B

Jazzyk SDK

To facilitate and support development of new KR modules, *Jazzyk* plug-ins, I developed *Jazzyk Software Development Kit*. *Jazzyk SDK* is distributed as a separate package jazzyk-sdk-[version].tar.gz. To use it, the host system must have the core *Jazzyk* interpeter installation. In particular, it needs the two header files jazzyk_common.hpp and jazzyk.hpp, which are installed by default to /usr/local/include/jazzyk/ directory. For a KR module developer, only the latter header file is important. The file contains an abstract class AbstractKRModule which provides the template as well as an implementation of a stock KR module. Building a custom KR module is as simple as inheriting this class and implementing the required methods.

In the file module.cpp (listed later in this appendix), the package contains a sample implementation of a dummy KR module querying and "updating" standard input and output. Additionally, all the boilerplate GNU Autotools configuration, build and deployment files are included as well.

Sample KR module implementation (module.cpp)

```
/*
 ****************************************************************************
 * Jazzyk - Modular BDI Agent Architecture programming language interpreter
 * Copyright (C) 2006, 2007 Peter Novak <pno at aronde.net>
 *
 * Jazzyk is free software; you can redistribute it and/or modify it under the
 * terms of the GNU General Public License as published by the Free Software
 * Foundation; either version 2 of the License, or (at your option) any later
 * version.
 *
 * This program is distributed in the hope that it will be useful, but WITHOUT
 * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
 * FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for
 * more details.
 *
 * You should have received a copy of the GNU General Public License along with
 * this program; if not, write to the Free Software Foundation, Inc., 59 Temple
 * Place, Suite 330, Boston, MA 02111-1307 USA
 ****************************************************************************
*/

/*
 * $Id: module.cpp 1760 2009-07-02 22:11:58Z pno $
 * $Author: pno $
 * $Description: Template file for a Jazzyk KR module development (SDK).$
 *
 * $Revision: 1760 $
 *
*/

/*!
 * This is a template example file for development of a custom knowledge
 * representation module for the Jazzyk interpreter.
 *
 * The full documentation on how to create new KR modules can be found in the
 * official Jazzyk documenation to the jazzyk.hpp file.
 *
 * For more documentation, please check the README file of the Jazzyk SDK.
 *
 */

/*
 * If you get a compile error here because of missing jazzyk.hpp header, either
 * 1) your compiler is misconfigured and does not find the correct header where
 * it expects (by default /usr/local/include/), or 2) you did not install the
 * package Jazzyk (Jazzyk interpreter) correctly.
 */
#include <jazzyk/jazzyk.hpp>


//! Knowledge representation module implementation class
/*!
 * In order to create a new module, simply create a subclass of
 * jazzyk::AbstractKRModule, implement required virtuals, implement yor
 * specialized query/update interface and finally register this interface using
 * macros of the JZMODULE_MANIFEST family.
```

```
 *
 * NOTE:
 *  Methods' definitions are implemented in the body class for brevity
 *  only. Normally, you should put them into a separate .cpp implementation
 *  file, of course.
 */

#include <iostream>

class CMyModule :
    public jazzyk::AbstractKRModule
{
public:

    //! KR module initialization callback
    /*!
     * The routine is invoked right after the module is loaded before the
     * Jazzyk program interpretation itself.
     *
     * The argument is a KR module specific code to be executed on the
     * initialize notification specified in the Jazzyk program.
     */
    virtual jazzyk::EKRModuleError initialize(const std::string& szCode)
    {
        /* Put your initialization code here. */

        std::cout << "\"" << szCode << "\"" << std::endl;

        return jazzyk::OK;
    }

    //! KR module finalization callback
    /*!
     * The routine is invoked right before the module is about to be
     * unloaded.
     *
     * The argument is a KR module specific code to be executed on the
     * finalize notification specified in the Jazzyk program.
     *
     * Note:
     *  In the case of a non normal interpreter exit, the interpreter
     *  does it's best to call finalize on all its modules, however it
     *  might happen that it won't be called.
     */
    virtual jazzyk::EKRModuleError finalize(const std::string& szCode)
    {
        /* Put your finalization code here. */

        std::cout << "\"" << szCode << "\"" << std::endl;

        return jazzyk::OK;
    }

    //! KR module cycle callback
    /*!
     * The routine is invoked after each completed interpreter deliberation
     * cycle.
     *
```

```
 * The argument is a KR module specific code to be executed on the
 * cycle notification specified in the Jazzyk program.
 */
virtual jazzyk::EKRModuleError cycle(const std::string& szCode)
{
    /* Put your cycle notification code here. */

    std::cout << "\"" << szCode << "\"" << std::endl;

    return jazzyk::OK;
}

//! Custom KR module query operation
/*!
 * When the module is queried, a code block (query formula) [first
 * argument] is passed to it together with the current substitutions of
 * the relevant variables [second argument]. The query routine is
 * supposed to return a query formula execution result [fifth argument]
 * and a new variable substitution for the query-formula-relevant free
 * variables [fourth argument].
 *
 * Remark:
 *  This method is normally invoked very often. Therefore
 *  optimization of its time complexity is important for a general
 *  KR module quality.
 */
virtual jazzyk::EKRModuleError queryImplementation(
        const std::string& szQueryCode,
        const jazzyk::TKRVarSubstitution& inSubst,
        jazzyk::TKRVarSubstitution& outSubst,
        bool& bResult)
{
    /*
     * Implementation of the query interface goes here and
     * to methods with the same signature.
     */

    std::cout << "\"" << szQueryCode << "\"" << std::endl;

    // Example of variable substitution handling
    if (!outSubst.empty())
    {
        std::cout
            << "substituting␣variable␣"
            << outSubst.begin()->first << std::endl;
        outSubst[outSubst.begin()->first] = "A␣value";
    }

    // IMPORTANT: fill in the query result
    bResult = true;

    return jazzyk::OK;
}

//! Custom KR module update operation
/*!
 * When finally the Jazzyk interpreter performs an update operation on
 * a KR module, this routine is invoked. The module gets a code block
```

```
     * to be executed (update formula) [first argument], together with a
     * relevant variable substitution [second argument].
     */
    virtual jazzyk::EKRModuleError updateImplementation(
            const std::string& szUpdateCode,
            const jazzyk::TKRVarSubstitution& mapSubstitution)
    {
        /*
         * Implementation of the update interface goes here and
         * to methods with the same signature.
         */

        std::cout << "\"" << szUpdateCode << "\"" << std::endl;

        if (!mapSubstitution.empty())
        {
            std::cout
                << "Variable:␣" << mapSubstitution.begin()->first
                << "␣Value:␣" << mapSubstitution.begin()->second
                << std::endl;
        }

        return jazzyk::OK;
    }
};

//! KR module query/update interface manifest
/*!
 * KR module must publish its query/update interface and so bind the custom
 * query/update routine implementations to the query/update operations used in
 * Jazzyk programs. To do so, macros of the JZMODULE_MANIFEST* family are used.
 *
 * The manifest block starts with the JZMODULE_MANIFEST_BEGIN macro with an
 * argument of the KR module implementation class. It has to be finished by the
 * JZMODULE_MANIFEST_END macro. Between the pair, query and update operations
 * are registered using REGISTER_QUERY and REGISTER_UPDATE registration macros.
 * The format of their arguments is the same: the first is the operation
 * identifier to be used in the Jazzyk programs, while the second is its
 * binding to a custom KR module query/update routine. In a basic case, a
 * module should have at least one query and one update operation.
 *
 * NOTE:
 *  Only one manifest can be defined in the KR module! Therefore be careful
 *  about its placement (header files are a bad candidate – the main .cpp
 *  file is a good one).
 */
JZMODULE_MANIFEST_BEGIN(CMyModule)
    REGISTER_QUERY("query", queryImplementation)
    REGISTER_UPDATE("update", updateImplementation)
JZMODULE_MANIFEST_END
```

Bibliography

Adobbati, R.; Marshall, A.N.; Scholer, A.; Tejada, S.; Kaminka, G.A.; Schaffer, S. and Sollitto, C. *Gamebots: A 3D Virtual World Test-Bed For Multi-Agent Research*. In *Proceedings of the Second International Workshop on Infrastructure for Agents, MAS, and Scalable MAS*. 2001.

AlienTrap Community. *AlienTrap Software*. `http://www.alientrap.org/`, 2009.

Arnold, Ken; Gosling, James and Holmes, David. *The Java Programming Language, Third Edition*. Addison-Wesley, 2000. ISBN 0-201-70433-1.

Astefanoaei, L.; Mol, C. P.; Sindlar, M. P. and Tinnemeier, N. A. M. *Going for gold with 2APL*. In *Proceedings of Fifth international Workshop on Programming Multi-Agent Systems, ProMAS'07*, volume 4908 of *LNAI*. Springer Verlag, 2008.

Baral, Chitta. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, Cambridge, 2003.

Behrens, Tristan M.; Dix, Jürgen; Dastani, Mehdi; Köster, Michael and Novák, Peter. *Technical Foundations of the Agent Contest 2008*. Technical Report IfI-08-05, Clausthal University of Technology, 2008. URL `http://www.in.tu-clausthal.de/fileadmin/homes/techreports/ifi0805behrens.pdf`.

Behrens, Tristan M.; Dix, Jürgen; Dastani, Mehdi; Köster, Michael and Novák, Peter. *MASSim: Technical Infrastructure for AgentContest Competition Series*. `http://www.multiagentcontest.org/`, 2009a.

Behrens, Tristan Marc; Dix, Jürgen; Dastani, Mehdi; Köster, Michael and Novák, Peter. *Multi-Agent Programming Contest*. `http://www.multiagentcontest.org/`, 2009b.

Bellifemine, Fabio; Bergenti, Federico; Caire, Giovanni and Poggi, Agostino. *JADE - A Java agent development framework*, chapter 5, pp. 125–147. Volume 15 of Bordini et al. (2005a), 2005.

Bergenti, Federico; Gleizes, Marie-Pierre and Zambonelli, Franco (eds.). *Methodologies and Software Engineering for Agent Systems: The Agent-Oriented Software Engineering Handbook*, volume 11 of *Multiagent Systems, Artificial Societies, and Simulated Organizations*. Kluwer Academic Publishers, 2004. ISBN 1-4020-8057-3. doi: 10.1007/1-4020-8058-1_17.

de Boer, Frank S.; Hindriks, Koen V.; van der Hoek, Wiebe and Meyer, John-Jules Ch. *A verification framework for agent programming with declarative goals. J. Applied Logic*, volume 5(2):pp. 277–302, 2007.

Bordini, Rafael H.; Braubach, Lars; Dastani, Mehdi; Seghrouchni, Amal El Fallah; Gomez-Sanz, Jorge J.; Leite, João; O'Hare, Gregory; Pokahr, Alexander and Ricci, Alessandro. *A survey of programming languages and platforms for multi-agent systems. Informatica*, volume 30:pp. 33–44, 2006. ISSN 03505596.

Bordini, Rafael H.; Dastani, Mehdi; Dix, Jürgen and Fallah-Seghrouchni, Amal El (eds.). *Multi-Agent Programming: Languages, Tools and Applications*. Springer, Berlin, 2009. ISBN 978-0-387-89298-6. URL `http://www.springer.com/computer/artificial/book/978-0-387-89298-6`.

Bordini, Rafael H.; Dastani, Mehdi; Dix, Jürgen and Seghrouchni, Amal El Fallah. *Multi-Agent Programming Languages, Platforms and Applications*, volume 15 of *Multiagent Systems, Artificial Societies, and Simulated Organizations*. Kluwer Academic Publishers, 2005a. ISBN 0-387-24568-5.

Bordini, Rafael H.; Fisher, Michael; Pardavila, Carmen and Wooldridge, Michael. *Model checking AgentSpeak*. In *AAMAS*, pp. 409–416. ACM, 2003. ISBN 1-58113-683-8.

Bordini, Rafael H.; Hübner, Jomi F. and Vieira, Renata. *Jason and the Golden Fleece of Agent-Oriented Programming*, chapter 1, pp. 3–37. Volume 15 of Bordini et al. (2005a), 2005b.

Bordini, Rafael H.; Hübner, Jomi Fred and Wooldridge, Michael. *Programming Multi-agent Systems in AgentSpeak Using Jason*. Wiley Series in Agent Technology. Wiley-Blackwell, 2007. ISBN 978-0-470-02900-8.

Börger, Egon and Stärk, Robert F. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer, 2003.

Bratman, Michael E. *Intention, Plans, and Practical Reason*. Cambridge University Press, 1999. ISBN 1575861925.

Cheyer, Adam and Martin, David L. *The Open Agent Architecture. Journal of Autonomous Agents and Multi-Agent Systems*, volume 4(1/2):pp. 143–148, 2001.

Cohen, Philip R. and Levesque, Hector J. *Intention is choice with commitment. Artif. Intell.*, volume 42(2-3):pp. 213–261, 1990.

Cyberbotics Inc. Webots 6 *mobile robotics simulation software*. `http://www.cyberbotics.com/products/webots/`, 2009.

Dastani, Mehdi. *2APL: a practical agent programming language. Autonomous Agents and Multi-Agent Systems*, volume 16(3):pp. 214–248, 2008.

Dastani, Mehdi; Dix, Jürgen and Novák, Peter. *The first contest on multi-agent systems based on computational logic.* In Francesca Toni and Paolo Torroni (eds.), *CLIMA VI*, volume 3900 of *Lecture Notes in Computer Science*, pp. 373–384. Springer, 2005a. ISBN 3-540-33996-5.

Dastani, Mehdi; Dix, Jürgen and Novák, Peter. *The second contest on multi-agent systems based on computational logic.* In Katsumi Inoue; Ken Satoh and Francesca Toni (eds.), *CLIMA VII*, volume 4371 of *Lecture Notes on Computer Science*, pp. 266–283. Springer, 2006. ISBN 978-3-540-69618-6.

Dastani, Mehdi; Dix, Jürgen and Novák, Peter. *Agent Contest Competition - 3rd edition.* In *Proceedings of Fifth international Workshop on Programming Multi-Agent Systems, ProMAS'07*, volume 4908 of *LNAI*, pp. 221–240. Springer Verlag, 2008a.

Dastani, Mehdi; Dix, Jürgen and Novák, Peter. *Agent Contest Competition - 4th edition.* In *Proceedings of Sixth international Workshop on Programming Multi-Agent Systems, ProMAS'08*, volume 5442 of *LNAI*. Springer Verlag, 2008b.

Dastani, Mehdi; Fallah-Seghrouchni, Amal El; Ricci, Alessandro and Winikoff, Michael (eds.). *Programming Multi-Agent Systems, 5th International Workshop, ProMAS 2007, Honolulu, HI, USA, May 15, 2007, Revised and Invited Papers*, volume 4908 of *Lecture Notes in Computer Science*. Springer, 2008c. ISBN 978-3-540-79042-6.

Dastani, Mehdi; Hindriks, Koen V.; Novák, Peter and Tinnemeier, Nick A. M. *Combining multiple knowledge representation technologies into agent programming languages.* In Matteo Baldoni; Tran Cao Son; M. Birna van Riemsdijk and Michael Winikoff (eds.), *DALT*, volume 5397 of *Lecture Notes in Computer Science*, pp. 60–74. Springer, 2008d. ISBN 978-3-540-93919-1.

Dastani, Mehdi; Mol, Christian P. and Steunebrink, Bas R. *Modularity in agent programming languages.* In The Duy Bui; Tuong Vinh Ho and Quang-Thuy Ha (eds.), *PRIMA*, volume 5357 of *Lecture Notes in Computer Science*, pp. 139–152. Springer, 2008e. ISBN 978-3-540-89673-9.

Dastani, Mehdi; van Riemsdijk, Birna; Hulstijn, Joris; Dignum, Frank and Meyer, John-Jules Ch. *Enacting and deacting roles in agent programming.* In James Odell; Paolo Giorgini and Jörg P. Müller (eds.), *AOSE*, volume 3382 of *Lecture Notes in Computer Science*, pp. 189–204. Springer, 2004. ISBN 3-540-24286-4.

Dastani, Mehdi; van Riemsdijk, M. Birna and Meyer, John-Jules. *Programming Multi-Agent Systems in 3APL*, chapter 2, pp. 39–68. Volume 15 of Bordini et al. (2005a), 2005b.

Davis, Randall; Shrobe, Howard E. and Szolovits, Peter. *What is a knowledge representation? AI*, volume 14(1):pp. 17–33, 1993.

DeLoach, Scott A. *The MaSE Methodology*, chapter 6, pp. 107–125. Volume 11 of Bergenti et al. (2004), 2004. doi:10.1007/1-4020-8058-1_17.

Dennett, Daniel C. *The Intentional Stance (Bradford Books)*. The MIT Press, Cambridge, MA, 1987. ISBN 0262540533.

Dennis, Louise A. and Farwer, Berndt. *Gwendolen: A BDI Language for Verifiable Agents*. In Benedikt Löwe (ed.), *Logic and the Simulation of Interaction and Reasoning*. AISB, Aberdeen, 2008. AISB'08 Workshop.

Dennis, Louise A.; Farwer, Berndt; Bordini, Rafael H.; Fisher, Michael and Wooldridge, Michael. *A Common Semantic Basis for BDI Languages*. In Dastani et al. (2008c), pp. 124–139.

Dereń, Slawomir. *Inter-Agent-Kommunikation: Modul für Jazzyk*. Master's thesis, Department of Informatics, Clausthal University of Technology, forthcoming, 2009.

Emerson, E. Allen. *Temporal and modal logic*. In Jan van Leeuwen (ed.), *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, pp. 995–1072. Elsevier and MIT Press, 1990. ISBN 0-444-88074-7, 0-262-22039-3.

EPFL. *e-puck: EPFL education robot*. `http://www.e-puck.org/`, 2006.

Epic Games, Inc. *Unreal Tournament*. `http://www.unrealtournament3.com/`, 2004.

FIPA. *FIPA Abstract Architecture Specification*. available from `http://www.fipa.org/`, 2000–2009a. #SC00001.

FIPA. *FIPA Agent Management Specification*. available from `http://www.fipa.org/`, 2000–2009b. #SC00023.

FIPA. *Foundation for Intelligent Physical Agents, FIPA specification*. 2000–2009c. Available from `http://www.fipa.org/`.

Fisher, Michael. *Concurrent MetateM - a language for modelling reactive systems*. In Arndt Bode; Mike Reeve and Gottfried Wolf (eds.), *PARLE*, volume 694 of *Lecture Notes in Computer Science*, pp. 185–196. Springer, 1993. ISBN 3-540-56891-3.

Fisher, Michael. *A survey of Concurrent MetateM - the language and its applications*. In Dov M. Gabbay and Hans Jürgen Ohlbach (eds.), *ICTL*, volume 827 of *Lecture Notes in Computer Science*, pp. 480–505. Springer, 1994. ISBN 3-540-58241-X.

Fisher, Michael and Hepple, Anthony. *Executing Logical Agent Specifications*, chapter 1, pp. 3–27. In Bordini et al. (2009), 2009. URL `http://www.springer.com/computer/artificial/book/978-0-387-89298-6`.

Flanagan, David and Matsumoto, Yukihiro. *The Ruby Programming Language*. O'Reilly Media, Inc., 2008. ISBN 978-0596516178.

Fuhrmann, Bernd. *Implementierung eines URBI- und Rubymoduls für Jazzyk zur Entwicklung von Robotern*. Master's thesis, Department of Informatics, Clausthal University of Technology, forthcoming, 2009.

GCC team. *GCC, the GNU Compiler Collection*. `http://gcc.gnu.org/`, 2009.

Gelfond, Michael and Lifschitz, Vladimir. *The Stable Model Semantics for Logic Programming*. In *ICLP/SLP*, pp. 1070–1080. 1988.

Gemrot, Jakub; Kadlec, Rudolf; Bída, Michal; Bukert, Ondřej; Píbil, Radek; Havlíček, Jan; Zemčák, Lukáš; Šimlovič, Juraj; Vansa, Radim; Štolba, Michal and Brom, Cyril. *Pogamut 3 can assist developers in building AI for their videogame agents*. In *Proceedings of Agents for Games and Simulations, AGS 2009, AAMAS 2009 collocated workshop*, pp. 144–148. 2009.

Georgeff, Michael P. and Lansky, Amy L. *Reactive reasoning and planning*. In *AAAI*, pp. 677–682. 1987.

Giacomo, Giuseppe De; Lespérance, Yves and Levesque, Hector J. *ConGolog, a concurrent programming language based on the situation calculus. Artif. Intell.*, volume 121(1-2):pp. 109–169, 2000.

Ginsberg, Matthew L. *Universal planning: An (almost) universally bad idea. AI Magazine*, volume 10(4):pp. 40–44, 1989.

Giorgini, Paolo; Kolp, Manuel; Mylopoulos, John and Pistore, Marco. *The Tropos Methodology*, chapter 5, pp. 88–106. Volume 11 of Bergenti et al. (2004), 2004. doi: 10.1007/1-4020-8058-1_17.

Global InfoTek, Inc. *CoABS Grid*. `http://coabs.globalinfotek.com/`, 2009.

Gostai. *Gostai - robotics for everyone*. `http://www.gostai.com/`, 2009a.

Gostai. *URBI Doc for Webots*. `http://www.gostai.com/doc/en/webots/`, 2009b.

Gostai. *Urbi tutorials & documentation for robotics*. `http://www.gostai.com/doc.php`, 2009c.

Gumstix Inc. *Gumstix: Computer-on-module.* `http://www.gumstix.com/store/catalog/motherboards.php`, 2009.

Gurevich, Yuri. *Evolving algebras.* In *IFIP Congress (1)*, pp. 423–427. 1994.

Hale, Forest "LordHavoc". *DarkPlaces Quake Modification.* `http://icculus.org/twilight/darkplaces/`, 2009.

Harel, David; Kozen, Dexter and Tiuryn, Jerzy. *Dynamic logic.* In *Handbook of Philosophical Logic*, pp. 497–604. MIT Press, 1984.

Helsinger, Aaron; Thome, Michael and Wright, Todd. *Cougaar: a scalable, distributed multi-agent architecture.* In *SMC (2)*, pp. 1910–1917. IEEE, 2004. ISBN 0-7803-8566-7.

Henriksen, Jesper G. and Thiagarajan, P. S. *Dynamic linear time temporal logic. Ann. Pure Appl. Logic*, volume 96(1-3):pp. 187–207, 1999.

Hindriks, Koen V. *Agent Programming Languages: Programming with Mental Models.* Ph.D. thesis, Utrecht University, 2001.

Hindriks, Koen V. *Modules as policy-based intentions: Modular agent programming in goal.* In Dastani et al. (2008c), pp. 156–171.

Hindriks, Koen V. *Programming Rational Agents in GOAL*, chapter 4, pp. 119–157. In Bordini et al. (2009), 2009. URL `http://www.springer.com/computer/artificial/book/978-0-387-89298-6`.

Hindriks, Koen V.; de Boer, Frank S.; van der Hoek, Wiebe and Meyer, John-Jules Ch. *Agent Programming in 3APL. Autonomous Agents and Multi-Agent Systems*, volume 2(4):pp. 357–401, 1999.

Hindriks, Koen V.; Jonker, Catholijn M. and Pasman, Wouter. *Exploring heuristic action selection in agent programming.* In Koen Hindriks; Alexander Pokahr and Sebastian Sardina (eds.), *Proceedings of the Sixth International Workshop on Programming Multi-Agent Systems, ProMAS'08, Estoril, Portugal*, volume 5442 of *LNAI*. 2008.

Hindriks, Koen V. and Novák, Peter. *Compiling GOAL agent programs into Jazzyk Behavioural State Machines.* In Ralph Bergmann; Gabriela Lindemann; Stefan Kirn and Michal Pěchouček (eds.), *MATES*, volume 5244 of *Lecture Notes in Computer Science*, pp. 86–98. Springer, 2008. ISBN 978-3-540-87804-9.

Hindriks, Koen V. and van Riemsdijk, M. Birna. *A Computational Semantics for Communicating Rational Agents Based on Mental Models.* In *Proceedings of The Seventh International Workshop on Programming Multi-Agent Systems ProMAS 2009, AAMAS 2009 collocated workshop.* 2009.

Hübner, Jomi F. and Bordini, Rafael H. *Developing a team of gold miners using Jason.* In *Proceedings of Fifth international Workshop on Programming Multi-Agent Systems, ProMAS'07*, volume 4908 of *LNAI*. Springer Verlag, 2008.

Hübner, Jomi F.; Bordini, Rafael H. and Picard, Gauthier. *Using Jason to develop a team of cowboys: a preliminary design for Agent Contest 2008.* In *Proceedings of Sixth International Workshop on Programming Multi-Agent Systems, ProMAS 2008*. 2008.

Hübner, Jomi Fred; Bordini, Rafael H. and Wooldridge, Michael. *Plan patterns for declarative goals in AgentSpeak.* In Nakashima et al. (2006), pp. 1291–1293.

Hübner, Jomi Fred; Bordini, Rafael H. and Wooldridge, Michael. *Programming declarative goals using plan patterns.* In Matteo Baldoni and Ulle Endriss (eds.), *DALT*, volume 4327 of *Lecture Notes in Computer Science*, pp. 123–140. Springer, 2006b. ISBN 3-540-68959-1.

id Software, Inc. *Quake III Arena.* `http://www.idsoftware.com/games/quake/quake3-arena/`, 1999.

Jones, Randolph M. and Wray III, Robert E. *Comparative analysis of frameworks for knowledge-intensive intelligent agents. AI Magazine*, volume 27(2):pp. 45–56, 2006.

Jones, Simon Peyton. *How to write a great research paper.* `http://research.microsoft.com/en-us/um/people/simonpj/papers/giving-a-talk/writing-a-paper-slides.pdf`, 2005.

Josefsson, Simon. *RFC 4648: The Base16, Base32, and Base64 Data Encodings.* `http://tools.ietf.org/html/rfc4648/`, 2006.

Kahn, Martha L. and Cicalese, Cynthia Della Torre. *The CoABS Grid.* In Walt Truszkowski; Christopher Rouff and Michael G. Hinchey (eds.), *WRAC*, volume 2564 of *Lecture Notes in Computer Science*, pp. 125–134. Springer, 2002. ISBN 3-540-40725-1.

Köster, Michael. *Implementierung eines autonomen Agenten in einer simulierten 3D-Umgebung - Wissensrepräsentation.* Master's thesis, Department of Informatics, Clausthal University of Technology, 2008.

Köster, Michael; Novák, Peter; Mainzer, David and Fuhrmann, Bernd. *Two case studies for Jazzyk BSM.* In *Proceedings of Agents for Games and Simulations, AGS 2009, AAMAS 2009 co-located workshop*, pp. 31–45. 2009.

Kripke, Saul. *Semantical Considerations on Modal Logic. Acta Phil. Fennica*, volume 16:pp. 83–94, 1963.

Laird, John E. *It knows what you're going to do: adding anticipation to a Quakebot.* In *Proceedings of the fifth international conference on Autonomous agents*, pp. 385–392. ACM New York, NY, USA, 2001.

Laird, John E. and van Lent, Michael. *Human-level AI's killer application: Interactive computer games. AI Magazine*, volume 22(2):pp. 15–26, 2001.

Leite, João Alexandre. *Evolving Knowledge Bases*, volume 81 of *Frontiers of Artificial Intelligence and Applications*. IOS Press, 2003. ISBN 1-58603-278-X.

Levesque, Hector J.; Pirri, Fiora and Reiter, Raymond. *Foundations for the situation calculus. Electron. Trans. Artif. Intell.*, volume 2:pp. 159–178, 1998.

Levesque, Hector J.; Reiter, Raymond; Lespérance, Yves; Lin, Fangzhen and Scherl, Richard B. *Golog: A logic programming language for dynamic domains. J. Log. Program.*, volume 31(1-3):pp. 59–83, 1997.

M4 team. *GNU M4, version 1.4.13*. `http://www.gnu.org/software/m4/`, 2009.

Madden, Neil and Logan, Brian. *Modularity and compositionality in Jason.* In *Proceedings of The Seventh International Workshop on Programming Multi-Agent Systems ProMAS 2009, AAMAS 2009 collocated workshop*. 2009.

Mainzer, David. *Implementierung eines autonomen Agenten in einer simulierten 3D-Umgebung - Interaktion mit der Umwelt.* Master's thesis, Department of Informatics, Clausthal University of Technology, 2008.

Manna, Zohar and Pnueli, Amir. *The temporal logic of reactive and concurrent systems.* Springer-Verlag New York, Inc., New York, NY, USA, 1992. ISBN 0-387-97664-7.

Markov, Andrey Andreyevich. *Extension of the law of large numbers to dependent quantities (in Russian). Izvestiya Fiziko-matematicheskogo obschestva pri Kazanskom Universitete*, volume 2(15):pp. 135–156, 1906.

Matsumoto, Yukihiro. *Ruby Programming Language.* `http://www.ruby-lang.org/`, 2009.

Meyer, Bertrand. *Introduction to the Theory of Programming Languages.* Prentice-Hall, 1990. ISBN 0-13-498510-9.

Meyn, S. P. and Tweedie, R. L. *Markov Chains and Stochastic Stability.* Springer-Verlag, London, 1993. URL `http://probability.ca/MT/`.

Michel, Olivier. *Webots: Symbiosis between virtual and real mobile robots.* In *Virtual Worlds*, volume 1434 of *Lecture Notes in Computer Science*, pp. 254–263. Springer

Berlin / Heidelberg, 1998. ISBN 978-3-540-64780-5. ISSN 0302-9743 (Print) 1611-3349 (Online). doi:10.1007/3-540-68686-X_24. URL `http://www.springerlink.com/content/x66rj837q1021189`.

Michel, Olivier. Webots^{TM}*: Professional Mobile Robot Simulation. International Journal of Advanced Robotic Systems*, volume 1(1), 2008. ISSN 1729-8806.

Mondada, F.; Bonani, M.; Raemy, X.; Pugh, J.; Cianci, C.; Klaptocz, A.; Magnenat, S.; Zufferey, J.-C.; Floreano, D. and Martinoli, A. *The e-puck, a Robot Designed for Education in Engineering.* In *Proceedings of the 9th Conference on Autonomous Robot Systems and Competitions*, pp. 59–65. 2009.

Nakashima, Hideyuki; Wellman, Michael P.; Weiss, Gerhard and Stone, Peter (eds.). *5th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2006), Hakodate, Japan, May 8-12, 2006*. ACM, 2006. ISBN 1-59593-303-4.

Nexuiz Team. Nexuiz *computer game, version 2.3*. `http://www.alientrap.org/nexuiz/`, 2007.

Novák, Peter. *Behavioural State Machines: programming modular agents.* In *AAAI 2008 Spring Symposium: Architectures for Intelligent Theory-Based Agents, AITA'08*. 2008a.

Novák, Peter. *Communication platform for open heterogeneous MASs.* Technical Report IfI-08-13, Clausthal University of Technology, 2008b. URL `http://www.in.tu-clausthal.de/fileadmin/homes/techreports/ifi0813novak.pdf`.

Novák, Peter. *Jazzyk: A programming language for hybrid agents with heterogeneous knowledge representations.* In *Proceedings of the Sixth International Workshop on Programming Multi-Agent Systems, ProMAS'08*, volume 5442 of *LNAI*, pp. 72–87. 2008c.

Novák, Peter. Jazzyk*, the project website*. `http://jazzyk.sourceforge.net/`, 2009a.

Novák, Peter. *Probabilistic Behavioural State Machines.* In *Proceedings of The Seventh International Workshop on Programming Multi-Agent Systems ProMAS 2009, AAMAS 2009 co-located workshop*, pp. 103–115. 2009b.

Novák, Peter and Dix, Jürgen. *Modular BDI architecture.* In Nakashima et al. (2006), pp. 1009–1015.

Novák, Peter and Dix, Jürgen. *Adding structure to agent programming languages.* In Mehdi Dastani; Amal El Fallah-Seghrouchni; Alessandro Ricci and Michael Winikoff (eds.), *Proceedings of Fifth international Workshop on Programming Multi-Agent Systems, ProMAS'07*, volume 4908 of *LNAI*, pp. 140–155. Springer Verlag, 2007.

Novák, Peter and Jamroga, Wojciech. *Code patterns for agent-oriented programming.* In *Proceedings of The Eighth International Conference on Autonomous Agents and Multi-Agent Systems, AAMAS 2009*. 2009.

Novák, Peter and Köster, Michael. *Designing goal-oriented reactive behaviours.* In *Proceedings of the 6th International Cognitive Robotics Workshop, CogRob 2008, ECCAI co-located workshop, July 21-22 in Patras, Greece*, pp. 24–31. 2008.

Object Management Group. *Mobile Agent Facility Specification, version 1.0*. 2000. Available from `http://www.omg.org/technology/documents/formal/mobile_agent_facility.htm`.

Padovani, Luca and Provetti, Alessandro. *Qsmodels: ASP planning in interactive gaming environment.* In José Júlio Alferes and João Alexandre Leite (eds.), *JELIA*, volume 3229 of *Lecture Notes in Computer Science*, pp. 689–692. Springer, 2004. ISBN 3-540-23242-7.

Plotkin, Gordon D. *A Structural Approach to Operational Semantics.* Technical Report DAIMI FN-19, University of Aarhus, 1981.

Pnueli, Amir. *The temporal logic of programs.* In *Proceedings of FOCS*, pp. 46–57. 1977.

Pokahr, Alexander; Braubach, Lars and Lamersdorf, Winfried. *Jadex: A BDI Reasoning Engine*, chapter 6, pp. 149–174. Volume 15 of Bordini et al. (2005a), 2005.

Rabin, S. *AI Game Programming Wisdom 2.* Charles River Media, 2004.

Rao, Anand S. *AgentSpeak(L): BDI Agents Speak Out in a Logical Computable Language.* In Walter Van de Velde and John W. Perram (eds.), *MAAMAW*, volume 1038 of *Lecture Notes in Computer Science*, pp. 42–55. Springer, 1996. ISBN 3-540-60852-4.

Rao, Anand S. and Georgeff, Michael P. *Modeling Rational Agents within a BDI-Architecture.* In *KR*, pp. 473–484. 1991.

Rao, Anand S. and Georgeff, Michael P. *An Abstract Architecture for Rational Agents.* In *KR*, pp. 439–449. 1992.

van Riemsdijk, Birna. *Cognitive Agent Programming: A Semantic Approach.* Ph.D. thesis, Utrecht University, 2006.

van Riemsdijk, M. Birna; de Boer, Frank S. and Meyer, John-Jules Ch. *Dynamic logic for plan revision in intelligent agents.* In João Alexandre Leite and Paolo Torroni (eds.), *CLIMA V*, volume 3487 of *Lecture Notes in Computer Science*, pp. 16–32. Springer, 2004. ISBN 3-540-28060-X.

van Riemsdijk, M. Birna; de Boer, Frank S. and Meyer, John-Jules Ch. *Dynamic logic for plan revision in agent programming. J. Log. Comput.*, volume 16(3):pp. 375–402, 2006a.

van Riemsdijk, M. Birna; Dastani, Mehdi; Meyer, John-Jules Ch. and de Boer, Frank S. *Goal-oriented modularity in agent programming.* In Nakashima et al. (2006), pp. 1271–1278.

van Riemsdijk, M. Birna; Dastani, Mehdi and Winikoff, Michael. *Goals in agent systems: a unifying framework.* In Lin Padgham; David C. Parkes; Jörg Müller and Simon Parsons (eds.), *AAMAS (2)*, pp. 713–720. IFAAMAS, 2008. ISBN 978-0-9817381-1-6.

Rosner, Roni and Pnueli, Amir. *A choppy logic.* In *LICS*, pp. 306–313. IEEE Computer Society, 1986.

Russell, Stuart J. and Norvig, Peter. *Artificial Intelligence: A Modern Approach (2nd Edition).* Prentice Hall, 2002. ISBN 0137903952.

Sardiña, Sebastian; de Silva, Lavindra and Padgham, Lin. *Hierarchical planning in BDI agent programming languages: a formal approach.* In Nakashima et al. (2006), pp. 1001–1008.

Schoppers, Marcel. *Universal plans for reactive robots in unpredictable environments.* In *IJCAI*, pp. 1039–1046. 1987.

Shapiro, Ehud and Sterling, Leon. *The Art of Prolog: Advanced Programming Techniques.* The MIT Press, 1994. ISBN 0262691639.

Shehory, Onn and Sycara, Katia P. *The RETSINA Communicator.* In *Agents*, pp. 199–200. 2000.

Shoham, Yoav. *Agent-oriented programming. Artif. Intell.*, volume 60(1):pp. 51–92, 1993.

SRI International. *The Open Agent Architecture^{TM}, version 2.3.2.* `http://www.ai.sri.com/oaa/`, 2007.

Subrahmanian, V. S.; Bonatti, Piero A.; Dix, Jürgen; Eiter, Thomas; Kraus, Sarit; Ozcan, Fatma and Ross, Robert. *Heterogenous Active Agents.* MIT Press, 2000. ISBN 0-262-19436-8.

Sun Microsystems Inc. *Java^{TM} Platform, Standard Edition 6.* `http://java.sun.com/`, 2006.

Sussman, Gerald J. and Steele Jr., Guy L. *Scheme: An interpreter for extended lambda calculus. Higher-Order and Symbolic Computation*, volume 11(4):pp. 405–439, 1998.

Sycara, Katia P.; Paolucci, Massimo; Velsen, Martin Van and Giampapa, Joseph A. *The RETSINA MAS infrastructure*. *Autonomous Agents and Multi-Agent Systems*, volume 7(1-2):pp. 29–48, 2003.

Syrjänen, T. *Implementation of local grounding for logic programs with stable model semantics*. Technical Report B18, Digital Systems Laboratory, Helsinki University of Technology, 1998.

Syrjänen, Tommi and Niemelä, Ilkka. *The Smodels System*. In Thomas Eiter; Wolfgang Faber and Miroslaw Truszczynski (eds.), *LPNMR*, volume 2173 of *Lecture Notes in Computer Science*, pp. 434–438. Springer, 2001. ISBN 3-540-42593-4.

Telecom Italia Lab and The JADE Board. *Java Agent DEvelopment Framework*. `http://jade.tilab.com/`, 2009.

W3C. *Extensible Markup Language (XML) 1.0*. `http://www.w3.org/TR/REC-xml/`, 2008a.

W3C. *Extensible Markup Language (XML) 1.0, Section 6, Notation*. `http://www.w3.org/TR/REC-xml/#sec-notation`, 2008b.

Wielemaker, Jan. *An overview of the SWI-Prolog programming environment*. In Fred Mesnard and Alexander Serebenik (eds.), *Proceedings of the 13th International Workshop on Logic Programming Environments*, pp. 1–16. Katholieke Universiteit Leuven, Heverlee, Belgium, 2003. CW 371.

Winikoff, Michael. *An AgentSpeak Meta-interpreter and Its Applications*. In Rafael H. Bordini; Mehdi Dastani; Jürgen Dix and Amal El Fallah-Seghrouchni (eds.), *PROMAS*, volume 3862 of *Lecture Notes in Computer Science*, pp. 123–138. Springer, 2005a. ISBN 3-540-32616-2.

Winikoff, Michael. *JACK^{TM} Intelligent Agents: An Industrial Strength Platform*, chapter 7, pp. 175–193. Volume 15 of Bordini et al. (2005a), 2005b.

Wooldridge, Michael. *Reasoning about rational agents*. MIT Press, London, 2000.

Yi, Weiyu. *JzMASSim: Jazzyk plug-in for communication with the MASSim server*. `http://jazzyk.sourceforge.net/projects/modules/jzMASSim.html`, 2009.

Abstract

One of the original long-term aims of Artificial Intelligence is to build intelligent entities, i.e., such which are able to function and act in a manner similar to human beings. Intelligent agents are autonomous entities, which are also assumed to be proactive, reactive, as well as socially able. To enable development of such systems, in this thesis, I am focusing on methods supporting design and implementation of cognitive agents, a specific subtype of agent systems which internally construct a model of their environment, themselves or their peers and use it as a basis for decisions about their future actions.

Since different application domains require different knowledge representation approaches, one of the outstanding problems of the field is the integration of heterogeneous knowledge representation approaches into a single agent system. I propose a framework of *Behavioural State Machines* (*BSM*), drawing a strict distinction between the representational and behavioural layer of an agent system. While keeping the former as abstract and open as possible, its focus lies on enabling flexible specification of agent's behaviours and the overall agent reasoning model, i.e., specification of its lifecycle. Thus, the *BSM* framework features a plug-in architecture and enables transparent integration of heterogeneous knowledge representation technologies into a single agent system.

Provided a suitable programming framework, the next problem is the pragmatics of its use. Formal specification and semantic characterization of subprograms is essential means for getting a grip on composition of complex programs from lower-level modules. I introduce *Dynamic Computation Tree Logic DCTL**, a novel hybrid logic marrying branching time temporal logic *CTL** with features inspired by *Dynamic Logic*. It allows semantic characterization of *BSM* subprograms and in turn enables construction of a sequel of formally specified design patterns implementing useful agent-oriented concepts, such as *achievement* and *maintenance goals.* The thesis culminates in a proposal of *commitment-oriented programming*, an abstract methodology for programming cognitive agents generalizing the approach taken by construction of the design patterns.

To demonstrate the feasibility and usefulness of the proposed approach to agent-oriented programming I describe three case-studies implemented in *Jazzyk*, a concrete implemented programming language for the *BSM* framework. Finally, to show the robustness of the *BSM* framework, I describe its straightforward probabilistic extension and present a formal study of its relationship to an agent-oriented programming language *GOAL*. I show that *BSM* can be seen as an intermediary language into which programs of other agent programming languages can be compiled.

Zusammenfassung

Eines der Langzeitziele der Künstlichen Intelligenz (KI) ist es, intelligente Entitäten, also solche, die ähnlich wie Menschen funktionieren und handeln, zu konstruieren. Intelligente Agenten sind autonome Entitäten, die zusätzlich proaktiv und reaktiv sind, sowie soziale Fähigkeiten besitzen. Der Fokus dieser Dissertation liegt auf Methoden, die die Entwicklung von kognitiven Agenten unterstützen; Agenten einer speziellen Unterklasse, die intern ein Modell über ihre Umgebung, sich selbst und anderen Agenten konstruieren und es als Basis für ihre Entscheidungen über ihre zukünftigen Aktionen benutzen.

Da verschiedene Anwendungsdomänen unterschiedliche Wissensrepräsentationstechniken (KR) benötigen, ist eines der ausstehenden Probleme der KI die Integration von heterogenen KR-Techniken in ein Agentensystem. Ich stelle das Framework der *Behavioural State Machines* (*BSM*) vor, welches eine strikte Trennung zwischen der Repräsentations- und der Verhaltensschicht eines Agenten vorsieht. Während die KR-Schicht abstrakt gehalten ist, liegt der Fokus des Frameworks auf der flexiblen Spezifikation des Agentenverhaltens und seinem globalen Deliberationsmodell, d.h., die Spezifikation seines Lebenszyklus. Das *BSM* Framework weist eine Plug-in-Architektur auf und ermöglicht somit die Integration von heterogenen KR-Techniken in einem Agenten.

Neben dem Programmierframework ist auch die Frage dessen richtiger Nützung entscheidend. Die formale Spezifikation und die semantische Charakterisierung von Unterprogrammen eröffnet die Einsicht in die Komposition con komplexen Programmen aus einfacheren Modulen. Ich stelle eine neue hybride Logik, die *Dynamic Computation Tree Logic DCTL** vor, welche eine Erweiterung der *Branching Time Temporal Logic CTL** um Eigenschaften von *Dynamic Logic* ist. Sie erlaubt die semantische Charakterisierung von *BSM* Unterprogrammen und ermöglicht die Konstruktion einer Sequenz formal spezifizierter Entwurfsmuster, die nützliche Agenten-orienterte Konzepte, wie zum Beispiel *achievement goal* und *maintenance goal* implementieren. Diese Arbeit mündet in dem Vorschlag des *Commitment-orientiertem Programmieren*s, eine abstrakte Methodik für die Programmierung von kognitiven Agenten, die auf den Entwurfsmustern basiert.

Um die Machbarkeit und Nützlichkeit des vorgeschlagenen Ansatzes zu zeigen, beschreibe ich drei Fallstudien, die in der Programmiersprache *Jazzyk*, eine konkrete Instanz des *BSM* Frameworks, implementiert sind. Um die Robustheit des Frameworks zu demonstrieren, beschreibe ich schließlich eine probabilistische Erweiterung und präsentiere eine Studie über die Beziehung von *BSM* zu der Programmiersprache *GOAL*. Sie zeigt, dass *BSM* als eine Zwischencode-Sprache gesehen werden kann, in die Programme anderer Agenten-orientierten Programmiersprachen übersetzt werden können.

Résumé

Personal trivia

Date and place of birth:	March 23th, 1979, Ružomberok, Slovak Republic
Citizenship/Nationality:	Slovak Republic/Slovak

Education

2004–2009	*Ph.D. candidate* Department of Informatics, Clausthal-Zellerfeld, Clausthal University of Technology, Germany
1997–2002	*computer science/Mgr. (MSc. equiv.)* major: artificial intelligence minor: distributed and parallel computing Faculty of Mathematics, Physics and Informatics, Comenius University in Bratislava, Slovakia

Professional experience

2004–2009	*research assistant* Department of Informatics, Clausthal University of Technology, Germany
2004–2009	*external collaborator* Department of Applied Informatics, Faculty of Mathematics, Physics and Informatics, Comenius University in Bratislava, Slovakia
2004	*freelance software engineer* self employed, Nitra, Slovakia/Lausanne, Switzerland
1999-2003	*team leader/programmer/analyst* Whitestein Technologies s.r.o., Bratislava, Slovakia
1998–1999	*programmer/analyst* Microstep s.r.o., Bratislava, Slovakia
1996–1997	*programmer/analyst* DIO s.r.o., Nitra, Slovakia