Czech Technical Univiersity in Prague
Faculty of Electrical Engineering

DIPLOMA THESIS

Study Of Expressivity Between Agent-Oriented Programming Languages AgentSpeak(L) And Behavioural State Machines

Prague, 2012 **Author: Bc. Marek Juras**

Prohlášení

Prohlašuji, že jsem svou diplomovou práci vypracoval samostatně a použil jsem pouze podklady (literaturu, projekty, SW atd.) uvedené v přiloženém seznamu.

V Praze dne 11. 5. 2012

podpis

Acknowledgement

First of all, I would like to thank my supervisor, Dr. Peter Novák, for his constructive comments, advices and neverending patience during the developement of the thesis. I'm also very grateful to my family and friends that supported my effort.

Abstrakt

Hlavním cílem této diplomové práce je provedení formální analýzy expresivity mezi dvěma agentově orientovanými jazyky: AgentSpeak(L) a Behavioural state machines (BSM). Oba jazyky využívají odlišné filosofie. První z nich pracuje na principu paradigmatu Belief-Desire-Intention (BDI), zatímco ten druhý poskytuje vysokou úroveň modularity a nezavazuje k žádné konkrétní technologii.

V této práci navrhuji kompilovací funkci schopnou překladu syntaktických struktur a sémantických pravidel jazyka AgentSpeak(L) do formalismu BSM tak, že agent a jeho přeložený ekvivalent se mohou vzájemně simulovat.

Důkaz založený na pojmu translační bisimulace ukazuje, že BSM má alespoň takovou vyjadřovací sílu jako AgentSpeak(L).

Abstract

The main aim of this diploma thesis is to perform a formal analysis of expressivity between two agent-oriented programming languages: AgentSpeak(L) and Behavioural state machines (BSM). Both languages incorporate different philosophies. The former works on the basis of the Belief-Desire-Intention paradigm, while the latter provides high level of modularity, and does not commit to any concrete technology.

In this work, I propose the compiling function capable of translating the syntactic structures and semantic rules of AgentSpeak(L) into the formalism of BSM such that an agent and it's translated equivalent can simulate each other.

The proof based on the notion of translation bisimulation shows that BSM has at least the same expressive power as AgentSpeak(L).

Czech Technical University in Prague
Faculty of Electrical Engineering

Department of Cybernetics

DIPLOMA THESIS ASSIGNMENT

Student: **Bc. Marek Juras**

Study programme: Open Informatics

Specialisation: Artificial Intelligence

Title of Diploma Thesis: **Study of expressivity between agent-oriented programming languages AgentSpeak(L) and Behavioural State Machines**

Guidelines:

The aim of the proposed thesis is to perform a formal comparison of expresivity of AgentSpeak(L) and the BSM framework. The working hypothesis is that AgentSpeak(L) programs can be compiled (translated) to semantically equivalent BSM programs. In particular, the thesis should include:

1. Problem formulation
2. AgentSpeak(L) vs. BSM expressivity analysis
3. Discussion and possible paths towards generalizations of the result w.r.t. other state of the art agent-oriented programming languages (e.g., 2APL)

Bibliography/Sources:

Peter Novák: Jazzyk: A Programming Language for Hybrid Agents with Heterogeneous Knowledge Representations. ProMAS 2008: 72-87

Koen V. Hindriks, Peter Novák: Compiling GOAL Agent Programs into Jazzyk Behavioural State Machines. MATES 2008: 86-98

Anand S. Rao. AgentSpeak(L): BDI Agents Speak Out in a Logical Computable, Language. In Walter Van de Velde and John W. Perram, editors, MAAMAW, volume 1038 of Lecture Notes in Computer Science, pages 42–55. Springer, 1996.

Rafael H. Bordini, Jomi Fred Hübner, and Michael Wooldridge. Programming, Multi-agent Systems in AgentSpeak Using Jason. Wiley Series in Agent Technology. Wiley-Blackwell, 2007.

Diploma Thesis Supervisor: Dr. Peter, Novák rer. nat.

Valid until: the end of the summer semester 2011/2012

prof. Ing. Vladimír Mařík, DrSc.
Head of Department

L.S.

prof. Ing. Boris Šimák, CSc.
Dean

Prague, March 25, 2011

Contents

List of Figures

Chapter 1

Introduction

The main result of this thesis is a formulation of a formal proof that Behavioural state machines (BSM) has at least the same expressive power as AgentSpeak(L). The proof is based on concepts of computation and observation, and utilizes the translation bisimulation as the underlying comparing method.

1.1 Motivation

In the field of agent-oriented programming, the mutual expressivity analysis between particular languages can serve as a benchmarking feature. We can measure the amount of code, necessary for implementing a given problem in particular programming language and study its efficiency, clarity and so on.

It is shown, in [10], that the language GOAL[1] can be compiled relatively easy to BSM, and furthermore, it is shown that GOAL agent language is not committed to any particular knowledge representation technology. Hindriks et al [3] proposed a formal embedding of AgentSpeak(L) in 3APL. Also this comparison yielded some new insights into the features of the agent languages. One of the results, proposed in [3], was that the notion of events does not increase the expressive power of AgentSpeak(L).

[15] introduces BSM code patterns and uses the temporal logic to reason about the properties of its executions. Since this thesis introduces a relation between BSM and AgentSpeak(L), the properties might be transferable from the former to the latter.

1.2 Structure of the thesis

Chapter 2 introduces the syntax and the semantics of the first language, AgentSpeak(L) [6]. There are certain omissions and improvements that make the language usable for our purposes. All the changes are discussed in corresponding sections. Then I performed a detailed analysis of the interpreter incorporated in AgentSpeak(L) that helped to clarify the usage of language's semantics. In chapter 3, the latter language is introduced. It is called Behavioural state machines (BSM). It is a simple and flexible formalism, and it's expressivity relative to AgentSpeak(L) is examined in this thesis. The chapter 4 proposes a formal mechanism for the languages' expressivity comparison. It involves the notion of bisimulation. Chapter 5 expresses my research efforts aimed towards the embedding of AgentSpeak(L) into BSM. Since the language BSM is highly modular, first I created the basic building blocks (KR modules), and then sticked these modules together using the mental state transformer claiming that such BSM compilation can emulate the behaviour of the AgentSpeak(L) agent. An important result of the chapter involves the specification of translation function that maps every AgentSpeak(L) to its BSM counterpart. This chapter ends with a theorem (and its proof) stating that the language of BSM has at least the same expressive power as AgentSpeak(L). Last chapter 6 contains a discussion and a conclusion of the thesis.

Chapter 2

AgentSpeak (L)

This chapter first informally introduces origins and properties of AgentSpeak(L). Then it provides important notions of syntax, and a detailed description of its internal mechanisms, i.e., its semantics and its interpreter.

2.1 Overview

The language AgentSpeak(L) [6] is a theoretical agent-oriented programming language based on the BDI paradigm [7]. It was created by Anand Rao in 1996 and since then, many papers were proposed discussing and improving its properties. There are several versions of AgentSpeak. AgentSpeak(L) is its very first incarnation. For example, in [17], there is an extension of AgentSpeak(L) (called AgentSpeak(XL)), which defines additional semantics and introduces communication issues between agents. In [11], there is another extension of AgentSpeak(L), called AgentSpeak(RT), involving deadlines and priorities to figure out real-time applications.

AgentSpeak(L) became quite popular thanks to Jason [16] - a Java based interpreter of another extension of AgentSpeak(L). However, in this thesis, I'm talking about the original version AgentSpeak(L) as it was proposed in [6].

An AgentSpeak(L) agent consists of beliefs, goals, events, intentions and plan rules. Each agent is equipped with a set of actions to be able to change the environment it is currently in. Actions are executed as a result of adopted plans. Adopted plans are called intentions.

Beliefs play a key role in the agent architecture. The set of beliefs represents the

situation the agent believes it is in. It is important to point out that the information stored in the belief base may not correspond to the current state of the environment. It is a common situation that an agent (represented by a robot capable of moving and sensing the environment) gathers information through sensors of limited range. Once the agent senses its local neighbourhood, updates its beliefs, and then moves out, it might be no longer capable of checking whether the information is still valid unless it returns back to perform checking sensing. Thus, beliefs may not reflect the actual state of the environment but represent the only information the agent has about the state of the outer world. Note that this is a typical property of beliefs.

There are two types of goals in AgentSpeak(L): achievement goals and test goals. These goals and actions create building blocks for plan bodies. Achievement goals represent the states of affairs the agent wants to achieve. For each achievement goal, there is a plan (designed by an agent designer) to reach the goal, i.e., the set of actions or another goals to be performed. An attempt to accomplish an achievement goal is initiated by a search in the plan library for plan rules that provide appropriate means to succeed.

Plan rules reside in the plan library. Such rules represent the know-how of the agent. Each plan to be executed must be adopted by the internal commitment mechanism. Such mechanism involves retrieval in the plan library on the basis of triggering event, its adopting, partial instantiation, and inserting at the top of particular intention stack.

An agent acts on the basis of its intentions. An agent executes plans located at the top of selected intention. In the case it finishes executing, the execution continues with the next plan on the stack (if exists such a plan).

In further sections the syntax and semantics of AgentSpeak(L) is specified. There are certain omissions and deviations from [6]. These discrepancies are commented below the relevant definitions in following sections.

2.2 Syntax

This section specifies and formally introduces notions, that were informally presented in the overview of AgentSpeak(L). Following definitions were adopted from [6], but their final form (presented in the further text) is significantly influenced by AgentSpeak(L) revision and partial reformulation in [3].

Definition 2.1: (signature for AgentSpeak(L)) A *signature* for AgentSpeak(L) is a

tuple $\langle \mathsf{Pred}, \mathsf{Func}, \mathsf{Cons}, \mathsf{Act} \rangle$, where:

- Pred is a set of *predicate symbols*,
- Func is a set of *function symbols*,
- Cons is a set of *constant symbols*,
- Act is a set of *action symbols*.

Assuming that Pred, Func, Cons and Act are disjoint sets.

Definition 2.2: (terms) Let Var be as set of *variables*. The set of *terms* T is inductively defined by:

- $\mathsf{Var} \subseteq \mathsf{T}$,
- $\mathsf{Cons} \subseteq \mathsf{T}$ is a set of *constant symbols*,
- if $f \in \mathsf{Func}$ of arity n and $t_1, \ldots, t_n \in \mathsf{T}$, then $f(t_1, \ldots, t_n) \in \mathsf{T}$.

Definition 2.3: (belief atoms, literals) The set of *atoms* At and *literals* Lit are defined by:

- $\mathsf{At} = \left\{ P(t_1, \ldots, t_n) \;\middle|\; P \in \mathsf{Pred} \text{ of arity } n, \text{ and } t_1, \ldots, t_n \in \mathsf{T} \right\}$,
- $\mathsf{Lit} = \mathsf{At} \cup \neg\mathsf{At}$.

A ground atom is also called a *base belief*.

Definition 2.4: (actions) The set of *actions* A is defined by:

- if $a \in \mathsf{Act}$ of arity n and $t_1, \ldots, t_n \in \mathsf{T}$, then $a(t_1, \ldots, t_n) \in \mathsf{A}$.

The actions of an agent are the basic means of the agent to change its environment. The set of these actions can be viewed as specifying the capabilities of the agent.

Two different types of goals are distinguished in AgentSpeak(L). First, the achievement goal $!\phi$ denotes the fact that an agent has a goal to establish a state of affairs where ϕ is the case. The second type, test goal $?\phi$ is used for a belief base inspection.

Definition 2.5: (achievement goals, test goals) The set of AgentSpeak(L) *goals* G is defined by:

– if $\phi \in \mathsf{At}$, then $!\phi \in \mathsf{G}$,

– if $\phi \in \mathsf{At}$, then $?\phi \in \mathsf{G}$,

A goal $!\phi \in \mathsf{G}$ is called an *achievement goal*. An achievement goal $!\phi$ expresses that the agents has a goal to achieve a state of affairs where ϕ is the case. A goal $?\phi$ is called a *test goal*. A test goal $?\phi$ is a test on the belief base to check whether ϕ is or is not believed.

In [6], the authors define six types of *triggering events*. A triggering event plays a key role in plan adoption. All the plan rules in the plan library are indexed by those triggering events. The idea is that a triggering event $+!\phi$ adds an achievement goal. Conversely, a triggering event $-!\phi$ deletes an achievement goal. The same logic is used for addition and deletion of test goals and belief literals denoted as $+?\phi$, $-?\phi$, $+\phi$ and $-\phi$ respectively. However, [6] specifies the semantics for addition of achievement goal $+!\phi$. Remaining triggering events are not specified in [6] and that's why they are not mentioned in this thesis.

Definition 2.6: (triggering events) The set of *triggering events* TrEv is defined in a following way:

– if $!\phi \in \mathsf{G}$, then $+!\phi \in \mathsf{TrEv}$.

Plan rules in AgentSpeak(L) provide the means for bringing about an achievement goal. Plan rules are of the form: $e : b_1 \wedge \cdots \wedge b_m \leftarrow h_1; \ldots; h_n$. e is a triggering event and it indicates for which achievement goal the rule provides the plan. e is also called the *head* of the plan. The following part of a plan $b_1 \wedge \cdots \wedge b_m$ is the *context* of the plan and specifies the condition that must be satisfied in order to make the plan applicable. The rest of the plan $h_1; \ldots; h_n$ is called the *body*.

Definition 2.7: (plan rules) The set of *plan rules* $\mathsf{PlanRule}$ is defined as follows:

– if $e \in \mathsf{TrEv}$, $b_1, \ldots, b_m$ are belief literals, and $h_1, \ldots, h_n \in (\mathsf{G} \cup \mathsf{A})$, then $e : b_1 \wedge \cdots \wedge b_m \leftarrow h_1; \ldots; h_n \in \mathsf{PlanRule}$.

An intention in the context of AgentSpeak(L) agents is a stack of partially instantiated plans that were adopted by a certain commitment strategy. The set of intentions work as a storage of such plans. Each intention is created on the basis of a triggering event. Each layer of the intention stack contains a plan that handles an achievement goal located one entry below in the stack (if there is such an entry).

Definition 2.8: (intentions) The set of *intentions* Int is defined as follows:

– if $\rho_1, \ldots, \rho_z \in \mathsf{PlanRule}$, then $[\rho_1 \ddagger \ldots \ddagger \rho_z] \in \mathsf{Int}$.

Last notion to define is a notion of *events*. In fact, an event is a kind of a middle step between the execution of an achievement goal and an intention creation. When the achievement goal execution happens, an event is created by compounding an intention and a triggering event.

Definition 2.9: (events) The set of *events* Ev is defined as follows:

– if $e \in \mathsf{TrEv}$ and $i \in \mathsf{Int}$, then $\langle e, i \rangle \in \mathsf{Ev}$.

Events are necessary for the creation of intentions. In [6], events are generated when an agent registers changes in the environment or an achievement goal is about to performed. The latter case creates an internal event and only this kind of event in considered that may happen. The former case, caused by changes in the environment, creates external events. External events are not used in this text since the semantics of their creation is not specified, and for the purposes of the formal proof they are not necessary. We assume that simulated agent is a kind of a blind agent, i.e., it does not sense the environment. The notion of an environment is not even specified. A blind agent does not act upon the basis of outer (external) events but only the internal ones.

Initially, we assume that the set of events is empty and all the intention stacks are of the form $[+!\texttt{true} : \texttt{true} \leftarrow h_1; \ldots; h_n]$. Informally, it means that no events have been generated when the execution of the agent is started. On the other hand, an agent may have adopt some plans to execute. This is reflected in the form of initial intentions, i.e., the sequence of goals or actions $h_1; \ldots; h_n$ states that an agent may have adopted a plan to act upon initially. It is important that all the initial intention stacks have one layer only.

Definition 2.10: (AgentSpeak(L) agent) An AgentSpeak(L) agent is a tuple $\langle E, B, I, P \rangle$, where:

– if $E \subseteq \mathsf{Ev}$ is a set of events,

– if $B \subseteq \mathsf{Lit}$ is a set of ground belief literals,

– if $I \subseteq \mathsf{Int}$ is a set of intentions,

– if $P \subseteq \mathsf{PlanRules}$ is a set of plan rules

such that $E = \{\}$ and all $i \in I$ are of the form $[+!true : true \leftarrow h_1; \ldots; h_n]$.

In the original paper [6] on AgentSpeak(L), there are definitions of three functions for selecting events ($\mathcal{S}_\mathcal{E}$), for applicable plans also called options ($\mathcal{S}_\mathcal{O}$), and finally for intentions ($\mathcal{S}_\mathcal{I}$). As argued in [3], these functions should be incorporated in the interpreter implementing the semantics of AgentSpeak(L). For this reason, I do not mention these functions in the agent's semantics definition.

2.3 Semantics

The semantics of AgentSpeak(L) is defined through evolutions of *BDI configurations*. Originally (in [6]), the configuration contains a set of actions A to keep track all the actions used during the live of an agent. In that paper, the set A is a part of the BDI configuration. As mentioned in [3], storing of performed actions into a set A is not necessary, since the set A plays no role in the agent's semantics. Actions that are about to be performed should be queued, and such a queue should be an input for the agent's actuators capable of direct affecting the environment. New sensing of the environment then leads to an appropriate change of the belief base. All the effects of an action performance is reflected and projected to agent's belief base by function $\mathcal{T}$ that specifies the update semantics of all the actions.

The operational semantics is defined

Definition 2.11: (BDI configuration) A *BDI configuration* is a tuple $\langle C_E, C_B, C_I \rangle$, where:

- if $C_E \subseteq \mathsf{Ev}$ is a set of events,
- if $C_B \subseteq \mathsf{Lit}$ is a set of ground belief literals,
- if $C_I \subseteq \mathsf{Int}$ is a set of intentions

BDI configuration contains those components of an agent that do change during the computation. Other components, like the plan library is not reflected in the BDI configuration since it is assumed, that the set of plans is constant. All the proof rules in the proof system are defined by the form:

$$\frac{\langle C_E, C_B, C_I \rangle}{\langle C'_E, C'_B, C'_I \rangle} \qquad conditions,$$

where both, the premise (above the line) and the conclusion (below the line) of the proof rule, are configurations.

Moreover, the semantics of the actions that an agent can perform is defined through the semantic function $\mathcal{T} : \mathsf{Act} \times \wp(\mathsf{Lit}) \to \wp(\mathsf{Lit})$. Note, that the set Act represents the set of all the action symbols, and $\wp(\mathsf{Lit})$ denotes the powerset of the set of (grounded) literals. In the following text, the function $\mathcal{T}$ appears in the proof rule that deals with the update of belief base.

There are four proof rules in the AgentSpeak(L) proof system. The first of them (**IntendMeans**) deals with the handling of (internal) events. As mentioned above, events are generated exclusively by achievement goals. Such goals together with their intention are stored in the event. A proper handling plan for the achievement goal needs to be found in the plan library, and then put at the top of the intention. The event itself is removed, and the intention is inserted to the set of intentions.

Definition 2.12: (proof rule IntendMeans)

$$\frac{\Big\langle \big\{ \ldots, \langle +!p(\mathbf{t}), [\rho_1 \ddagger \ldots \ddagger \rho_z] \rangle, \ldots \big\}, C_B, C_I \Big\rangle}{\Big\langle \big\{ \ldots \big\}, C_B, C_I \cup \big\{ [\rho_1 \ddagger \ldots \ddagger \rho_z \ddagger \rho] \eta\theta \big\} \Big\rangle}$$

where:

- $\rho_z = e : \phi \leftarrow !p(\mathbf{t}); g_2; \ldots; g_l$,
- $\rho = +!p(\mathbf{s}) : \psi \leftarrow h_1; \ldots; h_n$ is a variant of a plan rule in P, such that the variables that occur in ρ do not occur in either the event base or the intention base,
- η is the most general unifier of $p(\mathbf{t})$ and $p(\mathbf{s})$, such that $p(\mathbf{t})\eta = p(\mathbf{s})\eta$,
- $C_B \models \psi\eta\theta$.

The proof rule states, that a triggering event $+!p(\mathbf{t})$ of the selected event is unified with the triggering event of a plan ρ through the substitution η. This substitution is then used to instantiate some of the variables in the context of the plan. It must hold that $C_B \models \psi\eta\theta$. Substitution θ is used for instantiation of remaining variables in the context ψ, so ψ is fully grounded. The plan ρ, which is then pushed to the top of the intention stack. This updated stack is then inserted to the set of intentions C_I. Compound substitution $\eta\theta$ is then applied to the whole intention.

There is a significant difference between the proof rule defined here and the one defined in [6]. Here, a *variant* of the plan rule ρ is required to be picked up from the plan library

P, i.e., all the variables in the plan ρ must be renamed not to collide with names of the variables already included in the intention base or the event base. In [3], it is shown that such variable renaming is necessary to avoid unwanted implicit bindings of variables. The method of renaming assures that there won't be any problematic variable bindings.

Another difference between the original paper and our formalism concerns the application of the compound substitution $\eta\theta$. In [6], the substitution is applied only for instantiating of the plan. Note, that here it is applied to the whole intention. Thanks to the mechanism of variable renaming, such application of the substitution is possible.

IntendMeans expresses the handling of internal events. In [6] there is a proof rule handling with external events. External event is an event with an intention part of the form: $[true : true \leftarrow true]$. Since the semantics of the creation of such events is not provided in [6], the proof rule is not mentioned here.

The second proof rule, **ExecAch**, deals with generation of (internal) events. Internal events are emitted on the basis of the performance of an achievement goal. The event is then created from such a goal and the whole intention the achievement goal comes from.

Definition 2.13: (proof rule ExecAch)

$$\frac{\Big\langle C_E, C_B, \{\ldots, j, \ldots\}\Big\rangle}{\Big\langle C_E \cup \{\langle +!p(\mathbf{t}), j\rangle\}, C_B, \{\ldots\}\Big\rangle}$$

where:

– $j = [\rho_1 \ddagger \ldots \ddagger \rho_{z-1} \ddagger (e : \phi \leftarrow !p(\mathbf{t}); h_2; \ldots; h_n)]$.

There is a little inaccuracy in the formulation of the proof rule **ExecAch** in [6]. Here, the intention j is removed. It does not make too much sense to generate an event from the intention j and not to remove it from the intention set (which is done in [6]). The intention j is marked as *suspended* (by putting it into an event). It is returned back after picking an appropriate event. In latter formulations of the operational semantics of AgentSpeak(L) (e.g., in [8] or [16]), suspended intentions are always removed from the intention base.

The following proof rule **ExecAct** tells how to handle with actions. The action execution is the task for a semantic function $\mathcal{T}$ which specifies the update semantics for the belief base.

Definition 2.14: (proof rule ExecAct)

$$\frac{\Big\langle C_E, C_B, \big\{ \ldots, [\rho_1 \ddagger \ldots \ddagger (e : \phi \leftarrow a(\mathbf{t}); h_2; \ldots; h_n)], \ldots \big\} \Big\rangle}{\Big\langle C_E, C'_B, \big\{ \ldots, [\rho_1 \ddagger \ldots \ddagger (e : \phi \leftarrow h_2; \ldots; h_n)], \ldots \big\} \Big\rangle}$$

where:

– $\mathcal{T}\big(a(\mathbf{t}), C_B\big) = C'_B$.

It states that C_B is updated according to the specification of $\mathcal{T}$ and then the action is removed from the intention.

Finally, the last proof rule **ExecTest** covers the handling of the execution of a test goal. The test goal $?p(\mathbf{t})$ invokes an inspection of belief base. It is checked whether $p(\mathbf{t})$ is a logical consequence of C_B. Substitution θ gathers the new information and instantiate variables in the rest of the whole intention. Original AgentSpeak(L) paper applies the substitution θ only to the top of the intention but because the notion of renaming variables, it is possible to use it as defined below.

Definition 2.15: (proof rule ExecTest)

$$\frac{\Big\langle C_E, C_B, \big\{ \ldots, [\rho_1 \ddagger \ldots \ddagger (e : \phi \leftarrow ?p(\mathbf{t}); h_2; \ldots; h_n)], \ldots \big\} \Big\rangle}{\Big\langle C_E, C_B, \big\{ \ldots, [\rho_1 \ddagger \ldots \ddagger (e : \phi \leftarrow h_2; \ldots; h_n)]\theta, \ldots \big\} \Big\rangle}$$

where:

– $C_B \models p(\mathbf{t})\theta$,

– θ is a ground substitution w.r.t. $p(\mathbf{t})$.

The most significant deviation from the original presentation of AgentSpeak(L) operational semantics lies in the notion of renaming variables of attended plan rules and the application of substitutions. In [3] it is shown that such improvements lead to better generalisation. These changes were also acknowledged by Rao, the author of AgentSpeak(L).

2.4 BDI interpreter

Despite the precise definition of the proof system of AgentSpeak(L), it is not clear when to select each rule. The order of application is not specified implicitly. The specification

is made explicitly by the BDI interpreter. I studied the interpreter in order to determine all possible traces (i.e., all possible sequences of application of the proof rules) that an AgentSpeak(L) agent can generate. I made an analysis of the BDI interpreter (Algorithm 1) to identify the implementation of particular proof rules. As a result of the analysis, two more proof rules (**CleanStackEntry** and **CleanIntSet**) need to be added to the proof system. It will be discussed in more details in further text. The analysis also determines all possible courses of applications of the proof rules.

See Algorithm 1. It describes an adaptation of BDI interpreter proposed in [6]. Note that there are some deviation from the original algorithm proposed in the original paper. These deviations are discussed in the further text as well.

The BDI interpreter utilizes several auxiliary functions with following meanings:

- $top(I)$ returns the top of an intention stack I,
- $head(\rho)$ returns the head of an intended plan ρ,
- $body(\rho)$ returns the body of an intended plan ρ,
- $first(s)$ returns the first element of the sequence s,
- $rest(s)$ returns all but the first element of a sequence s,
- $push(i, I)$ takes an intention frame and an intention (i.e., the stack of intention frames) and pushes the intention frame i on to the top of the intention I, and
- $pop(I)$ removes and returns the top of an intention I.

In addition, the triplet of selection functions ($\mathcal{S}_\mathcal{E}$, $\mathcal{S}_\mathcal{I}$, and $\mathcal{S}_\mathcal{O}$) is used. $\mathcal{S}_\mathcal{E}(C_E)$ selects and returns one event from the set of events C_E. Similarly, $\mathcal{S}_\mathcal{I}(C_I)$ selects and returns one intention from the set of intentions C_I. $\mathcal{S}_\mathcal{O}(\mathcal{O}_\epsilon)$ selects and returns one applicable plan (also called *an option*) from the set of options $\mathcal{O}_\epsilon$.

The whole interpreter works in a cycle. The cycle terminates when the set of events C_I gets empty. This is the first deviation. In original paper, the "while" condition checks whether $C_E \neq \emptyset$. But due to the fact that an initialized agent has $C_E = \emptyset$ (see def. 2.10), the agent's program never enters the loop. Furthermore, an empty set of events does not mean that an agent has nothing to do. It might have intentions in C_I waiting for an execution, and simultaneously the C_E should be empty. This would lead to an unexpected termination of an agent's program. In our case, an agent has nothing to do when the set of intentions C_I is empty.

Note that in the AgentSpeak(L) proof system there is no proof rule, that would delete an empty intention. This proof rule (**CleanIntSet**) must be added to the proof system unless the agent's program would not work properly.

Definition 2.16: (proof rule CleanIntSet)

$$\frac{\Big\langle C_E, C_B, \{\ldots, [+!p(\mathbf{t}) : \phi \leftarrow], \ldots\}\Big\rangle}{\Big\langle C_E, C_B, \{\ldots\}\Big\rangle}$$

By inspection of Algorithm 1, a single cycle can be divided into two independent modes. Let's call these modes as *intention update* (lines 2 - 8), and *intention execution*[1] (lines 10 - 25).

The former (*intention update*) selects a single event ϵ from the set of events C_E (line 3) and removes it (line 4). Then, the set of applicable plans $\mathcal{O}_\epsilon$ is created (line 5). All the plans in $\mathcal{O}_\epsilon$ are applicable in a sense of Definition 10 from [6]. Note that an applicable unifier denoted as θ (in line 5) and σ (in line 7) represent the compound substitution $\eta\theta$ from def. 2.12. The event ϵ must be internal since external ones are omitted in this thesis. In line 7, the function $\mathcal{S}_\mathcal{O}$ selects one applicable plan and adds it on to the top of an intention stack i associated with selected event ϵ. This is a specification of behaviour of the proof rule **IntendMeans** (def. 2.12).

The latter (*intention execution*) forms into four cases. All the cases are distinguished on the basis of the item appearing at the first position in the body of the plan that resides on the top of the selected intention $\mathcal{S}_\mathcal{I}(C_I)$. For convenience, let's call this item as an *actual item*. Particular cases are discussed below:

– *case #1* is applied in the situation when an *actual item* is *true*. It occurs when a plan residing on the top of the selected intention is completely executed, i.e., all the actions, test goals and achievement goals have been already performed. See lines 12 and 13. The topmost intention entry of an intention $\mathcal{S}_\mathcal{I}(C_I)$ containing an empty plan is moved to the variable x. Note that the plan (which is now empty) must have been adopted as a response for an internal event issued by an achievement goal that must have been located as a first item of a plan one entry below the empty plan in the intention stack $\mathcal{S}_\mathcal{I}(C_I)$. Such an achievement goal is removed in line 13. In addition, θ instantiates variables in the revealed intention. This additional parameter passing is needed since the way of applying substitutions in [6] is "entry-wise". In this thesis, the substitution is always applied to the whole intention, and

[1] Notions of an *intention update* and *intention execution* were taken from [2]

Algorithm 1 BDI interpreter

1: **while** $C_I \neq \emptyset$ **do**
2: **if** $C_E \neq \emptyset$ **then**
3: $\epsilon = \langle te, i \rangle = \mathcal{S}_{\mathcal{E}}(C_E)$
4: $C_E = C_E \setminus \epsilon$
5: $\mathcal{O}_\epsilon = \{\rho\theta \mid \theta \text{ is an applicable unifier for event } \epsilon \text{ and plan } \rho\}$
6: *// handling of external events is omitted*
7: $push\big(\mathcal{S}_{\mathcal{O}}(\mathcal{O}_\epsilon)\sigma, i\big)$, where σ is an applicable unifier for event ϵ
8: **end if**
9:
10: *// case #1*
11: **case** $first\Big(body\Big(top\big(\mathcal{S}_{\mathcal{I}}(C_I)\big)\Big)\Big) = true$
12: $x = pop\big(\mathcal{S}_{\mathcal{I}}(C_I)\big)$
13: $push\Big(head\Big(top\big(\mathcal{S}_{\mathcal{I}}(C_I)\big)\Big)\theta \leftarrow rest\Big(body\Big(top\big(\mathcal{S}_{\mathcal{I}}(C_I)\big)\Big)\Big)\theta,\ \mathcal{S}_{\mathcal{I}}(C_I)\Big)$,
where θ is the correct answer substitution where θ is an mgu such that
$x\theta = head\Big(top\big(\mathcal{S}_{\mathcal{I}}(C_I)\big)\Big)\theta$
14: *// case #2*
15: **case** $first\Big(body\Big(top\big(\mathcal{S}_{\mathcal{I}}(C_I)\big)\Big)\Big) = !g(\mathbf{t})$
16: $C_E = C_E \cup \big\langle +!g(\mathbf{t}), \mathcal{S}_{\mathcal{I}}(C_I)\big\rangle$
17: *// case #3*
18: **case** $first\Big(body\Big(top\big(\mathcal{S}_{\mathcal{I}}(C_I)\big)\Big)\Big) = ?g(\mathbf{t})$
19: $pop\big(\mathcal{S}_{\mathcal{I}}(C_I)\big)$
20: $push\Big(head\Big(top\big(\mathcal{S}_{\mathcal{I}}(C_I)\big)\Big)\theta \leftarrow rest\Big(body\Big(top\big(\mathcal{S}_{\mathcal{I}}(C_I)\big)\Big)\Big)\theta,\ \mathcal{S}_{\mathcal{I}}(C_I)\Big)$,
where θ is the correct answer substitution
21: *// case #4*
22: **case** $first\Big(body\Big(top\big(\mathcal{S}_{\mathcal{I}}(C_I)\big)\Big)\Big) = a(\mathbf{t})$
23: $pop\big(\mathcal{S}_{\mathcal{I}}(C_I)\big)$
24: $push\Big(head\Big(top\big(\mathcal{S}_{\mathcal{I}}(C_I)\big)\Big) \leftarrow rest\Big(body\Big(top\big(\mathcal{S}_{\mathcal{I}}(C_I)\big)\Big)\Big),\ \mathcal{S}_{\mathcal{I}}(C_I)\Big)$
25: $A = A \cup \{a(\mathbf{t})\}$
26: **end while**

hence no further variable instantiation is required. There is no proof rule in the proof system that clears empty stack entries, but it is important to have it. Hence, the rule **CleanStackEntry** (def. 2.17) is added to the system. In addition, the rule **CleanStackEntry** covers the *case #1*.

- *case #2* tries to handle the execution of an achievement goal (an *actual item* is an achievement goal). In fact, line 16 implements the proof rule (ExecAch) as it is specified in [6], i.e., generates an internal event. The proof rule **ExecAch** (def. 2.13) enriches the original one with a deletion of selected intention $\mathcal{S}_{\mathcal{I}}(C_I)$ which makes it *suspended*, i.e., unable to be selected by $\mathcal{S}_{\mathcal{I}}$ again. The rule **ExecAch** covers the *case #2*.

- *case #3* implements the execution of a test goal $?g(\mathbf{t})$ (an *actual item* is an test goal). In line 20, $?g(\mathbf{t})$ is removed from the beginning of the top-most intention of the stack $\mathcal{S}_{\mathcal{I}}(C_I)$, and the *correct answer substitution* θ is applied to it. The notion of *correct answer substitution* is defined in [6] (Definition 10), and it corresponds to the substitution θ in the definition of the rule **ExecTest** (def. 2.15). The rule **ExecTest** covers the *case #3*.

- *case #4* performs an action $a(\mathbf{t})$ (an *actual item* is an action). Line def. 24 removes the action in the same way as stated in the *case #3*, unless it applies no substitution to the intention entry. Furthermore, the action $a(\mathbf{t})$ is added to the set of actions A (line def. 25) to be performed physically by agent's actuators (this is not specified in the interpreter). As mentioned earlier in this thesis, I don't use the set A. Instead, the semantic function $\mathcal{T}$ is introduced and reflects the changes caused by the action in the belief base. Note that the rule **ExecAct** covers the *case #4*.

Definition 2.17: (proof rule CleanStackEntry)

$$\frac{\Big\langle C_E, C_B, \big\{ \ldots, [\rho_1 \ddagger \ldots \ddagger \rho_z \ddagger +!p(\mathbf{t}) : \phi \leftarrow], \ldots \big\} \Big\rangle}{\Big\langle C_E, C_B, \big\{ \ldots, [\rho_1 \ddagger \cdots \ddagger \rho'_z], \ldots \big\} \Big\rangle}$$

where:

- $\rho_z = e : \psi \leftarrow !p(\mathbf{t}); h_2; \ldots; h_n$,

- $\rho'_z = e : \psi \leftarrow h_2; \ldots; h_n$.

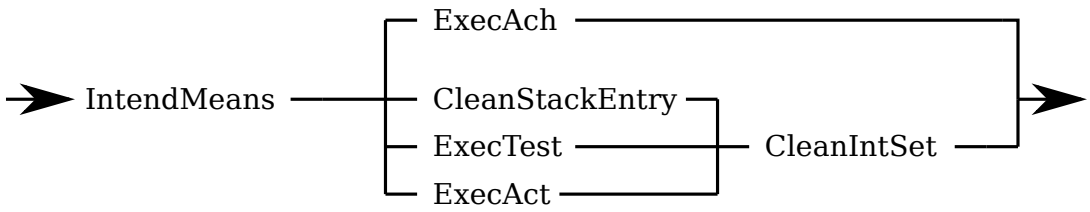

Figure 2.1: This scheme specifies all (four) possible courses of attempts to invoke the proof rules by the interpreter.

The application of the rule **CleanIntSet** is not specified in the interpreter. I think, it is reasonable to apply it behind each rule that removes an item from the plan. There are tree such rules (**CleanStackEntry**, **ExecTest** and **ExecAct**).

At this point, I gathered enough information to derive all possible orders of the proof rules that the interpreter can generate. By considering the analysis, there are four possible scenarios, i.e., four possible courses of attempts to invoke the proof rules by the interpreter. Let's call there courses as (labelled) sequences. Figure 2.1 depicts the scheme of the creation of the sequences. The scheme represents an oriented graph (all edges are oriented from left to right). The arrow on the left is the input and the arrow on the right represents the output. Each walk through this graph (from the input to the output) forms a sequence. All (four) possible sequences ($\bar{\alpha}, \bar{\beta}, \bar{\gamma}$ and $\bar{\delta}$) are specified below.

Definition 2.18: (admissible sequences) Within one cycle of the BDI interpreter, there are four possible sequences of attempts to invoke the proof rules. Such sequences are called *admissible* and are labelled as $\bar{\alpha}, \bar{\beta}, \bar{\gamma}$ or $\bar{\delta}$, and are specified as follows:

- $\bar{\alpha}$ is specified as: $\bar{\alpha}$ = **IntendMeans**, **ExecAch**,
- $\bar{\beta}$ is specified as: $\bar{\beta}$ = **IntendMeans**, **CleanStackEntry**, **CleanIntSet**,
- $\bar{\gamma}$ is specified as: $\bar{\gamma}$ = **IntendMeans**, **ExecTest**, **CleanIntSet**, and
- $\bar{\delta}$ is specified as: $\bar{\delta}$ = **IntendMeans**, **ExecAct**, **CleanIntSet**.

Definition 2.19: (set of proof rule labels) All the labels of the AgentSpeak(L) proof system create a set **ProofRules** = {**IntendMeans**, **ExecAch**, **CleanStackEntry**, **ExecTest**, **ExecAct**, **CleanIntSet**}.

Chapter 3

Behavioural State Machines

This chapter introduces the syntax and sematics of Behavioural state machines (BSM).

3.1 Overview

Behavioural state machines (BSM) is a general purpose computational model, based on the Gurevich's *Abstract State Machines* [9]. An essence of this theoretical framework is a strict distinction between a knowledge representation and an agent's behaviour. Specification of knowledge representation provides only a lightweight interface, which makes this layer simple, flexible and highly modular. This open strategy allows an agent designer to utilize different underlying technologies, that can work together and exploit their particular advantages in various domains of knowledge representation. The behavioural layer creates a roof above those various modules and works as a glue to make these modules work together.

Theoretical fundamentals of BSM as a multi-agent framework were proposed in [13] and [12]. Both of them incorporate BSM as a theoretical basis for a new agent-oriented programming language, called *Jazzyk* [14].

3.2 Syntax

BSM agent utilizes knowledge representation (KR) modules as underlying building blocks as its internal structure. Module's interfaces allow query and update operations. Query

operators allow to evaluate whether a formula ϕ is a logical consequence of the module's state. Each KR module encapsulates a partial state. Collection of all partial states forms *a mental state* of an agent. Transitions between agent's mental states are induced by *mental state transformers (mst)* (atomic updates of mental states), typically denoted by τ. Various types of *mst's* determine the behaviour that an agent can generate.

Definition 3.1: (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 $\models: \mathcal{S} \times \underline{\mathcal{L}} \rightarrow \{\top, \bot\}$,
- a set of update operators $\mathcal{U}$. An update operator $\oplus \in \mathcal{U}$ is a mapping $\oplus : \mathcal{S} \times \underline{\mathcal{L}} \rightarrow \mathcal{S}$.

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

Each KR module can be queried with query formulae as a syntactical means to retrieve information:

Definition 3.2: (query) Let $\mathcal{M}_1, \ldots \mathcal{M}_n$ be a set of compatible KR modules. Query formulae are inductively defined:

- if $\varphi \in \mathcal{L}$, and $\models \in \mathcal{Q}_i$ 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 $\neg\phi_1$.

Definition 3.3: (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:

1. **skip** is a mst (primitive),
2. 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),

3. if ϕ is a query expression, and τ is a mst, then $\phi \longrightarrow \tau$ is a mst as well (conditional),

4. if τ and τ' are mst's, then $\tau|\tau'$ and $\tau \circ \tau'$ are mst's too (choice and sequence).

A standalone mental state transformer is also called an *agent program*. So far, all the necessary terms were defined for proper definition of a BSM agent. BSM agent $\mathcal{A}$ is fully described as $\mathcal{A} = (\mathcal{M}_1, \ldots, \mathcal{M}_n, \tau)$, i.e., a collection of agent KR modules and an associated agent program.

I will use $\mathcal{Q}(\mathcal{A}) = \bigcup_{i=1}^{n} \mathcal{Q}_i \times \mathcal{L}_i$ to denote a set of primitive queries of the *agent* $\mathcal{A}$. In similar manner $\mathcal{U}(\mathcal{A}) = \bigcup_{i=1}^{n} \mathcal{U}_i \times \mathcal{L}_i$ will denote the set of primitive updates of $\mathcal{A}$.

3.3 Semantics

The underlying notion in semantics definition is the notion of mental state. These mental states are evolved by transitions caused by application of updates to the states. Such updates are yielded by mental state transformers. The mental state transformation is defined in terms of the yields calculus.

Definition 3.4: (mental state)
Let $\mathcal{A}$ be a BSM over KR modules $\mathcal{M}_1, \ldots, \mathcal{M}_n$. A *state* (also called *mental 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. $\mathcal{S}$ denotes the space of all states over $\mathcal{A}$.

Definition 3.5: (applying an update)
The result of applying an update $\pi = (\oplus, \psi)$ on a state $\sigma = \langle \sigma_1, \ldots, \sigma_n \rangle$ of a BSM $\mathcal{A}$ over KR modules $\mathcal{M}_1, \ldots, \mathcal{M}_n$ is a new state $\sigma' = \sigma \bigoplus \pi$, such that $\sigma' = \langle \sigma_1, \ldots, \sigma_i', \ldots, \sigma_n \rangle$, where $\sigma_i' = \sigma_i \oplus \psi$, and both $\oplus \in \mathcal{U}_i$ and $\psi \in \mathcal{L}_i$ correspond to some $\mathcal{M}_i$ of $\mathcal{A}$. Applying an 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 $\pi_1 \bullet \pi_2$ is a new state $\sigma'' = \sigma' \bigoplus \pi_2$, where $\sigma' = \sigma \bigoplus \sigma_1$. $\sigma \xrightarrow{\pi_1 \bullet \pi_2} \sigma'' = \sigma \xrightarrow{\pi_1} \sigma' \xrightarrow{\pi_2} \sigma''$ denotes the corresponding compound transition.

Definition 3.6: (yields calculus)

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

$$\frac{yields(\tau, \sigma, \theta, \pi), \sigma \models \phi\theta}{yields(\phi \longrightarrow \tau, \sigma, \theta, \pi)} \qquad \frac{yields(\tau, \sigma, \theta, \pi), \sigma \not\models \phi\theta}{yields(\phi \longrightarrow \tau, \sigma, \theta, \mathbf{skip})} \quad (\textbf{Conditional})$$

$$\frac{yields(\tau_1, \sigma, \theta, \pi_1), yields(\tau_2, \sigma, \theta, \pi_2)}{yields(\tau_1 | \tau_2, \sigma, \theta, \pi_1), yields(\tau_1 | \tau_2, \sigma, \theta, \pi_2)} \quad (\textbf{Choice})$$

$$\frac{yields(\tau_1, \sigma, \theta, \pi_1), yields(\tau_2, \sigma \bigoplus \pi_1, \theta, \pi_2)}{yields(\tau_1 \circ \tau_2, \sigma, \theta, \pi_1 \bullet \pi_2)} \quad (\textbf{Sequence})$$

We say that τ yields an update set ν in a state σ under a substitution θ iff $\nu = \{\pi \mid yields(\tau, \sigma, \theta, \pi)\}$.

The definition of yields calculus was taken from [15], but one little change in notation was made to avoid ambiguity with previous symbols. In [15], an update yielded by mst is noted as ρ, but in this thesis, symbol ρ is associated with a plan rule of AgentSpeak(L) agent. So an update is denoted as π in the text, and this notational convention is strictly applied throughout all the chapters.

The mst **skip** yields the update **skip**. A primitive update mst $\oslash\psi$ under a substitution θ yields an appropriate update $(\oslash, \psi\theta)$. In def. 3.6 I assume that the provided substitution θ is ground. Conditional mst $\phi \longrightarrow \tau$ yields an appropriate update if the condition ϕ is satisfied under provided substitution θ, otherwise empty **skip** operation is yielded. In the case of non-deterministic choice one of the mst τ_1 or τ_2 is chosen and performed. Sequential mst $\tau_1 \circ \tau_2$ 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.

Definition 3.7: (BSM semantics)
A BSM $\mathcal{A} = (\mathcal{M}_1, \ldots, \mathcal{M}_n, \tau)$ can make a step from state σ to state σ' (induces transition $\sigma \xrightarrow{\pi} \sigma'$), if there exists a ground variable substitution θ, s.t., the agent program τ yields a non-empty update set ν in σ under θ and $\sigma' = \sigma \bigoplus \pi$, where $\pi \in \nu$ is an update.

A possibly infinite sequence of states $\sigma_1, \ldots, \sigma_n, \ldots$ is a *run of the BSM* $\mathcal{A}$ iff for each $i \geq 1$, $\mathcal{A}$ induces a transition $\sigma_i \rightarrow \sigma_{i+1}$.

The semantics of an agent system characterized by a BSM $\mathcal{A}$, is a set of all runs of $\mathcal{A}$.

In following text, an abstract BSM interpreter is presented in a pseudocode. In each deliberation cycle, the set ν of all possible updates is computed. Then one of them is non-deterministically chosen and is applied to the current state σ. Under _ in the $yields(...)$,

we denote a substitution of the set of all free variables used in the encoding of the agent program τ.

Algorithm 2 Abstract BSM interpreter
input: agent program τ, initial mental state σ_0

1: $\sigma \leftarrow \sigma_0$
2: **loop**
3: $\quad \nu \leftarrow \{\pi \mid yields(\tau, \sigma, \_, \pi)\}$
4: $\quad$ **if** $\nu \neq \emptyset$ **then**
5: $\quad\quad$ non-deterministically choose $\pi \in \nu$
6: $\quad\quad \sigma \leftarrow \sigma \oplus \pi$
7: $\quad$ **end if**
8: **end loop**

3.3.1 Operational view

The semantics of BSM can be seen in terms of traces within a labelled transition system over agent's mental states. Mental state transformers are interpreted as traces in a transition system over mental states and transitions induced by updates. The tool for connecting *yields calculus* and transitions in sense of operational semantics is called *behavioural frame* and was proposed in [15]. This notion formally captures the semantic structure induced by a set of KR modules of a BSM agent. It encapsulates the set of all mental states constructed from local states if the KR modules and applications of the corresponding updates between them. Note, that due to this fact, updates can be arbitrarily compound and complex.

Definition 3.8: (behavioural frame) Let $\mathcal{A} = (\mathcal{M}_1, \ldots, \mathcal{M}_n, \tau)$ be a BSM agent 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* $LTS(\mathcal{A}) = (\mathcal{S}, \mathcal{R})$, where $\mathcal{S} = \mathcal{S}_1 \times \cdots \times \mathcal{S}_n$ is the set of mental states of $\mathcal{A}$, and the transition relation $\mathcal{R}$ is defined in the following way:

$$\mathcal{R} = \left\{\sigma \xrightarrow{\pi} \sigma' \in \mathcal{S} \times \mathcal{U}(\mathcal{A}) \times \mathcal{S} \;\middle|\; \sigma' = \sigma \bigoplus \pi\right\} \cup \left\{\sigma \xrightarrow{\textbf{skip}} \sigma \in \mathcal{S} \times \mathcal{U}(\mathcal{A}) \times \mathcal{S}\right\}$$

$LTS(\mathcal{A})$ 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.

Chapter 4

Simulating of agents

In this chapter, it is presented a formal mechanism for the comparison of expressivity of programming languages, called (translation) bisimulation. The formalism presented in the sections of this chapter is taken from [3].

4.1 Towards translation bisimulation

An agent is able to simulate another agent if every legal (finite or infinite) computation of latter agent is matched by a legal computation of the first agent. Matching is a way of comparing of computations. It is derived from the notion of observation. The concept of simulation is used to compare the expressive power of AgentSpeak(L) relative to BSM. To do so, two things have to be done:

1. I have to find a corresponding BSM agent for each AgentSpeak(L) agent, and
2. to show that the computations of the AgentSpeak(L) agent are simulated by that of the BSM agent.

Definition 4.1: (computation) Let $\rightarrow$ be a translation relation on, so called, agent's configurations. $A_C \rightarrow A'_C$ is called a *computation step* (of agent's configuration A_C). A finite or infinite sequence of agent's configurations $A^1_C, A^2_C, \ldots$ such that $A^i_C \rightarrow A^{i+1}_C$ for all i is called a *computation*.

Two agents are compared using a comparison of the set of possible computation steps of the agents. Each computational step of one of the agents is matched or *simulated* by

the other one. This way of comparison is called *(strong) bisimulation*. A bisimulation is a binary relation between agents, based on a transition relation which defines legal computational steps of agents. To be able to compare two computation steps, it is important to decide, what aspects of the agent's configuration are relevant enough to be representative and satisfactory for the process of matching. Such aspects are included in a state-based concept of *observable*. Let's define a function $\mathcal{O}$ capable of observing the computational steps.

$$\mathcal{O} : A \to \Omega$$

where A is a set of agent's configurations and Ω is a set of observables.

Definition 4.2: (strong bisimulation) Let A and B be two sets of agent's configurations. A binary relation $R \subseteq A \times B$ over agent's configurations is a *strong bisimulation* if $(A_C, B_C) \in R$ implies,

1. Whenever $A_C \to A'_C$, then for some B_C, $B_C \to B'_C$, and $(A'_C, B'_C) \in R$,
2. whenever $B_C \to B'_C$, then for some A'_C, $A_C \to A'_C$, and $(A'_C, B'_C) \in R$, and
3. $\mathcal{O}(A_C) = \mathcal{O}(B_C)$

The strong bisimulation itself is not the right tool for simulating and comparing BSM and AgentSpeak(L) agents. Definition 4.2 assumes that the number of transitions of both agents during the computation is precisely equal. This property is relaxed in the definition of *weak bisimulation*, that allows different "lengths" of computational runs. But first let's define a *derived transition relation*. Note that computational steps $A_C \to A'_C$ such that $\mathcal{O}(A_C) = \mathcal{O}(A'_C)$ are not observable. In other words, these computational steps of the simulated agent do not change *observables* and due to this fact the simulating agent does not "see" any change, so these steps cannot be simulated. Such steps are called *internal steps*.

Definition 4.3: (derived transition relation $\Rightarrow$)
Let $\to$ be a translation relation on agent's configurations from A, and $\to^*$ denote the reflexive, transitive closure of $\to$.

- the *internal step translation relation* $\xrightarrow{i}$ is obtained by restricting $\to$: $A_C \to_i A'_C$ iff $A_C \to A'_C$ and $\mathcal{O}(A_C) = \mathcal{O}(A'_C)$,
- the *derived translation relation* $\Rightarrow$ is defined by: $A_C \Rightarrow A'_C$ iff there are agent's configurations $X_C, X'_C \in A$ such that $A_C \to_i^* X_C$, $X_C \to X'_C$ or $X_C = X'_C$, and $X'_C \to_i^* A'_C$

With the notion of derived translation relation it is possible to define the weak bisimulation. The definitions of strong and weak bisimulations are very similar. The only difference is in incorporating of derived transition relation.

Definition 4.4: (weak bisimulation) Let A and B be two sets of agent's configurations. A binary relation $R \subseteq A \times B$ over agent's configurations is a *weak bisimulation* if $(A_C, B_C) \in R$ implies,

(i) Whenever $A_C \rightarrow A'_C$, then for some B_C, $B_C \Rightarrow B'_C$, and $(A'_C, B'_C) \in R$,

(ii) whenever $B_C \rightarrow B'_C$, then for some A'_C, $A_C \Rightarrow A'_C$, and $(A'_C, B'_C) \in R$, and

(iii) $\mathcal{O}(A_C) = \mathcal{O}(B_C)$.

The definition 4.4 is much more appropriate for comparing computational runs of BSM and AgentSpeak(L) agents. One internal step of one agent's computational run can be matched by zero or more internal steps of the other agent's run. But for the purposes of comparing the expressivity of different languages, the definition 4.4 must be slightly improved. Let's assume that the mapping from one's agent language $\mathcal{L}$ to other's agent language $\mathcal{L}'$ is given. Let's call this mapping a *translation function*. If a translation function $\mathfrak{C}$ defines a weak bisimulation, i.e. $\mathfrak{C} = R$ then such specialized notion of bisimulation finally yields a suitable concept for comparing the expressive power of agent programming languages. Note that the definition 4.4 also assumes that the observables of both agent's configurations must be equal. This is not very comfortable and in some situations even impossible. It would be very handy to have a tool for mutual translation of observables. For this purpose, let's introduce a mapping called a *decoder*. A decoder maps observables from language $\mathcal{L}'$ back onto observables of $\mathcal{L}$. This brings a little asymmetry into the following definition in the sense of an assumption that $\mathcal{L}'$ simulates $\mathcal{L}$, but the other direction is not necessary.

Definition 4.5: (translation bisimulation) Let $\rightarrow_A$, $\rightarrow_B$ be two transition relations defined on the sets of agent's configurations A and B, respectively. Let $\mathfrak{C} : A \rightarrow B$ be a (total) mapping from A to B. Furthermore, $\mathcal{O}_A : A \rightarrow \Omega_A$ and $\mathcal{O}_B : B \rightarrow \Omega_B$ are two functions defining the observables of agent's configurations from the A and B. Then $\mathfrak{C}$ is a *transition bisimulation* if $B_C = \mathfrak{C}(A_C)$ implies,

(i) Whenever $A_C \rightarrow_A A'_C$, then $B_C \Rightarrow_B B'_C$, where $B'_C = \mathfrak{C}(A'_C)$,

(ii) whenever $B_C \rightarrow_B B'_C$, then for some A'_C, $A_C \Rightarrow_A A'_C$ such that $B'_C = \mathfrak{C}(A'_C)$, and

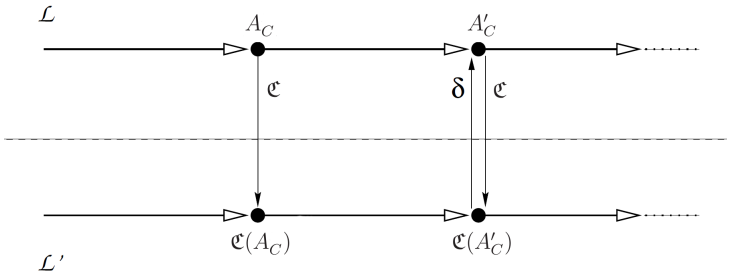

Figure 4.1: Scheme of translation bisimulation (adapted from[3]).

(iii) there is a *decoder* $\delta : \Omega_B \to \Omega_A$ such that $\delta(\mathcal{O}_B(B_C)) = \mathcal{O}_A(A_C)$.

Figure 4.1 represents the visualization of the transition bisimulation according to the definition 4.5. Upper line represents the computation run of the agent in language $\mathcal{L}$. Each agent's configuration $A_C \in A$ is translated into $\mathfrak{C}(A_C) \in B$. Each computational step $A \to_A A'$ can be simulated by more then one step (not depicted). It means that for each such transition there must exists a corresponding computation $\mathfrak{C}(A) \Rightarrow_B \mathfrak{C}(A')$ that affect the observables in the same way. This says the first and the third condition of the definition 4.5. In other words it is said that the behaviour of an agent $\mathcal{A}$ (represented by computation steps $A_C^i \to_A A_C^{i+1}$) can be simulated by an agent $\mathcal{B}$ (the behaviour of agent $\mathcal{B}$ is represented by computational steps $B_C^i \to_B B_C^{i+1}$). In the other direction, it must be sure, that a translated agent $\mathcal{B}$ generates only the behaviour that produces agent $\mathcal{A}$ and no other. This is formally covered in the second condition of the definition 4.5. It is a kind of dual situation. It must be possible to simulate each computational step of an agent $\mathcal{B}$ by an agent $\mathcal{A}$. The concept of translation bisimulation seems to be the right tool for examination of the expressive power of programming languages. It can be said that programming language $\mathcal{L}'$ has at last the same expressive power as the source language $\mathcal{L}$ in case, that it is possible to find such translation bisimulation function $\mathfrak{C}$ that maps all the agents from $\mathcal{L}$ to some suitable set of agents in the other language $\mathcal{L}'$. But as mentioned in [3], the function $\mathfrak{C}$ (as well as $\mathfrak{C}$) must be restricted because the whole mechanism of language expressivity comparison is not strong enough due to the lack of *compositionality* requirement. It states that the agents should not change its global structures. Agents' logical parts should be preserved during the simulation. In general this constraint can be formalized by the requirement that the translation function $\mathfrak{C}$ and the decoder δ are *compositional*: Every operator *op* of the source language is translated into a *context* $C[X_1, \ldots, X_n]$ (assuming that n is the arity of *op*) of the target language such that a program $op(A_1, \ldots, A_n)$ is translated into $C[\mathfrak{C}(A_1), \ldots, \mathfrak{C}(A_n)]$. In order to

account for the complex structure of an agent (that is, its various components comprising its beliefs, intentions etc.) let's assume that an agent is a tuple $A = \langle E_1, \ldots, E_n \rangle$, where each of the E_i is a subset of the expressions of a programming language $\mathcal{L}$, i.e. $E_i \subseteq \mathcal{L}$. Moreover let's assume that the expressions in $\mathcal{L}$ are inductively defined, and it is required that the translation function $\mathfrak{C}$ is defined compositionally as defined above. A mapping from agents $\langle E_1, \ldots, E_n \rangle$ from $\mathcal{L}$ to agents $\langle F_1, \ldots, F_m \rangle$ is induced by $\mathfrak{C}$ by means of the pre-specified selection criterion which determines for each expression $e \in E_i$ a corresponding target F_j such that $\mathfrak{C}(e) = F_j$. This is a way of agents' components translation.

4.2 Expressibility

The notion of expressibility is clarified in [4].

Definition 4.6: (expressibility) Let $\mathcal{O}_A : A \to \Omega_A$ and $\mathcal{O}_B : B \to \Omega_B$ be two functions from the set of all agent configurations A, formed by $\mathcal{L}$, and some suitable set of agent configurations B, formed by $\mathcal{L}'$, to the corresponding set of observables. Then we say that $\mathcal{L}'$ *has at least the same expressive power as* $\mathcal{L}$ if there is a mapping $\mathfrak{C} : A \to B$ and a mapping $\delta : \Omega_B \to \Omega_A$ which satisfy the following conditions:

(i) $\mathfrak{C}$ and δ are compositional,

(ii) $\mathfrak{C}$ is a translation bisimulation

Chapter 5

Embedding of AgentSpeak(L) in BSM

This chapter expresses my research efforts aimed towards the embedding of AgentSpeak(L) into BSM. Since the language BSM is highly modular, first I created the basic building blocks (KR modules), and then sticked these modules together using the mental state transformer claiming that such BSM compilation can emulate the behaviour of the AgentSpeak(L) agent. An important result of the chapter involves the specification of translation function that maps every AgentSpeak(L) to its BSM counterpart. This chapter ends with a theorem (and its proof) stating that the language of BSM has at least the same expressive power as AgentSpeak(L).

5.1 Computations in AgentSpeak(L)

The technique of bisimulation requires two notions to be specified for both examined languages. The notion of computation and the notion of observable. In the case of AgentSpeak(L), to formulate a computation, the derivation relation $\vdash$ on BDI configurations within the set of proof rules is used. Each proof rule defines the means of BDI configuration evolving as one computation step. The derivation relation is written as $C \vdash C'$ and specifies the set of all possible computations of an agent.

Definition 5.1: (BDI derivation) A *BDI derivation* D_{BDI} is a finite or infinite sequence of BDI configurations, i.e., $C_0, \ldots, C_n, \ldots$, where each C_{i+1} is derivable from C_i according to a proof rule, i.e., $C_i \vdash C_{i+1}$. We write $C_i \overset{p}{\vdash} C_j$ to emphasise that C_j is

derived from C_i using a proof rule p.

There is a relationship between a BDI derivation and an interpreter. An interpreter tries to apply the proof rules to BDI configurations in a specified order. Such an order is called an *invocation sequence*, and it is specified below.

Definition 5.2: (invocation sequence) Let $D_{BDI} = C_0, \ldots, C_n, \ldots$ be a BDI derivation such that $C_0 \overset{p_1}{\vdash} C_1 \overset{p_2}{\vdash} \cdots \overset{p_n}{\vdash} C_n \overset{p_{n+1}}{\vdash} \ldots$. An *invocation sequence* S_{inv} is a finite or infinite sequence of labels of the proof rules that an interpreter tries to invoke, i.e., $S_{inv} = p_1, \ldots, p_n, \ldots$, where $p_i \in$ **ProofRules**. The set of all invocation sequences is denoted as $\mathbb{S}$.

Furthermore, a BDI derivation D_{BDI} can be generated by a tuple $\langle C_0, S_{inv} \rangle$, where C_0 is an initial BDI configuration, and S_{inv} is an invoking sequence. We write

$$D_{BDI} \approx \langle C_0, S_{inv} \rangle$$

to denote that D_{BDI} is generated by $\langle C_0, S_{inv} \rangle$. The construction of D_{BDI} by $\langle C_0, S_{inv} \rangle$ is performed as follows:

$$D_{BDI} = C_0, C_1, \ldots, C_n, \ldots \text{ such that } C_{j-1} \overset{S_{inv_j}}{\vdash} C_j, \;\; j > 0,$$

where S_{inv_j} denotes the j^{th} item of the invocation sequence S_{inv}.

The result of the analysis of the BDI interpreter is that the proof rules are attempted to invoke in specified invocation sequences (def. 2.18). These sequences are called *valid*, and are defined as follows (def. 5.3).

Definition 5.3: (valid invocation sequence) An invocation sequence $S_{inv} \in \mathbb{S}$ is *valid* iff it is formed of admissible subsequences, i.e, $S_{inv} = p_1, \ldots, p_n$ such that $p_i \in \{\bar{\alpha}, \bar{\beta}, \bar{\gamma}, \bar{\delta}\}$. Furthermore, if C_0 is an initial BDI configuration, and S_{inv} is a valid invocation sequence, then a BDI derivation D_{BDI} generated by $\langle C_0, S_{inv} \rangle$, i.e., $D_{BDI} \approx \langle C_0, S_{inv} \rangle$, is called a *valid BDI derivation.*

Note that since a BDI interpreter invokes the proof rules in the order of a valid invocation sequence, the AgentSpeak(L) program creates only valid BDI configurations. This is an important consequence of the analysis of the interpreter.

At this point, two important choices have to be made. The bisimulation framework assumes two functions to be specified. The first of them maps the BDI configuration

to some set of observables. The question is, which set should represent the observables. Note, that the sequence of such observables is then used for the expressibility comparison. The sequences of both languages must match. The match proves that the behaviours of both agents are the same. It is a very natural choice, that the beliefs of the agent should form the set of observables.

The second function needed for bisimulation to be fully established is a decoder δ. This function maps the observables from the latter language to the former in order to be comparable. It will be shown later, that in this case, δ function is an identity.

Definition 5.4: (configuration observables) Let $\mathcal{C}_{BDI}$ be a set of all BDI configurations, and let $\langle C_E, C_B, C_I \rangle \in \mathcal{C}_{BDI}$ be such a configuration. The function

$$\mathcal{O}_{BDI} : \mathcal{C}_{BDI} \to \wp(\mathsf{Lit})$$

is defined by $\mathcal{O}_{BDI}\big(\langle C_E, C_B, C_I \rangle\big) = C_B$. $\mathcal{O}_{BDI}$ yields the *observable* of a configuration.

At this point, the syntax and the semantics of AgentSpeak(L) seem to be fully specified. All the notions introduced in previous sections are utilized for the formulation of the formal proof (presented in chapter 5) that AgentSpeak(L) has at least the same expressivity as Behavioural State Machines. Formal introduction of BSM is presented in the following chapter.

5.2 Towards BSM translation

The formalisms of both languages AgentSpeak(L) and BSM are described in previous chapters. At this moment, we are facing the problem of incorporating the framework of bisimulation (formally specified in chapter 4) to work with these two languages as underlying domains. The structure of an AgentSpeak(L) agent is clear. There are strictly defined entities (like belief base, set of intentions, the plan library and so on) and the proof system defining the interaction between those entities. On the other hand, the BSM does not commit to a certain type of technology or a programming language. The number of KR modules is entirely arbitrary as well as the module's capabilities defined through query and update interfaces. A mental state transformer acts as a glue, that sticks all the KR modules together, allows interaction between modules, and induces transitions.

The development of KR modules and the form of a mental state transformer is in the hands of an agent designer. Following sections of this chapter express my efforts

aimed towards the design of a BSM agent capable of performing the same behaviour as AgentSpeak(L) agent.

5.3 Creation of KR modules

AgentSpeak(L) agent consists of four parts. Set of events E, belief base B, set of intentions I and a plan library P. I think that a reasonable idea is to translate each part to a specific KR module. I tried to stay as close a possible to the premise that query and update operators of KR modules should be as trivial as possible. Complex behaviour patterns are provided by mst.

The first module $\mathcal{M}_E$ represents the formal translation of AgentSpeak(L) events. Particular events are couples of triggering event and an intention (formally defined by def. 2.9). Note that the state of the module can be modified only through update operators. By inspecting the proof rules of AgentSpeak(L) there are two update operations to be specified. One for addition ($\oplus_E$) and one for deletion ($\ominus_E$) of an event. The only query operator ($\models_E$) is used to check the current set of events.

Definition 5.5: (translation of events)

$$\mathfrak{C}(E) = \mathcal{M}_E = \left\langle E, \mathsf{Ev}, \{\models_E\}, \{\oplus_E, \ominus_E\} \right\rangle \tag{5.1}$$

where:

- $E \models_E \langle +!g, i\rangle = \begin{cases} \text{true} & \text{if there is } e \in E \text{ of the form } \langle +!h, i\rangle, \\ & \text{such that } g\eta = h\eta \text{ and } \eta \text{ is MGU} \\ \text{false} & \text{otherwise,} \end{cases}$
- $E \oplus_E \langle +!g, i\rangle = E \cup \{\langle +!g, i\rangle\}$,
- $E \ominus_E \langle +!g, i\rangle = E \setminus \{\langle +!g, i\rangle\}$.

The belief base of AgentSpeak(L) is translated into the second module $\mathcal{M}_B$. This module offers one query operator ($\models_B$) that corresponds to the classic logical entailment known from Prolog [5]. Note, that **ExecAct** (defined by def. 2.14) is the only proof rule affecting the belief base. The belief base is entirely replaced on the basis of the function $\mathcal{T}$ that defines an update semantics of actions. The update operator ($\oplus_B$) is defined to reflect this fact.

Definition 5.6: (translation of belief base)

$$\mathfrak{C}(B) = \mathcal{M}_B = \langle B, \mathsf{Lit}, \{\models_B\}, \{\oplus_B\} \rangle \tag{5.2}$$

where:

– $B \models_B \phi = B \models \phi\xi$, where ξ is a ground substitution,

– $B \oplus_B B' = B'$.

The translation of AgentSpeak(L) intentions represents most likely the toughest concern. Due to the fact that intentions are stacks, there are two update operators implementing two of the standard stack operations ($\oplus_{I_{PUSH}}$ and $\ominus_{I_{POP}}$). Note, that the former pushes the whole plan rule to the intention, the latter pops only one item of the plan body. The third update operator $\ominus_I$ removes the whole intention. Another update operator ($\ominus_{I_{CLEAN}}$) is used for cleaning purposes. It removes an empty plan from the top of an intention stack. Informally, the query operator $\models_I$ checks whether there exists an intention stack that contains a plan on its top starting with a particular element.

Definition 5.7: (translation of intentions)

$$\mathfrak{C}(I) = \mathcal{M}_I = \langle I, \mathsf{Int}, \{\models_I, \models_{single}, \models_{I_\circledast}\}, \{\oplus_{I_{PUSH}}, \ominus_I, \ominus_{I_{POP}}\} \rangle \tag{5.3}$$

where:

– $I \models_I (i, h) = \begin{cases} \text{true} & \text{if } \exists i \in I \text{ of the form } i = [\rho_1 \ddagger \ldots \ddagger \rho_z] \\ & \text{and } \rho_z = te : \psi \leftarrow h; \ldots; h_n \\ \text{false} & \text{otherwise,} \end{cases}$

– $I \models_{I_{empty}} i = \begin{cases} \text{true} & \text{if } i \text{ is empty} \\ \text{false} & \text{otherwise,} \end{cases}$

– $I \models_{I_\circledast} g = \begin{cases} \text{true} & \text{if } g \text{ is one of the following:} \\ & \quad i) \text{ an achievement goal (if } \circledast = ach), \\ & \quad ii) \text{ a test goal (if } \circledast = test), \\ & \quad iii) \text{ an action (if } \circledast = action), \\ & \quad iv) \text{ true (if } \circledast = true), \\ \text{false} & \text{otherwise,} \end{cases}$

– $I \oplus_{I_{PUSH}} (i, \rho) = I \cup \{[\rho_1 \ddagger \ldots \ddagger \rho_z \ddagger \rho]\xi\}$,

where:

$i = [\rho_1 \ddagger \ldots \ddagger \rho_z]$,

ξ is provided substitution for ρ,

– $I \ominus_I i = I \setminus \{i\}$,

– $I \ominus_{I_{POP}} i = \big(I \setminus \{i\}\big) \cup \{i'\}$,

such that:

$i = [\rho_1 \ddagger \ldots \ddagger (e : \psi \leftarrow h_1; h_2; \ldots; h_n)]$,

$i' = [\rho_1 \ddagger \ldots \ddagger (e : \psi \leftarrow h_2; \ldots; h_n)]$,

h_1 is an action or a test goal,

– $I \ominus_{I_{CLEAN}} i = \big(I \setminus \{i\}\big) \cup \{i'\}$,

such that:

$i = [\rho_1 \ddagger \ldots \ddagger \rho_z \ddagger (e : \psi \leftarrow)]$,

$i' = [\rho_1 \ddagger \ldots \ddagger \rho_z]$.

The fourth and the last module $\mathcal{M}_P$ represents the AgentSpeak(L) plan library. $\mathcal{M}_P$ serves the BSM agent as a plan provider. This is the only purpose of the module. An appropriate plan is provided through one query operator $\models_P$ dedicated exclusively to that purpose.

Definition 5.8: (translation of plan library)

$$\mathfrak{C}(P) = \mathcal{M}_P = \big\langle P, \mathsf{PlanRules}, \{\models_P\}, \{\}\big\rangle \tag{5.4}$$

where:

– $P \models_P \rho = \begin{cases} \text{true} & \text{if there is a plan } \rho \in P \\ & \text{of the form } \rho = +!g : \psi \leftarrow h_1; \ldots; h_n \\ \text{false} & \text{otherwise.} \end{cases}$

These four KR modules work together on the basis of the mental state transformer that is presented in the next section.

5.4 Creation of mental state transformer

The behaviour of AgentSpeak(L) agent is encoded within six proof rules (namely **IntendMeans**, **ExecAch**, **ExecAct**, **ExecTest**, **CleanStackEntry** and **CleanSetInt**). These proof rules are applied by an interpreter such that a valid invocation sequence (def. 5.3) is followed. In contrast, the behaviour of BSM is driven by a mental state transformer. In the further text, I create a set of six underlying msts (τ_{u_i}) corresponding to the particular proof rules. As a next step, I incorporate these msts into an overall single one ($\tau_{overall}$). The main requirement is that $\tau_{overall}$ should adopt the behaviour of the BDI interpreter, i.e., it should apply all the underlying msts τ_{u_i} such that corresponding proof rules should form an admissible sequence. Now, let's begin with the construction of particular underlying msts.

The first rule **IntendMeans** (def. 2.12) takes an event $\langle te, i\rangle$, searches for a suitable plan ρ (matching the triggering event te) and inserts the plan at the top of an intention i. To model this rule by mst, the capabilities of all the modules must be taken into account. See def. 5.9:

Definition 5.9: (mst for proof rule IntendMeans)

$$\tau_{im} \;=\; \big(\models_E \langle te, i\rangle\theta_E \wedge \models_B \psi\theta_E\theta_B \wedge \models_P \rho\theta_E\theta_B\big) \longrightarrow \big(\oplus_{I_{PUSH}}(i,\rho)\theta_E\theta_B \circ \ominus_E\langle te, i\rangle\theta_E\big)$$

where:

– $\rho = te : \psi \leftarrow h_1; \ldots; h_n$.

The presence of an event with te that is matched with the plan rule ρ is assured by the query operator $\models_E$ of module $\mathcal{M}_E$. Note that substitution θ_E now contains a concrete instance of the event (i.e., the triggering event te and the intention i). The concrete form of ρ is provided by module $\mathcal{M}_P$ through its query operator $\models_P$. Since θ_E contains a substitution for a triggering event te, it is used for an instantiation of a triggering event of the plan ρ. The plan is applicable if it is matched by triggering event, and its context ψ is a logical consequence of the belief base. The latter objective is managed by $\models_B$ taking the context ψ of the plan ρ as an argument. The concrete instantiation of ψ is stored in θ_B. The conjunction of these tree queries forms a condition in a conditional mst. By inspection of the conclusion of the rule in def. 2.12, the change in the resulting configuration can be reflected by the sequence of two consecutive updates. The first update must instantiate the plan ρ, insert the plan to the top of intention i and put the

intention itself to the set of intentions. Note that right this concern is defined in the semantics of $\oplus_{I_{PUSH}}$ operator (cf. def. 5.7). The second one deals with the removal of the event $\langle te, i\rangle$ from the set of events. This is managed by $\ominus_E$. Note that θ_E specifies what event to remove.

I will introduce another notational convention here. When an mst uses substitutions, e.g., τ_{im} in previous definition uses θ_E and θ_B, we can write:

$$\tau_{im} = \tau_{im}(\theta_E, \theta_B).$$

The proof rule **ExecAch** covers the situation when an execution of an achievement goal $!g$ is the case. In this situation new event has to be created, and the whole intention containing $!g$ must be removed from the set of intentions. See def. 5.10:

Definition 5.10: (mst for proof rule ExecAch)

$$\tau_{ach} \;=\; \tau_{ach}(\theta_I) \;= \models_{I_{ach}} g\theta_I \longrightarrow \big(\oplus_E \big\langle +g, i\big\rangle\theta_I \;\circ\; \ominus_I i\theta_I \big)$$

In this mst, I assume that θ_I is provided by some outer mst (in fact by $\tau_{overall}$ that will be defined later). θ_I must provide a concrete instance of an intention i, and the first element g of the plan residing on the top of i, i.e., I assume that i and g are known (the reason of this assumption will be clear in further text, when $\tau_{overall}$ will be defined). The query operator $\models_{I_{ach}}$ checks whether provided g is an achievement goal. If so, a new event $\langle +g, i\rangle\theta_I$ is created through the update operator $\oplus_E$. Finally, the intention i is removed from the set of intentions of module $\mathcal{M}_I$. Note that if g is not an achievement goal then an empty **skip** operation is yielded (def. 3.6).

Another proof rule **ExecAction** deals with an action execution. τ_{action} models the proof rule **ExecAction**. See def. 5.11:

Definition 5.11: (mst for proof rule ExecAction)

$$\tau_{action} \;=\; \tau_{action}(\theta_I) \;= \models_{I_{action}} g\theta_I \longrightarrow \ominus_{I_{POP}} i\theta_I \;\circ\; \oplus_B \mathcal{T}\big(g\theta_I, B\big)$$

Again, we assume that an intention i, and the the first item g of the topmost plan of i is provided in θ_I (the same assumption as in previous case). τ_{action} does the following. First, it ensures that g is an action. If so, it continues by removing the action from the plan $\ominus_{I_{POP}}$, and the belief base is modified using the same function $\mathcal{T}$ used in def. 2.14. The update operator $\oplus_B$ replaces the current content of belief base with a return value of the function $\mathcal{T}$ that specifies the update semantics of actions.

See mst τ_{test} (def. 5.12) modelling another proof rule, **ExecTest**:

Definition 5.12: (mst for proof rule ExecTest)

$$\tau_{test} \;=\; \tau_{test}(\theta_I, \theta_B) \;=\; \big(\models_{I_{test}} g\theta_I \wedge \models_B g\theta_B \big) \longrightarrow \ominus_{I_{POP}} i\theta_I\theta_B$$

The substitution must θ_I be provided by outer mst. That is an assumption. The condition in τ_{test} is now formed of a conjunction of two parts. The former checks whether g is a test goal. The latter checks whether the test goal is a logical consequence of the belief base (the instantiation of the variables of the test goal is stored in θ_B). If both queries are evaluated to `true`, the test goal is removed from the plan, and a compound substitution is applied to the rest of the intention i. This operation is performed by $\ominus_{I_{POP}}$.

The proof rule **CleanStackEntry** is modelled by τ_{cse} as follows:

Definition 5.13: (mst for proof rule CleanStackEntry)

$$\tau_{cse} \;= \models_{I_{true}} g\theta_I \longrightarrow \ominus_{I_{CLEAN}} i\theta_I \circ \ominus_{I_{POP}} i\theta_I$$

We assume that θ_I contains a selected intention i and the first item g of the plan residing on the top of i. The same assumption was made in previous cases. θ_I is provided by outer mst. τ_{cse} checks whether the g contains `true` indicating that the plan is empty. An empty plan on the top of i is removed by an update operator $\ominus_{I_{CLEAN}}$. The plan one entry below contains an achievement goal that has been accomplished by the removed plan. The achievement goal is removed as well ($\ominus_{POP}$).

Now, we have all but one proof rule modelled by msts. **CleanIntSet** is remaining, and it will be specified later. Let's begin with the creation of overall mst $\tau_{overall}$. As mentioned in previous text, $\tau_{overall}$ should incorporate the underlying msts such that the corresponding proof rules should form an admissible sequence. By inspection of def. 2.18, we see, that the first proof rule of each admissible sequence is **IntendMeans**. Due to this fact, the overall mst should definitely begin by τ_{im}. When we look back to the sequences, after applying **IntendMeans**, the situation is branching into four different scenarios, since **IntendMeans** is followed by a different proof rule in each sequence. We know that the application of **IntendMeans** covers one mode of the BDI interpreter (*intention update*), and this mode is followed by another one (*intention execution*). The branching corresponds to the cases (*#case1 - 4*) discussed in the BDI interpreter analysis, i.e., there must be an intention i, and an *actual item* g that represents the first item in the body

of the plan residing on the top of the intention i. See the first part of $\tau_{overall}$ called τ^1 (equation 5.5).

$$\begin{aligned}\tau^1 \;=\; \tau^1(\theta_E,\theta_B,\theta'_B,\theta_I,\theta_I^{\exists}) \;= \models_I (i^{\exists},g^{\exists})\theta_I^{\exists} \rightarrow \Big(\overbrace{\tau_{im}(\theta_E,\theta_B)}^{\text{intention update}} \circ \\ \circ \models_I (i,g)\theta_I \rightarrow \big(\underbrace{\tau_{ach}(\theta_I) \circ \tau_{cse}(\theta_I) \circ \tau_{test}(\theta_I,\theta'_B) \circ \tau_{action}(\theta_I)}_{\text{intention execution}} \big) \Big)\end{aligned} \tag{5.5}$$

The first query operator $\models_I$ is used to check whether there exists an intention in the module $\mathcal{M}_I$. Note that concrete values of elements superscripted with '$\exists$' sign are not relevant, however, their existence is necessary since this condition corresponds to the condition stated in the 'while cycle' of the BDI interpreter (in line 1 of the algorithm 1). It is followed by τ_{im} (representing the *intention update* mode of BDI interpreter). The second query operator $\models_I$ tries to find an intention i with its actual item g. This information is stored to θ_I, and is propagated further. This is the place where θ_I comes from.

Now, let's take a closer look at the *intention execution* part of the τ^1. It is form of a sequence of four underlying msts. The order of the sequence is the same as specified in the pseudo-code of the BDI algorithm (see Algorithm 1). Note that all the msts in the sequence utilize the substitution θ_I, i.e., all the msts assume that an intention i, and its actual item g is provided. Also note that all the msts forming the sequence (i.e., $\tau_{ach}, \tau_{clean}, \tau_{test}$ and τ_{action}) are implemented as *conditional*. All the conditions contain a test for g. Since g is an element of the body of a plan, it may be one of the following:

- *an achievement goal*: it is captured in the condition of τ_{ach},
- `true`: it is captured in the condition of τ_{clean},
- *a test goal*: it is captured in the condition of τ_{test},
- *an action*: it is captured in the condition of τ_{action}.

There is no other possibility that g may acquire. Since the conditions are mutually disjoint, it is guaranteed that only one mst is enabled at a time. For example, if θ_I is such that g is an action, the conditions of τ_{ach}, τ_{clean} and τ_{test} will be evaluated to `false`, and the condition of τ_{action} will be evaluated to `true` (since g is an action). By def. 3.6 we know that if the condition of *conditional* mst is evaluated to `false`, an empty **skip** update is yielded. In fact, this feature can provide the branching procedure needed for emulation

of admissible sequences. Notice that τ^1 can invoke msts that corresponds to $\bar{\alpha}$ sequence. Other sequences cannot be modelled, since a mst for the proof rule **CleanIntSet** is not yet established. This is done in the following text.

Cleaning rule **CleanIntSet** removes an empty intention i from the set of intentions. The intention i must be checked whether it is empty. Note that $\ominus_{I_{POP}}$ is the only update operator that removes items from the plan body of intentions, i.e., it is the only operator that can get the intention empty. And hence, it is necessary to apply the intention revision (mst that models **CleanIntSet**) after the application of $\ominus_{I_{POP}}$. All the msts that implement this update operator include: τ_{clean}, τ_{test} and τ_{action}. This fact also corresponds with the form of admissible sequences $\bar{\beta}, \bar{\gamma}$ and $\bar{\delta}$ since **CleanIntSet** is contained only in these (see def. 2.18). Conversely, the sequence $\bar{\alpha}$ is the only one without **CleanIntSet**. When all these facts are taken into account, the resulting τ_{cis} states as follows:

Definition 5.14: (mst for proof rule CleanIntSet)

$$\tau_{cis} \; = \tau_{cis}(\theta_I) \; = \; \neg \models_{I_{ach}} g\theta_I \wedge \; \models_{I_{empty}} i\theta_I \longrightarrow \ominus_I i\theta_I$$

Assuming that θ_I is provided by outer mst, the intention i is removed if

– the actual item g is not an achievement goal ($\neg \models_{I_{ach}} g\theta_I$), and

– the intention i is empty ($\models_{I_{empty}} i\theta_I$)

At this point we have all the constituents of the $\tau_{overall}$ which is defined as follows:

Definition 5.15: (overall mst)

$$\tau_{overall} \; = \; \tau_{overall}(\theta_E, \theta_B, \theta'_B, \theta_I, \theta_I^{\exists}) \; = \models_I (i^{\exists}, g^{\exists})\theta_I^{\exists} \rightarrow \Big(\overbrace{\tau_{im}(\theta_E, \theta_B)}^{\text{intention update}} \circ$$
$$\circ \models_I (i, g)\theta_I \rightarrow \big(\underbrace{\tau_{ach}(\theta_I) \circ \tau_{cse}(\theta_I) \circ \tau_{test}(\theta_I, \theta'_B) \circ \tau_{action}(\theta_I)}_{\text{intention execution}} \circ \underbrace{\tau_{cis}(\theta_I)}_{\text{cleaning}} \big) \Big)$$

5.5 Translation of the agent

In the previous text, I have translated all the components of AgentSpeak(L), i.e., the set of events E, the belief base B, the set of intentions I, and the plan library P into corresponding KR modules. Then I created the overall mst based on the analysis of the BDI

interpreter. Now, everything is ready to propose the translation of the AgentSpeak(L) agent into BSM.

Definition 5.16: (translation of an agent) Let $\mathcal{A} = \langle E, B, I, P\rangle$ be an AgentSpeak(L) agent definition. $\mathcal{A}$ is translated into BSM as follows:

$$\mathfrak{C}\big(\mathcal{A}\big) = \big\langle \mathfrak{C}(E), \mathfrak{C}(B), \mathfrak{C}(I), \mathfrak{C}(P), \tau_{overall}\big\rangle = \big\langle \mathcal{M}_E, \mathcal{M}_B, \mathcal{M}_I, \mathcal{M}_P, \tau_{overall}\big\rangle,$$

As a next step, I will introduce the correspondence between BDI configurations and mental states. It is specified by the translation function $\mathfrak{C}$ as well. The specification of the function $\mathfrak{C}$ is overloaded such that it translates BDI configurations into a mental state. This relationship is trivial since $\mathfrak{C}$ maps BDI configurations to mental states by identity. The translation of the BDI configuration $\langle C_E, C_B, C_I\rangle$ into BSM mental state is performed in a following way:

Definition 5.17: (translation of a BDI configuration) Let $\langle C_E, C_B, C_I\rangle$ be a BDI configuration. The function $\mathfrak{C}$ translates BDI configurations into mental state as follows:

$$\mathfrak{C}\big(\langle C_E, C_B, C_I\rangle\big) = \big\langle \mathfrak{C}(C_E), \mathfrak{C}(C_B), \mathfrak{C}(C_I)\big\rangle = \langle \sigma_E, \sigma_B, \sigma_I\rangle,$$

such that each item of the configuration is translated by identity:

- $\mathfrak{C}(C_E) = C_E = \sigma_E \in E$ where E is a set of states of $\mathcal{M}_E$,
- $\mathfrak{C}(C_B) = C_B = \sigma_B \in B$ where B is a set of states of $\mathcal{M}_B$, and
- $\mathfrak{C}(C_I) = C_I = \sigma_I \in I$ where I is a set of states of $\mathcal{M}_I$.

The notion of computation for a BSM agent was equivalently defined in def. 3.7 as a BSM run, i.e., the possibly infinite sequence of mental states. New mental states are induced from old ones by mst τ. τ yields an update π (possibly an update set ν) on the basis of yields calculus by def. 3.6. π induces a transition from old mental state σ to new state σ'. We write $\sigma \xrightarrow{\pi} \sigma'$.

Next, the function for extracting observable elements from a mental state needs to be specified. The mental state of a BSM agent is formed of the states of the particular KR modules. The following definition is general and selects one constituent of the mental state to define observables.

Definition 5.18: (mental state observables) Let Σ_{BSM} be a set of all mental states, and let $\sigma = \langle \sigma_E, \sigma_B, \sigma_I \rangle \in \Sigma_{BSM}$ be such a mental state. The function

$$\mathcal{O}_{BSM} : \Sigma_{BSM} \rightarrow \wp(\mathsf{Lit})$$

is defined by $\mathcal{O}_{BSM}\big(\langle \sigma_E, \sigma_B, \sigma_I \rangle\big) = \sigma_B$. σ_B here represents the internal state representing the translated belief base. $\mathcal{O}_{BSM}$ yields the *observable* of a mental state.

The operations of particular modules are designed in order to reflect the behaviour of the proof rules in the AgentSpeak(L) proof system. Each proof rule is modelled by a mst. All these (underlying) msts are put together to create one (overall) mst ($\tau_{overall}$). I claim that $\tau_{overall}$ simulates the BDI interpreter.

We know that an AgentSpeak(L) program tries to invoke the proof rules in a manner of admissible sequences ($\bar{\alpha}, \bar{\beta}, \bar{\gamma}$ and $\bar{\delta}$) defined in def. 2.18. This statement is based on the analysis of the BDI interpreter. We also know that such invocations generate BDI configurations such that a valid BDI derivation D is formed (def. 5.3). To prove that BSM has at least the same expressive power as AgentSpeak(L), I have to show that the behaviour caused by an AgentSpeak(L) program can be simulated by $\tau_{overall}$, and furthermore $\tau_{overall}$ must be capable of generating the sequence of mental states S_σ such that D is simulated by S_σ (no other sequences of mental states are allowed).

Recall that $\tau_{overall}$ has following form:

$$\tau_{overall} = \models_I (i^\exists, g^\exists)\theta_I^\exists \rightarrow \big(\overbrace{\tau_{im}}^{\sigma_{im}} \circ \models_I (i, g)\theta_I \rightarrow (\overbrace{\tau_{ach}}^{\sigma_{ach}} \circ \overbrace{\tau_{cse}}^{\sigma_{cse}} \circ \overbrace{\tau_{test}}^{\sigma_{test}} \circ \overbrace{\tau_{action}}^{\sigma_{action}} \circ \overbrace{\tau_{cis}}^{\sigma_{cis}})\big) \quad (5.6)$$

Overbraced 'sigmas' represent the mental state after the application of corresponding mst. For example σ_{test} is a mental state after the application of τ_{test}.

Definition 5.19: (BSM evolution cycle) Let σ_0 be an initial mental state, and let $\sigma_{im}, \sigma_{ach}, \sigma_{cse}, \sigma_{test}, \sigma_{action}$ and σ_{cis} be mental states induced during the execution od $\tau_{overall}$ (eq. 5.6). Then a BSM run R_e defined as:

$$R_e = \sigma_0 \xrightarrow{\pi_{im}} \sigma_{im} \xrightarrow{\pi_{ach}} \sigma_{ach} \xrightarrow{\pi_{cse}} \sigma_{cse} \xrightarrow{\pi_{test}} \sigma_{test} \xrightarrow{\pi_{action}} \sigma_{action} \xrightarrow{\pi_{cis}} \sigma_{cis}$$

is called the *BSM evolution cycle.*

$\tau_{overall}$ can generate a BSM evolution cycle only if when the first query operator $\models_I (i^\exists, g^\exists)\theta_I^\exists$ is evaluated to `true`, i.e., the substitution $\theta_I^\exists$ is found. It means that the set of intentions must contain at least one intention. Note that the same condition is stated

in the 'while cycle' of the BDI interpreter (in line 1 of the algorithm 1). Let's examine what happens when this condition is not satisfied: since the condition of the 'while cycle' in the BDI interpreter is no longer satisfied, it terminates an generates an empty trace (an empty sequence of BDI configurations). The BSM agent yields an empty update (**skip**) since there is no intention in the intention set, i.e., there is no such $\theta_I^{\exists}$, hence $\models_I (i,g)\theta_I^{\exists}$ is evaluated to `false`, and **skip** is yielded according to def. 3.6.

In the following text, I will assume, that the set of intentions is not empty. In that case, the BDI interpreter can generate valid BDI derivations.

Theorem 5.1: *Let C_0 be an initial BDI configuration and $\bar{\alpha}$ is an admissible sequence according to def. 2.18. Then the BDI derivation $D_{\bar{\alpha}} \approx \langle C_0, \bar{\alpha}\rangle$ (generated by the AgentSpeak(L) program) can be simulated by $\tau_{overall}$.*

Proof: I have to show that for each sequence of BDI configurations generated by the admissible sequence $\bar{\alpha}$, there exists the corresponding sequence (the BSM evolution cycle) of mental states that can be generated by $\tau_{overall}$.

Recall the sequence $\bar{\alpha}$ (by def. 2.18):

$$\bar{\alpha} = \mathbf{IntendMeans}, \mathbf{ExecAch}.$$

The interpreter will try to invoke the proof rule **IntendMeans** followed by another one, **ExecAch**. I say 'try to invoke' since the proof rule may not be applicable.

IntendMeans: Let's assume that the proof rule **IntendMeans** is applicable [1]. It means that the initial BDI configuration C_0 looks as follows (by def. 2.12):

$$C_0 = \Big\langle \big\{ \ldots, \langle +!p(\mathbf{t}), [\rho_1 \ddagger \ldots \ddagger \rho_z]\rangle, \ldots \big\}, C_B, C_I \Big\rangle,$$

i.e., the set of events contains an event $e = \langle +!p(\mathbf{t}), [\rho_1 \ddagger \ldots \ddagger \rho_z]\rangle$. Furthermore, the plan library must provide a variant of the rule $\rho = +!p(\mathbf{s}) : \psi \leftarrow h_1; \ldots; h_n$, and there must exist the most general unifier η of $p(\mathbf{t})$ and $p(\mathbf{s})$ such that $p(\mathbf{t})\eta = p(\mathbf{s})\eta$. Moreover, there must exist another substitution θ satisfying $C_B \models \psi\eta\theta$. The operation semantics of the proof rule **IntendMeans** dictates that C_0 should be evolved into C_{im} ($C_0 \vdash C_{im}$) such that:

$$C_{im} = \Big\langle \big\{ \ldots \big\}, C_B, C_I \cup \big\{ [\rho_1 \ddagger \ldots \ddagger \rho_z \ddagger \rho]\eta\theta \big\} \Big\rangle.$$

[1] It was not specified in the operational semtantics, but when the BDI interpreter tries to invoke the proof rule which is not applicable, no operation is performed, i.e., $C \vdash C'$ such that $C = C'$.

Note that the computational step $C_0 \vdash C_{im}$ defines an internal step since $\mathcal{O}^L(C_0) = \mathcal{O}^L(C_{im})$. An internal step can be simulated by arbitrary number of internal steps. I will show that $C_0 \vdash C_{im}$ can be simulated by π_{im}, which is an update yielded by τ_{im}.

The translation function $\mathfrak{C}$ translates the initial BDI configuration C_0 by identity:

$$\mathfrak{C}(C_0) = \sigma_0 = \Big\langle \big\{ \ldots, \langle +!p(\mathbf{t}), [\rho_1 \ddagger \ldots \ddagger \rho_z] \rangle, \ldots \big\}, \sigma_B, \sigma_I \Big\rangle.$$

Since the proof rule **IntendMeans** is applicable in C_0, the update π_{im} yielded by τ_{im} must be applicable as well, i.e., the condition of τ_{im} must be satisfied. Recall the form of τ_{im}:

$$\tau_{im} \;=\; \big(\models_E \langle te, i \rangle \theta_E \wedge \models_B \psi\theta_E\theta_B \wedge \models_P \rho\theta_E\theta_B \big) \longrightarrow \big(\oplus_{I_{PUSH}}(i, \rho)\theta_E\theta_B \circ \ominus_E \langle te, i \rangle \theta_E \big),$$

and $\rho = te : \psi \leftarrow h_1; \ldots; h_n$. The query operator $\models_E$ instantiates the event $\langle te, i \rangle$ and stores the instances into θ_E. Note that θ_E corresponds to η, i.e., $\theta_E \sim \eta$. Furthermore, $\models_P$ states that KR module $\mathcal{M}_P$ can provide a plan of the form $\rho = +!g : \psi \leftarrow h_1; \ldots; h_n$. Since $C_B \models \psi\eta\theta$, it must also state that $\sigma_B \models \psi\theta_E\theta_B$. See that $\theta_B \sim \theta$. Hence, the condition of τ_{im} is satisfied, and σ_0 can be updated according to this mst:

$$\sigma_{im} = \sigma_0 \bigoplus \pi_{im}.$$

Recall that π_{im} represents the update of τ_{im}. The transformation of σ_0 to σ_{im} is done in two phases since τ_{im} contains a sequence of two consecutive updates. First, the plan ρ is pushed onto the top of an intention i. This is performed by the update operator $\oplus_{I_{PUSH}}$ (specified in def. 5.7). In fact, $\oplus_{I_{PUSH}}$ is specified as:

$$I \oplus_{I_{PUSH}} (i, \rho) = I \cup \{ [\rho_1 \ddagger \ldots \ddagger \rho_z \ddagger \rho] \xi \}$$

where ξ is provided substitution for plan ρ. In this case: $\xi = \theta_E\theta_B$. The second update operator $\ominus_E$ removes the event $\langle te, i \rangle \theta_E$ from the set of events. When both these updates are applied, the resulting mental state σ_{im} looks as follows:

$$\sigma_{im} = \sigma_0 \bigoplus \pi_{im} = \Big\langle \big\{ \ldots \big\}, \sigma_B, \sigma_I \cup \big\{ [\rho_1 \ddagger \ldots \ddagger \rho_z \ddagger \rho] \theta_E\theta_B \big\} \Big\rangle.$$

Since the translation function $\mathfrak{C}$ maps BDI configurations to mental states by identity, and since $\theta_E \sim \eta$ and $\theta_B \sim \theta$, it's easy to see that

$$\sigma_{im} = \mathfrak{C}(C_{im}).$$

ExecAch: The next invoked proof rule in $\bar{\alpha}$ sequence is **ExecAch**. It means that the

BDI configuration C_{im} must contain an intention j (by def. 2.13) with the achievement goal as an actual item, i.e.,

$$C_{im} = \Big\langle C_E, C_B, \{\,\ldots, j, \ldots\,\} \Big\rangle$$

such that

$$j = [\rho_1 \ddagger \ldots \ddagger \rho_{z-1} \ddagger (e : \phi \leftarrow !p(\mathbf{t}); h_2; \ldots; h_n)].$$

C_{im} is then evolved into C_{ach} in a following way (by def. 2.13):

$$C_{ach} = \Big\langle C_E \cup \{\langle +!p(\mathbf{t}), j\rangle\}, C_B, \{\,\ldots\,\} \Big\rangle.$$

Obviously:

$$\sigma_{im} = \mathfrak{C}(C_{im}).$$

Next operation after τ_{im} execution in $\tau_{overall}$ is another test ($\models_I (i, g)\theta_I$) for the intention. We know, that there must be an intention i with its actual item g on its top since $\models_I (i^{\exists}, g^{\exists})\theta_I^{\exists}$ succeeded, and τ_{im} does not decrease the number of intentions. Hence, the concrete instances of i and g are recorded into θ_I (such that $\theta_I = [i \mapsto j,\ g \mapsto !p(\mathbf{t})]$), and the condition $\models_I (i, g)\theta_I$ is evaluated to `true`, and the execution of $\tau_{overall}$ may continue to the first mst (τ_{ach}) of the sequence denoted as *intention execution* (see def. 5.15). Mst τ_{ach} is of the following form (def. 5.10):

$$\tau_{ach} \;=\; \tau_{ach}(\theta_I) \;= \models_{I_{ach}} g\theta_I \longrightarrow \big(\, \oplus_E \big\langle +g, i\big\rangle\theta_I \;\circ\; \ominus_I i\theta_I\big).$$

The condition $\models_{I_{ach}} g\theta_I$ is satisfied since $g\theta_I = !p(\mathbf{t})$ is an achievement goal (see def. 5.7). The mental state σ_{im} is transformed by the update of τ_{ach}. Formally:

$$\sigma_{ach} = \sigma_{im} \bigoplus \pi_{ach},$$

where π_{ach} denotes the compound update of τ_{ach}. See what happens: a new event

$$\big\langle +g, i\big\rangle\theta_I = \big\langle j, +!p(\mathbf{t})\big\rangle$$

is issued by the update operator $\oplus_E$, and the intention

$$i\theta_I = j$$

is suspended (i.e., moved to the event) by $\ominus_I$. The resulting mental state σ_{ach} then looks as follows:

$$\sigma_{ach} = \Big\langle \sigma_E \cup \{\langle +!p(\mathbf{t}), j\rangle\}, \sigma_B, \{\,\ldots\,\} \Big\rangle$$

and again, since $\mathfrak{C}$ maps BDI configurations to mental states by identity, it yields that

$$\sigma_{ach} = \mathfrak{C}(C_{ach}).$$

See, that while the invocation sequence $\bar{\alpha}$ finished, the mst $\tau_{overall}$ did not since four more msts (tree in the *intention execution part*, and one in the *cleaning part*) of $\tau_{overall}$ are waiting for the execution. In the following, I will show that $\tau_{overall}$ did not affect σ_{ach}, i.e., only **skip** updates are yielded. In other words, I will show that the BSM evolution cycle $R_e^{\bar{\alpha}}$ has the form:

$$R_e^{\bar{\alpha}} = \sigma_0 \xrightarrow{\pi_{im}} \sigma_{im} \xrightarrow{\pi_{ach}} \sigma_{ach} \xrightarrow{\mathbf{skip}} \sigma_{cse} \xrightarrow{\mathbf{skip}} \sigma_{test} \xrightarrow{\mathbf{skip}} \sigma_{action} \xrightarrow{\mathbf{skip}} \sigma_{cis},$$

and hence

$$\sigma_{ach} = \sigma_{cse} = \sigma_{test} = \sigma_{action} = \sigma_{cis}.$$

It is obvious by inspecting of conditions of all remaining msts, i.e., $\tau_{cse}, \tau_{test}, \tau_{action}$ and τ_{cis}. All these msts check the actual item provided by θ_I. We know, that g is an achievement goal $!p(\mathbf{t})$ since no condition can be satisfied, and **skip** is yielded. Note that **skip** update does not modify the mental state, i.e., the computational step induced by this update is internal. □

I proved that the BDI derivation generated by the admissible sequence $\bar{\alpha}$ can be simulated by $\tau_{overall}$. The same must be proven for remaining three sequences ($\bar{\beta}, \bar{\gamma}$ and $\bar{\delta}$).

Theorem 5.2: *The behaviour of $\tau_{overall}$ can be simulated by the AgentSpeak(L) program that generates a valid BDI derivation.*

Proof: In the following text, I have to show that each sequence S_σ of mental states (representing the BSM evolution cycle) generated by $\tau_{overall}$ can be translated into the sequence of BDI configurations S_C such that S_C is a valid BDI derivation since we know that each AgentSpeak(L) program can generate only such sequences.

So, let's assume that $\sigma_0 = \langle \sigma_{0_E} \sigma_{0_B} \sigma_{0_I} \rangle$ is an initial mental state such that there is an intention $i' \in \sigma_{I_0}$ of the form $i' = [\rho_1 \ddagger \cdots \ddagger (te : \psi \leftarrow h; \ldots; h_n)]$. It means that the condition $\models_I$ in the very beginning of $\tau_{overall}$ can be evaluated to `true` since $\theta_I^\exists = [i^\exists \mapsto i', g^\exists \mapsto h]$. The first of all the underlying msts (τ_{ach}) can be executed. We know (from def. 5.9) that to satisfy the condition in τ_{ach}, there must be an event $\langle te, i \rangle$ in σ_{0_E}, and the KR module $\mathcal{M}_P$ must provide a variant of plan $\rho = te : \psi \leftarrow h_1; \ldots; h_n$

such that the plan is matched with the event through θ_E, and the context of the plan is a logical consequence of belief base, i.e., $\sigma_{0_E} \models_B \psi$. σ_0 is then transformed into σ_{im} such that

$$\sigma_{im} = \sigma_0 \bigoplus \pi_{im},$$

where π_{im} represents the sequence of updates in τ_{im}. Such transformation involves pushing the plan ρ at the top of i, and deleting the event from σ_{0_E}. The resulting mental state σ_{im} has the following form:

$$\sigma_{im} = \sigma_0 \bigoplus \pi_{im} = \Big\langle \sigma_{0_E} \setminus i', \sigma_B, \sigma_I \cup \big\{ [\rho_1 \ddagger \ldots \ddagger \rho_z \ddagger \rho]\theta_E\theta_B \big\} \Big\rangle.$$

The BDI configuration C_0 that corresponds to σ_0 can determined by the function $\mathfrak{C}$:

$$\sigma_0 = \mathfrak{C}(C_0).$$

Since $\mathfrak{C}$ maps BDI configurations to mental state trivially, C_0 can state as follows:

$$C_0 = \Big\langle \big\{ \ldots, \langle +!p(\mathbf{t}), [\rho_1 \ddagger \ldots \ddagger \rho_z] \rangle, \ldots \big\}, C_B, C_I \Big\rangle$$

Previously mentioned assumptions allow the proof rule **IntendMeans** to be applied, and derive a new configuration C_{im} by def. 2.12:

$$C_{im} = \Big\langle \big\{ \ldots \big\}, C_B, C_I \cup \big\{ [\rho_1 \ddagger \ldots \ddagger \rho_z \ddagger \rho]\eta\theta \big\} \Big\rangle.$$

By comparing of σ_{im} and C_{im}, we have that

$$\sigma_{im} = \mathfrak{C}(C_{im}).$$

See the form of $\tau_{overall}$ (def. 5.15). At this point, the query operator $\models_I (i, g)\theta_I$ is evaluated. We know, that the set of intentions is not empty (since the first query operator must have been evaluated to `true`), thus there are four possibilities that g may acquire (an achievement goal, `true`, a test goal or an action). Let's discuss the first option, the rest is similar. When g is an achievement goal, then the condition in τ_{ach} is satisfied (see def. 5.9), the the update of τ_{ach} can be executed. Thus, the event $\langle +g, i \rangle$ is issued, and i is suspended. And again, since g is an achievement goal, the conditions of remaining msts cannot be satisfied (**skip** is yielded), and hence the form of the evolution cycle R_e is following:

$$R_e = \sigma_0 \xrightarrow{\pi_{im}} \sigma_{im} \xrightarrow{\pi_{ach}} \sigma_{ach} \xrightarrow{\textbf{skip}} \sigma_{cse} \xrightarrow{\textbf{skip}} \sigma_{test} \xrightarrow{\textbf{skip}} \sigma_{action} \xrightarrow{\textbf{skip}} \sigma_{cis}.$$

Note that C_{im} evolves into C_{ach} by application **ExecAch** such that $\sigma_{ach} = \mathfrak{C}(C_{ach})$ under the same circumstances. □

Chapter 6

Discussion

In the previous chapter, I proved that BSM has at least the same expressive power as AgentSpeak(L) according to the definition def. 4.6. All the components of the AgentSpeak(L) agent (belief base, set of events, intention base and the plan library) are translated into KR modules. All the modules are equipped with a set of update and query operators and provide the elementary functionality which is handled by mental state transformer. The design of the modules and mst is performed in order to simulate the actions of the BDI interpreter. The interpreter is responsible for the selection and execution of proof rules. The BDI interpreter presented in [6] implemented a proof rule that deal with the empty intention removal, **CleanStackEntry**. But in [6], the operation semantic for such rule is not proposed, and must have been taken from [3]. It was shown in the previous chapter that one more cleaning rule (**CleanIntSet**) must be added into the proof system.

I think that four KR modules is a natural choice, since AgentSpeak(L) agent is consisted of four parts as well. The correspondence is obvious. However, the function $\mathcal{C}$ translates all the AgentSpeak(L) agent into the same BSM counterpart. The change in the plan library does not affect the structure of $\tau_{overall}$. One would suggest that this is not a correct translation.

I also tried other configurations of KR modules. The choice of only two KR modules is particularly interesting. Such BSM would contained only modules for events and the belief base. But all my efforts towards to this solution were unsuccessful. The feature of multiple intences and the pseudo-parallelism yielded by the possibility of execution a different intention each cycle of the interpreter give a raise of problems with interleaving. The implementation of interleaving by mst would lead to its factorial expansion. Such result would not be satisfactory at all. And thus I did not followed this direction.

Bibliography

[1] Frank S. de Boer, Koen Hindriks, Wiebe van der Hoek and John J. Meyer. A Verification Framework for Agent Programming with Declarative Goals. *Journal of Applied Logic*, 5(2):277–302, 2007.

[2] Koen Hindriks, Frank S. de Boer, Wiebe van der Hoek and John-Jules Ch. Meyer. Control structures of rule-based agent languages. In *Intelligent agents V, no. 1555 in lecture notes in artificial intelligence*, pages 381–396. Springer-Verlag, 1998.

[3] Koen Hindriks, Frank S. de Boer, Wiebe van der Hoek and John-Jules Ch. Meyer. A formal embedding of AgentSpeak(L) in 3APL. In *Advanced Topics in Artificial Intelligence*, volume 1502, pages 155–166. Springer Berlin / Heidelberg, 1998.

[4] Matthias Felleisen. On the expressive power of programming languages. In *Science of Computer Programming*, pages 134–151. Springer-Verlag, 1990.

[5] Sterling, Leon and Shapiro, Ehud. *The Art of Prolog.* MIT Press, Cambridge (MA), 1986.

[6] Anand S. Rao. Agentspeak(L): BDI agents speak out in a logical computable language. In Walter Van de Velde and John Perram, editors, *Agents Breaking Away*, volume 1038 of *Lecture Notes in Computer Science*, pages 42–55. Springer Berlin / Heidelberg, 1996.

[7] Anand S. Rao and Michael P. Georgeff. BDI agents: From theory to practice. In *Proceedings of the first international conference on Multiagent Systems (ICMAS-95*, pages 312–319, 1995.

[8] Bordini, R. H. and Moreira, A. F. Proving BDI properties of agent-oriented programming languages : the asymmetry thesis principles in agentspeak(l). 2004.

[9] Egon Bürger. Abstract state machines: A method for high-level system design and analysis, 2003.

[10] Hindriks, Koen and Novák, Peter. Compiling GOAL agent programs into Jazzyk Behavioural State Machines. In *Proceedings of the 6th German conference on Multiagent System Technologies*, MATES '08, pages 86–98, Berlin, Heidelberg, 2008. Springer-Verlag.

[11] Konstantin Vikhorev, Natasha Alechina, and Brian Logan. Agent programming with priorities and deadlines. In Liz Sonenberg, Peter Stone, Kagan Tumer, and Pinar Yolum, editors, *AAMAS*, pages 397–404. IFAAMAS, 2011.

[12] Peter Novák. Jazzyk: A programming language for hybrid agents with heterogeneous knowledge representations, 2009.

[13] Peter Novák. An open agent architecture: Fundamentals. In *Technical report IfI-07-10, Department of Informatics, Clausthal University of Technology*, 2007.

[14] Peter Novák. Programming multi-agent systems. chapter Jazzyk: A Programming Language for Hybrid Agents with Heterogeneous Knowledge Representations, pages 72–87. Springer-Verlag, Berlin, Heidelberg, 2009.

[15] Peter Novák and Wojciech Jamroga. Code patterns for agent-oriented programming. In *AAMAS (1)*, pages 105–112, 2009.

[16] Rafael H. Bordini, Jomi Fred Hübner, and Michael Wooldridge. *Programming multi-agent systems in AgentSpeak using Jason.* Wiley series in agent technology, 2007.

[17] R.H. Bordini, A.L.C. Bazzan, R. de O. Jannone, D.M. Basso, R.M. Vicari, and V.R Lesser. AgentSpeak(XL): Efficient Intention Selection in BDI Agents via Decision-Theoretic Task Scheduling. In *Proceedings of 1st International Joint Conference on Autonomous Agents and Multi-Agent Systems*, volume 3, pages 1294–1302, Bologna, Italy, July 2002. ACM Press.

Appendix A

Contents of attached CD

Attached CD contains the PDF version of this thesis.