BDI4MABS: A BDI programming approach for high-performance multiagent-based simulation

Benjamin C. Herd

Exeter College
University of Oxford

A dissertation submitted for the
MSc in Software Engineering

Abstract

Agent-based simulation has shown great success for the study of complex adaptive systems which are hard or even impossible to analyse using conventional analytical techniques. At present, agent-based simulation models are typically implemented in conventional, general purpose programming languages such as Java or C++ which severely limits the level of behavioural sophistication that a non-technical modeller is able to achieve. In the more general field of multiagent systems, agent-oriented programming has emerged as a powerful paradigm for the implementation of intelligent, practically reasoning agents. However, current agent-oriented programming languages tend to prioritise expressivity over performance which critically limits their application in a time-critical simulation context.

The goal of this work is to address this problem and develop a first version of an *agent-oriented programming approach for the purpose of high-performance simulation*. This involves the development of an *efficient* and *customisable C++-based BDI framework* together with an *agent-oriented programming interface* that allows for the implementation of BDI-based simulation models on a high level of abstraction. The balance between efficiency and convenience of development — the core challenge of the project — is to be achieved by utilising modern C++ template metaprogramming techniques.

Acknowledgements

- I would like to express my sincere gratitude to my supervisor, Professor Jeremy Gibbons, for the excellent supervision and his helpful comments during the preparation of this dissertation. I would also like to thank Professor Michael Wooldridge for his kind feedback and for showing an interest in my work.

- I am very grateful to Mr Andrew Skates who has always been supportive of my studying ambitions and who made this part-time degree possible in first place.

- The time at Oxford wouldn't have been the same without my classmates. I would like to thank Adrian, Jason, Marcel, Mario, Mike, Richard, and everyone else who helped to make the Oxford experience perfect.

- Doing a part-time degree would be impossible without strong family support. I am sincerely grateful to Julia for her endless patience and support during the last couple of years. I promise, I will never do two degrees and a full-time job in parallel again. Finally, I am greatly endebted to my parents for always standing by me, believing in me and backing all of my decisions, no matter how crazy they were. I wouldn't be even close to where I am without your love and unconditional support.

Declaration

The author confirms that:

- this dissertation does not contain material previously submitted for another degree or academic award; and
- the work presented here is the author's own, except where otherwise stated.

Contents

1 Introduction

Agent-based modelling and simulation (ABS) has shown great success for the analysis of complex and adaptive systems. As opposed to conventional analytical modelling which typically relates *macro level* variables using mathematical equations, ABS puts a particular emphasis on the *micro level*. It represents a *bottom-up* approach that uses populations of virtual *agents* that act and interact in an arbitrarily complex and dynamic environment. As opposed to pure macro-level modelling which has to pose implicit restrictions such as *homogeneity*, *full connectivity* or *full rationality* on the representation of individuals in order to be analytically tractable, ABS naturally allows for aspects such as *heterogeneity*, *autonomy*, *adaptation*, and *locality*. Due to its individual focus, ABS has shown great success for the analysis of complex problems in a variety of domains ranging from economics, geography, epidemiology, or psychology to logistics or business process modelling.

A central problem that severely limits the complexity of contemporary agent-based models is the level of behavioural sophistication. Simulation-based agents are traditionally rather 'myopic' in nature, i.e. their behaviour is largely pre-determined and based on simple behavioural rules. Albeit an abstract and highly simplified representation of the simulated individuals may be sufficient in many cases, it is certainly not for all of them. Particularly in the area of *social simulation*, where the entities to be simulated are human beings, a sufficiently high level of behavioural sophistication is often required to truthfully represent the real-world entities and reach a sufficient level of validity in the model. Furthermore, in cases where only the *environment* that the agents operate in, but not the actual rules that they follow are known, a higher level of autonomy with respect to the agents' decision making can help to develop more realistic and interesting models.

A computational treatment of intelligence has long been the focus of Artificial Intelligence (AI) research and numerous approaches have been proposed to date. In this work, we are not concerned with what has become to be known as 'strong AI', i.e. with the development of truly *conscious* machines; instead, we are primarily interested in mechanisms to design and develop agents that are capable of acting *autonomously* in an unknown environment. One particularly successful result of years of research in this area is the *BDI (belief, desires, intentions) architecture* whose theoretical origins date back to the philosophical work of Michael Bratman, in particular his work on human practical reasoning [7]. BDI provides a convenient balance between cognitively realistic, yet extraordinarily complex theories of human reasoning, and computationally efficient, yet cognitively overly simplistic representations of human behaviour such as those mostly employed in contemporary ABS. The BDI architecture has shown great success in implementing flexible, autonomous agents and serves as the conceptual basis of numerous dedicated agent-oriented programming (AOP) languages and development frameworks.

Unfortunately, in the ABS community, the BDI architecture is largely unknown. There are two major reasons for that. First, due to its interdisciplinary nature, ABS is largely conducted by non-computer scientists who are in general unfamiliar with the work in the AI community; and, even if they were, very few agent-based modellers would be willing and able to imple-

ment the complex logic of a BDI interpreter themselves. Second, despite its light weight in comparison to more complex, behavioural theories, the BDI architecture is still fairly complex and computationally demanding. It has been developed for the implementation of 'real' embodied agents such as robots or unmanned vehicles rather than for the purpose of simulation, in which case the requirements on execution time and scalability are typically much higher. As a consequence, existing implementations of the BDI architecture have prioritised usability and interoperability with existing software environments over performance, e.g. by using Java as the underlying programming language.

The purpose of this work is to address this problem and bring closer together the areas of AOP and ABS. This is to be achieved by providing an implementation of the BDI architecture that is tailored to the specific requirements of ABS, in particular *efficiency*, *flexibility*, and *usability*. A solution is *efficient* if it provides a high level of performance and if it can be integrated into an existing simulation model without much effort; it is *flexible* if it does not force the simulation programmer to be constrained to the BDI architecture in its full functional complexity exclusively, but allows him to vary the level of an agent's cognitive sophistication depending on his current needs; and, it is *usable* if it provides a convenient interface to the programmer which hides most of the functional complexity of the underlying BDI interpretation cycle from him. In summary, the dissertation makes the following contributions:

1. We provide a *C++ implementation of an efficient and customisable BDI framework* for the purpose of high-performance ABS.

2. The BDI framework is extended with an *embedded domain-specific language (EDSL)* that facilitates the formulation of the agent logic through a declarative interface.

The work is motivated by the observation that, in the simulation community, software engineering considerations tend to be largely neglected. In particular, impressive advances in techniques for both the development of high performance applications and the development of applications on a significantly higher level of abstraction have not been leveraged to a sufficient extent. The work aims to address this problem by considering the use of *generic programming* techniques, in particular *C++ template metaprogramming (TMP)* for the implementation of a BDI framework. A particular emphasis is to be put on the aforementioned three principles *efficiency*, *flexibility*, and *usability*. TMP represents a powerful paradigm to write 'programs about programs', i.e. to produce program code that itself produces program code at compile time. As opposed to conventional programs that represent *computations with values*, template metaprograms can thus be seen as *computations with types*. An important use case for TMP is the development of *embedded domain-specific languages* which can be processed efficiently. The performance gain comes from the fact that the domain-specific program code can be translated at compile time into a more low-level, significantly more performant program for the purpose of runtime execution. Since the translation happens as part of the usual compilation process, it is completely hidden from the user.

We believe that TMP represents a promising technique to achieve the three desired principles mentioned above. First, by reducing runtime polymorphism to the absolutely necessary, the performance of the application can be kept at a high level. Second, TMP is a core ingredient of C++, supported by all modern compilers, which facilitates its use in an existing development project. Third, TMP can be used to develop convenient, embedded domain-specific languages that get translated into more efficient code at compile time, thus equipping the framework with a convenient interface to the user whilst retaining a high level of performance.

The thesis is structured as follows. We start with the theoretical background necessary for this dissertation in Chapter 2. In particular, we provide a brief overview of agent-based modelling & simulation, the BDI architecture, and the idea of agent-oriented programming including the description of a selection of commonly used languages and frameworks in this area. We further provide a brief introduction to C++ template metaprogramming in general and the Boost Metaprogramming Library (MPL) and Boost Fusion in particular. Chapter 3 contains a comprehensive requirements analysis. Starting with a motivational example, we provide an overview of typical properties of agent-based simulations and their implications on the desired characteristics of the approach to be developed, together with the main technical challenges. Chapter 4 provides the conceptual foundation for the design of the framework through a formalisation of important BDI concepts. Chapter 5 is devoted to the description of the actual framework including its design and implementation. The performance and usability of the framework is evaluated in a case study which is described in Chapter 6. A critical reflection is given in Chapter 7. The thesis concludes with a summary and ideas for future work in Chapter 8.

2 Background

The purpose of this section is to introduce the background necessary for the remainder of this dissertation. We start with an overview of agent-based modelling and simulation in Section 2.1, followed by an introduction to the BDI architecture and the idea of agent-oriented programming in Section 2.3. Section 2.4 is devoted to an introduction to generic programming with a particular focus on C++ templates and template metaprogramming — the two techniques used extensively in this work. Section 2.1 summarises material from [18].

2.1 Agent-based modelling and simulation

One of the great benefits of cheap modern computer hardware is the possibility to use simulation as a virtual testbed for the exploration of various different scenarios [17, 33]. Shannon describes simulation as "the process of designing a model of a real system and conducting experiments with this model for the purpose either of understanding the behavior of the system or of evaluating various strategies (within the limits imposed by a criterion or set of criteria) for the operation of the system" [37]. Computer simulation makes it possible to explore complex real-world scenarios, determine correlations or conduct what-if analyses in an efficient and cost-effective way. This is particularly interesting when real-world experiments cannot be carried out, e.g. because of financial, legal or ethical constraints.

We focus here on the *agent-based modelling and simulation paradigm*[1] which is characterised by its individual, microscopic nature. Agent-based simulation is best understood by contrasting it with more conventional analytical, macroscopic approaches. One popular example is the analysis of disease transmission by means of an *SIR model* [21]. Here, the overall population is segmented into three compartments: *susceptible*, *infected*, and *recovered/removed*. The dynamics of the population are expressed by a set of differential equations that relate the different segments with each other. Additional parameters allow a modeller to incorporate aspects such as birth or death into the model. The SIR model is purely *macroscopic* in nature since it restricts its focus to the *global, aggregate* behaviour of the system and averages out individual differences and interactions in the underlying population. Modelling starts with the identification of macro-level variables (e.g. the number of individuals belonging to a particular population segment) and continues by defining their relationships; macroscopic modelling can thus be considered inherently 'top-down'.

Unfortunately, an approach that is confined to the macro level is not always appropriate. Many real-world complex systems such as social networks, financial markets, or the weather are characterised by *non-linear global behaviour* that *emerges* from the actions and interactions of their constituents. In such a case, macroscopic modelling may not be able to truthfully represent the dynamics of the system under consideration. Agent-based simulations takes a different route and models the individuals within a system explicitly, thereby taking into account aspects

[1] In the remainder of this dissertation, we use the terms 'agent-based modelling' and 'agent-based simulation' interchangeably.

• Business and Organisations – Manufacturing – Consumer markets – Supply chains – Insurance • Economics – Artificial financial markets – Trade networks • Infrastructure – Electric power markets – Hydrogen economy – Transportation • Crowds – Human movement – Evacuation modelling	• Society and culture – Ancient civilisations – Civil obedience • Terrorism – Social determinants – Organisational networks • Military – Command & control – Force-on-force • Biology – Ecology – Animal group behaviour – Cell behaviour – Sub cellular molecular behaviour

Table 2.1: Application areas for agent-based modelling (adapted from [22])

such as *heterogeneity*, *locality*, and *interaction*. Due to its individual focus, the agent-based paradigm can thus be considered inherently 'bottom-up'. It is particularly suited for the analysis of *complex adaptive systems* [28]. According to Parunak *et al.* [31], "agent-based modeling is most appropriate for domains characterised by a high degree of localisation and distribution and dominated by discrete decision. Equation-based modeling is most naturally applied to systems that can be modeled centrally, and in which the dynamics are dominated by physical laws rather than information processing".

In general, an agent-based model consists of a *population of agents*, each of which is equipped with its own, individual behaviour. Agents are situated in an *environment* which they are able to perceive and to act upon. Time typically progresses in *discrete time steps* or *ticks*. In each tick, either a single agent or the whole population (the latter being more common) is updated. During an update step, agents may perceive the environment, perform actions, communicate with each other, and adapt their own behaviour in response to environmental stimuli.

Due to its general applicability, agent-based simulation has been employed successfully in a wide range of different problem domains (see Table 2.1). A comprehensive overview can be found in Macal and North's tutorial [23].

2.2 The BDI architecture

The history of agent-oriented programming dates back to the 1980s when researchers at the Stanford Research Institute (SRI) started to work on computational approaches to the implementation of rational agents. Important in that context is the idea of *practical reasoning* which has its origins in the work of philosopher Michael Bratman [7]. As opposed to already existing approaches such as AI planning and decision theory which view agents as purely rational, computationally efficient reasoners, an important principle underlying Bratman's work was that of *resource boundedness*. To that end, he sought to describe the behaviour of agents by employing

ideas from *folk psychology*.

Bratman's theory of human practical reasoning forms the basis of an agent architecture which became known as *BDI*. As the name indicates, the BDI architecture is based on three central notions: *beliefs*, *desires*, and *intentions*. *Beliefs* represent the knowledge that an agent has about itself, about other agents, as well as about the environment. It is important to emphasise that beliefs need not necessarily represent *true* facts, nor need they be complete. Thus, agents may believe aspects of the world to be true which turn out to be false; for example, a cleaning robot may believe a certain piece of rubbish to be at position X when, in fact, it is at location Y. Furthermore, agents may not be aware of certain facts, simply because they have not yet learned about them. For example, the cleaning robot may not be aware that there is a piece of rubbish in its proximity, simply because it is beyond its field of vision. *Desires* (or *goals*) represent states of affairs that the agent wants to bring about. For example, the cleaning robot may have the desire to be at the location where it expects the piece of rubbish to be in order to be able to collect it. As opposed to more classical AI approaches which focus on planning from first principles, BDI assumes the plans to be pre-defined by the programmer. In order to be flexible enough, each plan may contain subgoals as part of its body. Those subgoals are replaced by appropriate plans as soon as the execution reaches that particular subgoal.

The central dynamic concept in the BDI architecture is *deliberation*. As opposed to purely reactive agents, deliberative agents have a symbolic representation of the world which they can use to reason about their actions. The central tasks of the deliberation process are (i) deciding which of the desires (or goals) will become *intentions* — i.e. goals that an agent has committed to — and (ii) deliberating how those intentions can be achieved. This gives rise to the following reasoning circle which (maybe slightly modified) forms the basis of all modern AOP languages:

1. The agent updates its belief base (i.e. its symbolic knowledge) based on perception (i.e. sensory input) and internal actions executed previously.

2. Based on its beliefs, the agent identifies new desires or *options*.

3. From the set of options, the agent selects those that it wants to pursue and adds them to its current set of intentions.

4. The agent selects an intention to execute and performs a step. A step can be (i) a belief update, (ii) the execution of an action, or (iii) the generation of a subgoal.

In order to simplify the development of practically reasoning agents, researchers began to work on tailored programming approaches in which the aforementioned notions — beliefs, desires, and intentions — are first-class citizens of the language. The resulting paradigm, *agent-oriented programming*, is still an active field of research in the AI community and described in more detail in the next section. A significantly more exhaustive formal description of the BDI cycle against the background of the logic implemented in the C++ framework developed as part of this work is given in Chapter 4 further below.

2.3 Agent-oriented programming

In 1993, Shoham (also at Stanford) presented *agent-oriented programming (AOP)* [38], a new programming paradigm that intends to facilitate and improve the development process for agent-based systems based on the BDI architecture. Since its first presentation, AOP gained significant influence in the agent community and has been serving as a central paradigm that

underlies many agent programming languages which have been developed since then. Similar to architectures that integrate folk-psychological notions like beliefs, goals and intentions into a conceptual framework, AOP attempts to make these notions an integral part of the programming language itself. As the name suggests, AOP puts a special emphasis on the agent concept as the central component of an application (similar to the concepts of 'classes' and 'objects' in object-oriented programming). Consequently, AOP programs are constructed using agents as their main building blocks. This is a significant extension to the principle of languages like *April* [25], which focus on the development of distributed and agent-oriented applications, yet without themselves providing constructs for agent-specific concepts. In general, an AOP system consists of three major components [45]:

1. A formal language with clear syntax and semantics that provides the logical framework for the definition of agents and their mental state. The specification of mentalistic notions such as beliefs, desires, commitments etc. is done using modal modifiers.

2. An interpreted programming language that bridges the gap between theoretical framework and practice and provides a number of language primitives to write agent-based applications.

3. An 'agentification process' that converts high-level agent programs into executable programs.

In AOP, an agent's mental state consists of beliefs, obligations or commitments and capabilities. Beliefs represent an agent's knowledge about itself and are thus similar to beliefs in the BDI model. Commitments describe an agent's actions and specify its behaviour in a specific situation. They can be communicative actions like *informing*, *requesting* or *offering* (according to *speech act theory* [36]) or private actions. Finally, capabilities define whether an action will be executed or not. If, for example, a robot needs to make a decision about whether to move the arm or not and the robot believes that it is incapable of moving, then it will not make the decision. Capabilities thus have a significant influence on the decision-making process. The reason for Shoham to choose these notions (e.g. instead of *beliefs*, *desires* and *intentions* as in the BDI architecture described above) is that he considers them more basic and more relevant for the concept of agenthood in general and agent-oriented programming in particular. Shoham's paper also defines a generic interpreter for an AOP system. An update cycle in AOP consists of two basic steps: (i) Reading the current messages and updating the mental state, and (ii) executing current commitments which may lead to further mental state updates.

The work at the SRI led to the development of PRS, the *Procedural Reasoning System*, the first computational implementation of the BDI architecture [19]. It was programmed in LISP and contained as its basic components (i) a database containing an agent's beliefs and goals, (ii) a set of intentions, (iii) a library of pre-defined plans, and (iv) an interpreter which performs the selection of plans and their execution. The interpretation cycle of PRS resembles that of the conceptual BDI architecture described above and comprises perception, belief updates, option selection, intention selection, and execution.

PRS has been largely experimental in nature. Nevertheless, it still enjoys significant importance in the area of AOP since it can be considered the first 'real' AOP system and the ancestor of all languages that have been developed since.

Since Shoham's proposal of the AOP paradigm and the development of PRS, a large number of BDI-based, agent-specific programming languages have been developed. In the following section, we provide a description of those AOP languages that we consider most relevant for this dissertation. For a more comprehensive overview, please refer to the literature [24, 3].

2.3.1 dMARS

dMARS, the *Distributed MultiAgent Reasoning System*, was implemented by the Australian Artificial Intelligence Institute (AAII) in Melbourne in the 1990s [15]. Representing an advanced version of PRS, it can be viewed as the first fully-fledged and industry-proof implementation of the BDI architecture. Agents in dMARS are characterised by a plan library and three *selection functions* that programmers can use to customise the dynamics of the interpreter. In dMARS, those functions include (i) the selection of an intention to be executed next, (ii) the selection of a plan to adopt, and (iii) the selection of an event to be handled next. Agents further comprise a belief base, a set of intentions, and a set of events. Events are handled by selecting appropriate plans which, in turn, manipulate the agent's internal goals and beliefs and thus trigger further events. The interpretation cycle of dMARS resembles that of the conceptual BDI architecture described above and comprises the following steps:

1. Selection of an event *e* from the set of events.
2. Identification of *relevant* plans, i.e. those that aim to achieve event *e*.
3. Identification of *applicable* plans, i.e. those that match the current context (see Section 2.3.2).
4. Selection of one of the applicable plans for execution.
5. Creation of a new intention from the plan instance (in the case of external events) or addition of the chosen plan instance to the intention stack (in the case of internal events).
6. Execution of the next action from the body of the topmost plan on the intention stack.

dMARS provides a sophisticated platform for the development of distributed multiagent systems. The dMARS ecosystem comprises numerous components such as a graphical editor, a compiler, an interpreter for a logic programming language, several programming libraries, a multithreading package, as well as a communication subsystem. dMARS was implemented in C++ and was used successfully in many real-world development projects in areas such as diverse as factory automation, simulation, and business and air traffic control systems, and in collaboration with notable companies such as NASA, AirServices, Thomson Airsys, and Daimler Chrysler [16]. A formal operational semantics of an idealised dMARS system has been presented by d'Inverno and Luck [12].

2.3.2 AgentSpeak(L)

One of the most influential AOP languages is AgentSpeak(L) [34]. The general motivation behind its development was a lack of coupling between theoretical, conceptual frameworks like BDI and current implementations like PRS. Despite being a realisation of BDI, PRS was developed using a more general-purpose language (LISP) and thus had no strict and formal underlying BDI-bounded semantics. AgentSpeak(L) is an attempt to bridge the gap between theory and practice by providing a formal, semantically well-defined programming language for the development of BDI-based agent systems.

Formally, an AgentSpeak(L) program consists of beliefs, desires (or goals), triggering events and actions. Programs in AgentSpeak(L) are written using a subset of first-order logic. The syntax is similar to conventional logic programs (e.g. written in Prolog) and augmented by a number of symbols in order to express achievement, testing, sequencing and implication.

Listing 2.1: Example of AgentSpeak(L) plans

```
+concert(A,V) : likes(A)
  <- !book_tickets(A,V).

+!book_tickets(A,V) : ~busy(phone)
  <- call(V);
     ...
     !choose_seats(A,V).
```

An agent in AgentSpeak(L) is specified by its knowledge base (beliefs), its goals and a list of available plans. A plan consists of a *head* and a *body*; its general form is given below:

```
event : context <- body.
```

A triggering event specifies when the plan is invoked (e.g. when a belief or desire has been added or removed). A plan that matches a given event is called *relevant* for that event. Plans are further *context-sensitive*, that is, specific conditions need to be fulfilled in order for a plan to be executable. Conditions are specified within the *context* section of the plan's head. A plan that both matches a given event and satisfies the context conditions is called *applicable*. Finally, the body specifies the single steps that need to be executed when the plan is being pursued. These steps can be either actions or the addition or removal of further facts and desires to an agent's knowledge base. A complete formalisation of AgentSpeak(L) using Z has been given by d'Inverno and Luck [13].

Beliefs describe an agent's current state, its knowledge about itself, about other agents, and about the world in general. The facts that represent the goal states the agent wants to bring about are its *desires*. In order to achieve a desired goal, an agent adopts a plan from the library and commits itself to the pursuit of the respective goal. The plans that have been adopted by the agent thus represent its *intentions*. Beliefs, desires and intentions are not represented explicitly through modular formulae; instead, they are ascribed to agents implicitly at design time.

An example of AgentSpeak(L) plans is shown in Listing 2.1 [4]. The first plan is triggered whenever the agent comes to believe that there is a concert by artist *A* at venue *V* (both of which are variables). The context information states that the plan is only applicable if the agent likes artist *A*. The plan body adds another subgoal (denoted by the exclamation mark), namely `book_tickets`, including the artist and venue name. The subgoal addition triggers the execution of a new plan which is given below. As the context denotes, it is only applicable if the phone is not currently busy. The plan contains a number of steps, the first of which is a 'call' action, the last of which is again a subgoal addition that includes the selection of appropriate seats. The example nicely illustrates that AgentSpeak(L) allows for the formulation of agent logic in a declarative way and on a very high level of abstraction.

The deliberation cycle of AgentSpeak(L) programs follows the BDI deliberation scheme described further above. In general, the following steps can be distinguished:

1. *Perceiving the environment* and *updating the belief base* accordingly.
2. *Selecting an event* to handle.
3. Determining all *relevant plans*.

4. Within the set of relevant plans, determining all *applicable plans* and *update intentions* accordingly.

5. *Selecting an intention* for execution.

6. *Executing one step* of the chosen intention and *modifying the intention stack and the belief base* accordingly.

Despite the similarities between an AgentSpeak(L) program and a conventional logic program, there are some differences [34]. First, in logic programming, the head of a rule is similar to a goal in the body of the rule. In AgentSpeak(L), the head of a plan is a triggering event. This allows for the differentiation between plan invocation by knowledge acquisition ('data-driven') and plan invocation by goal acquisition ('goal-driven'). Another important difference is that plans in AgentSpeak(L) are context-sensitive, which is not the case in logic programming.

Several practical interpreters for AgentSpeak(L) or variants thereof have been developed, the most notable of which is *Jason* [6]. Jason represents a hybrid approach which combines the convenience of a declarative, logic-based language with the flexibility of a general-purpose programming language (Java). The agent logic is formulated in a declarative way (as shown in Listing 2.1 above) whereas more complex, procedural logic can be easily formulated in Java and referred to from within the declarative code (e.g. the implementation of the 'call' function in the example above). Apart from the interpreter for AgentSpeak(L), Jason also provides a comprehensive IDE (both as a stand-alone tool and as a plug-in for Eclipse) which facilitates the formulation of AgentSpeak(L) programs through features such as syntax highlighting and auto completion. It further provides convenient debugging and monitoring facilities that help developers to understand the evolution of the agents' mental states or the dynamics inter-agent communications over time.

Although AgentSpeak(L) and its concrete implementations have mostly been used for the implementation of embodied reasoning agents, it has recently also been suggested as a tool for agent-based simulation development [5]. However, due to its complex internal dynamics, performance remains a critical bottleneck and its usefulness for the implementation of large-scale and performance-critical simulations is thus still limited.

2.3.3 2APL

Similar to AgentSpeak(L) and Jason, 2APL is a programming language based on the BDI architecture [11]. It provides programming constructs for the implementation of different agent concepts and attempts to unify declarative and imperative components. An important principle underlying 2APL is the possibility to implement systems based on various existing agent methodologies without losing the underlying formal semantics. 2APL features a separation between single-agent and multiagent concerns by providing two distinct sets of programming constructs. Those related to single agents consist of constructs to represent beliefs, goals, actions, plans, events and reasoning rules. As described above, 2APL integrates both declarative and imperative principles. Beliefs, goals and actions are described in declarative terms, which allows for automated reasoning as part of an agent's deliberation process. In contrast to that, plans, events and rules are implemented in an imperative way. This facilitates the realisation of more complex procedural logic as well as flow control, recursion and the integration with external imperative or object-oriented environments, e.g. those written in Java or C++.

2APL supports different types of action: *belief update actions*, *belief* and *goal test actions*, *external actions*, *communication actions* and actions to *adopt* and *drop goals*. Plans contain

a list of actions together with additional control statements such as *conditional choice operators* (e.g. `if...then...else`), *iteration operators* (e.g. `while...do`), *sequence operators* or *non-interleaving operators*. This allows for the expression of complex sequences of actions. Reasoning rules can be divided up into three different types. *Planning goal rules* or *PG-rules* can be used to generate plans dynamically based on specific goals and beliefs. It consists of a head, a condition and a body. Head and condition contain the query expressions necessary to test whether given goal and belief statements are true, the body contains a sequence of actions. *Procedure call rules* or *PC-Rules* can be used to specify responses to messages from other agents and external events produced by the environment. Finally, *plan repair rules* or *PR-rules* are used to replace failed plans. They consist of two abstract plan expressions and a belief query expression. If a plan fails, if there is a repair rule whose head can be unified with the failed plan and if the query expression in the repair rule can be satisfied based on the agent's current belief base, then the failed plan can be replaced with the one specified in the repair rule. External environments can be represented in 2APL by writing a class which implements the *environment interface*.

2.3.4 JACK

As opposed to Jason or 2APL, both of which provide declarative languages for the formulation of agent logic, JACK is an extension of Java and therefore entirely imperative in nature [44]. In order to facilitate the implementation of BDI-based deliberative agents at a high level of abstraction, JACK provides an extension to Java — the *JACK Agent Language* (JAL) — which contains new base classes, interfaces, and methods, as well as a range of syntactic extensions. Class extensions comprise the following components:

- `Agent`: represents the reasoning entities within a JACK system.
- `Capability`: encapsulates functional components for the use by agents; it is made up of plans, belief sets, events, and other entities that form an agent's *abilities*.
- `BeliefSet`: represents an agent's beliefs following a generic relational model. Beliefs which can be represented as arbitrary Java data structures generate events every time they are updated. They are designed such that they allow for logic queries like in Prolog.
- `View`: allows for general purpose queries about the underlying data model.
- `Event`: circumstances and messages that an agent can respond to.
- `Plan`: is analogous to a function; plans are executed in response to events.

Syntactic extensions can be classified according to three levels: *class*, *declaration*, and *statement*. On the class level, JAL introduces keywords such as `Agent`, `Plan`, and `Event`. On the declaration level, syntactic constructs which define relationships between the aforementioned classes are introduced. On the statement level, JAL provides statements that can operate on JAL-specific data structures.

As for the semantic extensions, JACK provides a multithreading capability that is handled in the background. Furthermore, JACK incorporates a complex agent reasoning cycle and also provides support for *unification* commonly used in logical programming.

An example code fragment is shown in Listing 2.2. The code shows a stylised implementation of an agent and illustrates the use of base classes (`Agent`), data types (e.g. `BeliefType`),

Listing 2.2: Example agent implementation in JACK

```
agent AgentType extends Agent [implements InterfaceName]
{
// Knowledge bases used by the agent are declared here.

#private data BeliefType belief_name(arg_list);

// Events handled, posted and sent by the agent are
// declared here.

#handles event EventType;
#posts event EventType reference;
#sends event EventType reference;

// Plans used by the agent are declared here.
// Order is important.

#uses plan PlanType;

// Capabilities that the agent has are declared here.

#has capability CapabilityType reference;

// other Data Member and Method definitions
}
```

and special keywords (e.g. `agent`, `#handles`). It becomes clear that, albeit based on conventional Java, JACK manages to raise the level of abstraction and thus provides a nice balance between the convenience of purely declarative AOP languages such as Jason or 2APL and the flexibility of purely imperative, general purpose programming languages such as Java or C++.

As opposed to other AOP approaches which are often experimental in nature, JACK is a commercial framework, developed and maintained by the AOS Group. It is integrated in a wide range of different commercial product packages and comes with extensive documentation and professional support.

2.3.5 Agent-oriented programming and agent-based simulation

Despite various calls for higher cognitive sophistication in agent-based modelling and simulation, the BDI architecture has largely been neglected in the simulation community. The major technical reason is complexity. The complex dynamics in the BDI reasoning cycle render the implementation of simulation models with hundreds, thousands, or even millions of agents largely infeasible. In the following sections, we briefly describe two attempts to integrate BDI-based reasoning into established agent-based modelling frameworks.

BDI for NetLogo

One attempt to bridge the gap between the world of cognitive agents and the agent-based modelling community is BDI for NetLogo [35]. NetLogo is arguably the most popular and most widely used agent-based modelling tool [43]. Besides a convenient and easy-to-learn domain-specific programming language tailored to the needs of agent-based modelling, it provides a completely self-contained IDE with sophisticated visualisation capabilities, an integrated

model library as well as advanced features such as a parameter exploration and an experimentation framework. Due to its high level of abstraction and its flat learning curve, it is particularly popular among modellers without a technical background.

The work by Sakellariou *et al.* [35] attempts to integrate ideas of the BDI architecture into NetLogo. Due to the functional nature of the underlying programming language, lists play an important role in the development of NetLogo models. Each belief is therefore represented as a list containing the *type* and the *content*. The belief type indicates the *class* that the belief belongs to (represented as a simple string) which facilitates belief management. The belief content can be any NetLogo structure, i.e. integers, lists, strings, etc. This is particularly facilitated by the fact that NetLogo lists can be of mixed type. Intentions are also represented as lists with two entries, *intention name* and *intention part*, both of which are strings. Intention names represent function calls including arguments; it is the developer's responsibility to ensure that the functions exist and that they are callable. Intention parts relate to *reporters*, i.e. to functions that return a Boolean value. The intention name is executed until the reporter returns true which corresponds with the idea of goal satisfaction. Once a reporter associated with an intention evaluates to true, the respective intention gets removed from the stack. Since intentions are just functions, they may themselves contain 'add-intention' commands. This realises the idea of subgoals in the BDI architecture.

BDI for Repast

Another attempt to bring closer together the two areas of AOP and agent-based modelling has been made by Padgham *et al.* [30] who integrated JACK (see Section 2.3.4) into Repast, a popular Java-based agent-based modelling framework [9]. According to the authors, the main reason for using JACK instead of developing an entirely new BDI interpreter on top of Repast is to leverage the maturity of the former platform. As a consequence of that, the main focus of the work is on the *synchronisation* between the two platforms rather than on the development of a new approach.

The work makes a distinction between the 'modelling world' containing the simulated environment and the 'cognitive world' containing the BDI-based agents. To this end, the authors subdivide agents into two parts: *sensor-actuator agents* and *reasoning agents*. Sensor-actuator agents are developed in Repast and represent conventional, largely myopic simulation-based agents. Their reasoning counterparts are based on JACK and use the BDI architecture to reason about the world. If an agent has to be both deliberative and able to influence the environment, it consists of both a sensor-actuator and a reasoning component; if it has to be deliberative but only communicates with other agents (and not with the environment), it can be represented as a plain reasoning agent; if no deliberative capabilities are necessary, then a pure sensor-actuator representation suffices.

Whenever a reasoning agents wants to influence the simulated environment, it performs a particular function call — a *BDI action* — to its sensor-actuator counterpart which then makes the actual change to the environment. On the other hand, input from the environment (obtained through perception) that is to be used as part of the deliberation process creates a *BDI percept* which is forwarded from the sensor-actuator agent to the reasoning agent. There are further additional signals between the two layers, mainly for communicating state information; they are not further explained here. BDI execution and agent-based model execution happen in different threads which requires synchronisation. Since simulations are typically based on a discrete time structure, synchronisation can be easily realised on a step-by-step basis.

Listing 2.3: Template types in C++

```
// Class template
template <class T>
class Foo {
  public:
    T getBar() const { return bar; }
  protected:
    T bar;
};

// Function template
template <class U>
U add(U const& v1, U const& v2) {
  return v1+v2;
}

// Variable template
template <class V>
V n = V(5);
```

2.4 C++ template metaprogramming

In C++, template metaprogramming (TMP) describes a technique that uses the concept of *templates*, a feature that allows for the operation with *generic types*, and their manipulation in the compiler to develope 'meta programs', i.e. 'programs about programs'. Before delving more into the details of TMP in Section 2.4.2, it is useful to briefly revisit the idea of *generic programming* since it forms the basis of TMP.

2.4.1 Generic programming

Generic programming describes a programming style in which code is written in a generic, type-independent way. Consider, for example, the manipulation of items in a list, e.g. sorting or removal. The code for the implementation of the actual manipulation routines is essentially independent of the type of items stored within the list (string, integer, float, etc.). Rather than developing different versions of the same algorithm that differ only in the types that they operate upon, generic programming allows for the development of more general code, and thus facilitates reuse and modularity. In C++, generic programming is achieved by means of different types of template: *class templates*, *function templates*, and *variable templates*. Class templates allow for the formulation of classes independent of the type of their members (e.g. member variables or methods); function templates allow for the formulation of functions independent of their argument or return type; variable templates allow for the formulation of families of variables. Examples are shown in Listing 2.3. The class template contains a class member `bar` of an unspecified type together with an accessor function; the function template represents an addition function for two arguments of the same, unspecified type; the variable template specifies a family of numeric variables of different type, all of which are assigned the value 5. Template parameters can be of three different kinds:

1. *type parameters*, e.g. `template <class T> class Foo { ... }`,

Listing 2.4: Calculating the factorial at compile time using TMP

```
template <int N>
struct fact {
  static const int value = N * fact<N-1>::value;
}

template <>
struct fact<1> {
  static const int value = 1;
}

void main() {
  int f = fact<5>::value;
}
```

2. *value parameters*, e.g. `template <int N> class Foo { ... }`, and
3. *template parameters*, e.g. `template <template <class T> > class Foo { ... }`.

Templates are instantiated (thus producing intermediate source code) and merged with the rest of the source code at compile time. Templates can thus only be used in computations where the types can be determined statically, i.e. by the compiler. Typical use cases include the implementation of generic containers as well as algorithms operating upon them. It is thus not surprising that the Standard Template Library (STL), the class library of C++, makes extensive use of generic programming techniques.

2.4.2 Template metaprogramming (TMP)

In 1994, Erwin Unruh of Siemens Nixford showed that, apart from the implementation of generic containers and algorithms such as those found in the STL, templates could also be used to perform actual *computations* at compile time. This gave rise to the idea of template metaprogramming (TMP), a technique to write 'programs about programs'. An interesting aspect is that, rather than being a product of purposeful engineering, TMP has been *discovered* accidentally while the template system was studied during the C++ standardisation process. For space limitations, we can only give a very brief overview of TMP in the paragraphs below; for more detailed descriptions, please refer to the relevant literature [10, 2, 1].

TMP is often referred to as 'programming with types'. A typical (albeit less practically useful) example to illustrate the idea of TMP is the calculation of the factorial of an integer at compile time; one possible implementation is shown in Listing 2.4. The code contains one class template `fact` with a template value parameter of type `int`. The template is subdivided into two cases: a general case which handles arbitrary integer values and a *template specialisation* for the particular case of the parameter being equal to 1. The general case recursively uses the value of the template class instantiated with the decremented current value of the integer parameter. The recursion stops once the base case has been reached. In the main function, `fact` is instantiated by the compiler with 5 as its template parameter. As part of the instantiation, the factorial of 5 (=120) is calculated — at compile time!

In the example, recursion is necessary since TMP does not provide support for mutable variables; as a consequence, looping constructs do not exist. It is important for the understanding

of TMP that templates can essentially be seen as *functions* that accept a set of input parameters (the type parameters) and produce a single output (the instantiated template). When thinking about TMP in that way, it becomes apparent that there exists a strong correspondence with functional programming. In fact, it has been shown that TMP *is* a purely functional, Turing-complete programming language [42], equally expressive as (but arguably significantly less elegant than) Haskell[2]. The functional nature of TMP has interesting consequences. All the abstractions and patterns that functional programmers use — functors, monoids, monads, folds, etc. — are equally applicable and useful in the world of TMP. As in conventional functional programming, they can significantly increase the elegance of the resulting code, as illustrated further below.

One major disadvantage is that the development of compile time computations in C++ requires a significant amount of redundant boilerplate code; furthermore, the cryptic syntax of the template language is anything but elegant which negatively impacts readability and maintainability. As a consequence, several efforts have been made to develop abstractions and create libraries that hide some of the intricacies of the template system behind more convenient interfaces [26]. One of the most popular efforts was the *Loki* library developed by Andrei Alexandrescu [2]. Loki contains a range of useful data structures (most notably *typelists*) and patterns (e.g. singleton, factory, visitor) that TMP programmers can use to develop their applications in a more convenient way and at a higher level of abstraction. The ideas underlying Loki have been hugely popular in the C++ community and have found their way into the famous Boost libraries, most notably into MPL and Fusion, both of which are briefly described in the following sections.

2.4.3 Facilitating TMP: Boost MPL

Similar to Loki, the Boost Metaprogramming Library (MPL) provides a collection of data structures, compile time algorithms, and design patterns with the overall goal of facilitating the development of TMP programs. The data structures and algorithms resemble those from the STL, yet they are purely designed for compile time computation. In a nutshell, the difference between STL and MPL is that the former operates on *values at runtime* and the latter operates on *types at compile time*. As mentioned above, the C++ template system represents a purely functional language; as a consequence, MPL contains several concepts (e.g. folds) that are well known to functional programmers.

Similar to the STL, MPL provides a range of data structures (vector, list, deque, etc.) for storing types. MPL also provides iterators, algorithms to traverse and manipulate data structures (e.g. folding or accumulating/reducing) as well as the concept of a *view* which can be used to develop alternative perspectives on an underlying sequence, without changing the sequence itself[3]. MPL further provides algorithms for type selection (e.g. `if`), arithmetic operations, comparison operations, logical operations, and many more.

As mentioned above, it is essential for the understanding of TMP to view a template as a function that maps types (template parameters) to other types. Despite its static structure, a template thus possesses a dynamic flavour. This forms the basis of the idea of a *metafunction* which is central to MPL. Essentially, a metafunction is just a template that has a particular internal structure. More precisely, a metafunction is a conventional class (or struct) that contains another, templated struct which, in turn defines a new type based on the template parameter. An

[2] It is important to note here that the recursion depth of the C++ compiler is often limited; it can be increased with the help of the compiler option `-ftemplate-depth-X`.

[3] An interesting aspect of views is that they are *lazy*, i.e. they are only computed on demand.

Listing 2.5: A metafunction in Boost MPL

```
struct createVector {
  template <class T>
  struct apply {
    typedef mpl::vector<T> type;
  };
};
```

example of a metafunction is shown in Listing 2.5. In MPL terms, the shown struct represents a function that accepts a template type parameter `T` and returns a vector of type `T` as a result. In order to 'execute' function `createVector` at compile time and obtain the result, we need to define a new type as follows:

```
typedef createVector::apply<int>::type
```

Why is the inner struct `apply` necessary? Why is it not sufficient to templatise the outer class `createVector` and omit `apply`? In general, the inner struct is not strictly necessary. However, the reason for defining the metafunction in that slightly convoluted way is that, if the template parameter only belongs to the inner struct, then `createVector` can be instantiated without having to pass a type parameter. The instantiation of the template can thus be postponed until it is strictly needed (i.e. when `apply` is used); in a sense, this achieves *lazy evaluation* which can be very beneficial for the definition of more elegant code. Nevertheless, the inner struct is not strictly necessary. In many cases, it may be more appropriate to simplify the structure, i.e. to templatise the outer class, to remove the inner struct, and to move the type definition directly into the outer class.

The usefulness of metafunctions is best illustrated with an example. Suppose that we want to create a compile time typelist of vectors, each of a different type. More precisely, we want to create a typelist that contains a vector of integers, a vector of floats, and a vector of strings. Instead of specifying the resulting typelist explicitly, we want to compute it at compile time, starting with a list of 'input types'. This list can be specified as follows:

```
typedef mpl::list<int,float,string> tlist;
```

We can now use the MPL algorithm `transform` to convert the list of basic types into a list of types `vector<int>`, `vector<float>`, and `vector<string>`. `transform` requires as its first parameter the input sequence (`tlist` in our case) and as its second parameter a metafunction that performs the actual conversion from the input type `T` to the wrapped type `vector<T>`. For our example, we can reuse metafunction `createVector` shown in Listing 2.5. The actual transformation can now be described as follows.

```
typedef mpl::transform<tlist,createVector::apply<mpl::_1>> tlist2;
```

Here, `transform` iterates through the elements of its first argument (which is always expected to be a sequence), passes the current element to the metafunction given as second argument, and constructs a new list containing the result of all metafunction calls in sequence. In the metafunction call, '`_1`' acts as a placeholder that can be used to refer back to the element

of `tlist` that is currently being processed. The result of this compile time computation is the new type `tlist2` which is equivalent to the following list.

```
mpl::list<vector<int>,vector<float>,vector<string>>
```

MPL allows for further simplifications. If the metafunction is provided in 'extended form', i.e. with an inner struct that contains the actual type definition (as described above), then the explicit call to the `apply` function can be omitted and the transformation can be simplified as follows:

```
typedef mpl::transform<tlist,createVector> tlist2;
```

MPL provides many more features whose description is beyond the scope of this work. For more information, please refer to the excellent book by Abrahams and Gurtovoy [1]. Nevertheless, despite its power, MPL has its limitations. Those limitations, along with another Boost library that provides solutions for them, are described in more detail in the next section.

2.4.4 Bridging the gap: Boost Fusion

When using the C++ template system, one of the hardest lessons is to make a clear conceptual distinction between computations performed at compile time and those performed at runtime. MPL is entirely focussed on the former. It can be used to perform computations with types by using typelists, compile time algorithms, etc. However, a pure compile time program is rarely of much use. At some point, it is almost always necessary to cross the bridge to the world of runtime execution and to use the computed types productively in a running program. Neither MPL nor STL are very helpful in that respect. For example, consider again the following compile time sequence:

```
typedef mpl::list<int,float,string> tlist;
```

Suppose that we want to create an instance of `tlist` and fill it with concrete values. Apart from `std::tuple`, STL does not support heterogeneous containers; MPL, on the other hand, does not provide algorithms that manipulate its sequences at runtime. The only runtime-related function that MPL provides is `for_each`; it allows for the execution of a templated functor[4] once for each type in a given typelist. For example, we could write a templated output functor and use it in combination with `for_each` in order to output each value of a compile time sequence:

```
// output functor
struct output {
  template< typename U >
  void operator()(U x) {
    cout << x << endl << flush;
  }
};
```

[4]The term *functor* in C++ refers to a function object, i.e. a class that provides an implementation of the function call operator. It should thus not be confused with the two other meanings of the term *functor* used in the functional programming community.

```
// creation and output of a compile-time list of integers
mpl::for_each< mpl::range_c<int,0,10> >( output() );
```

The problem with `mpl::for_each` is that the sequence that it operates upon still has to be constructed at compile time (such as the integer sequence in the example above). In most cases, this is not possible. What we rather want to do is to use a compile time sequence as a starting point, fill it with concrete values at runtime, and then use its values (e.g. output them) at some point in the program execution. This is not possible in MPL. There is a strict boundary between compile time and runtime execution.

Writing code that is executed at different 'times' is also referred to as *multi-stage programming* [41]. Each stage corresponds with the translation of one representation of the program into another. For example, at compilation time, source code is translated into machine code; at runtime, the machine code is executed which, conceptually, corresponds to a translation into some kind of human-readable representation (e.g. a computed value). C++ with its TMP capabilities can basically be considered a three-stage language since it allows for (i) template instantiation, (ii) compilation, and (iii) execution. However, since template instantiation and compilation are indivisible, it is more appropriate to view C++ TMP as 'two-and-a-half stage programming'.

Fusion, another well-known Boost library, addresses the problem of multi-stage programming and provides a bridge between the two distinct worlds of compile time and runtime computation. In terms of its structure, Fusion is similar to MPL. However, as opposed to MPL which is purely focussed on compile time execution, Fusion provides two versions of almost all of its data structures and algorithms: one for compile time execution and one for runtime execution. The compile time versions are largely similar to and compatible with those in MPL. The runtime versions differ significantly — not necessarily in terms of their syntax, but certainly in terms of their semantics.

Consider again the creation of an instance of `tlist` which cannot be done in MPL. Using Fusion, we can do the following:

```
typedef fusion::list<int,float,string> flist;
flist fl(42,3.142,"foo");
```

First, a typelist `flist` is created. This happens at compile time and has the same effect as the equivalent MPL version given above. However, in the second line, an instance of `flist` is created and filled with concrete values. At this point, we are in the world of runtime execution. We can now, for example, write an output function and call the Fusion-based runtime equivalent of `for_each` in order to output the content of `fl` as follows:

```
// output function (functor)
struct output {
  template <class T>
  void operator()(T const& v) const {
    cout << v << endl << flush;
  }
};

// outputs each element of fl
fusion::for_each(fl,output());
```

As mentioned above, STL operates on *values* whereas MPL operates on *types*. The power of Fusion lies in its capability to bridge the gap between the two worlds. This helps to develop programs that are elegant, readable, and maintainable, yet — due to their capability to perform some computations at compile time — very efficient.

Due to its focus on both compile time and runtime computation, Fusion can be seen as superseding MPL. However, MPL is anything but useless. It is optimised for compile time calculation and, as such, should be used for anything that does not rely on any information at runtime such as static type transformations. For the dissertation project, MPL was used for the translation of those parts of the agent logic that constitute the EDSL (e.g. the plans) into more efficient, yet more verbose and thus less conveniently usable counterparts that are then processed at runtime using Fusion algorithms and conventional C++ code. This separation of concerns is a central aspect of the work and is described in more detail in Chapters 3 and 5.

3 Requirements analysis

The purpose of this chapter is provide a thorough requirements analysis for the framework to be developed. We start with a quick overview of the notion of *behaviour trees* in Section 3.1. We further provide two motivational example scenarios, one very simple model and a more complex one from the area of consumer decision making, a typical area of application for agent-based modelling, in Sections 3.2 and 3.3. The examples will be used to identify the features that the framework should provide; they are described in Section 3.4. Finally, Section 3.4.2 contains an overview of technical challenges that need to be overcome in order to realise the desired features.

3.1 Behaviour trees: a quick overview

As described further above, the behaviour of simulation-based agents is typically pre-determined. One particularly convenient way of describing the action cycle that an agent undergoes every tick is to use a tree-based notation. In AI, especially in the area of game development, such a structure is often referred to as a *behaviour tree* [8]. An example (described in further detail below) is shown in Figure 3.1. A behaviour tree shares some similarities with a decision tree, yet there are also important differences. Whereas the latter is used to aid in the decision-making process, the purpose of the former is to *control* the behaviour of an agent. In a behaviour tree, nodes may have meta information associated with them, for example priority values that control the order in which they are executed, as well as conditions that describe under what circumstances the particular node becomes executable. Furthermore, a behaviour tree allows for two different types of control flow nodes: *sequence nodes* and *selector nodes*. As the name suggests, a sequence node denotes sequential execution of all its child nodes (which may, of course, themselves be control flow nodes). Sequence nodes are thus, in a way, equivalent to an AND-gate in digital logic. A selector node, on the other hand, stops the execution as soon as a child node has been executed successfully. It can thus be likened to an OR-gate.

3.2 Scenario 1: Party scenario

The first example model that we are looking at is taken from Padgham *et al.*'s paper [30]; it is shown in Figure 3.1. Here, the overall goal of the agent is to purchase a present for a party. In order to be able to do so, he first needs to get money, either from the ATM or from a friend. Once the money is available, the agent needs to walk to the shop and purchase the present. Getting money from the ATM requires two further actions: walking to the ATM and withdrawing cash; getting money from a friend also requires two actions: walking to the friend's place and collecting the money.

Using behaviour tree terminology, nodes 'Get present' and 'Get money' represent selector nodes, nodes 'Buy present', 'Get money from ATM', and 'Get money from friend' represent sequence nodes, and nodes 'Walk to shop', 'Walk to ATM', 'Withdraw cash', 'Walk to friend',

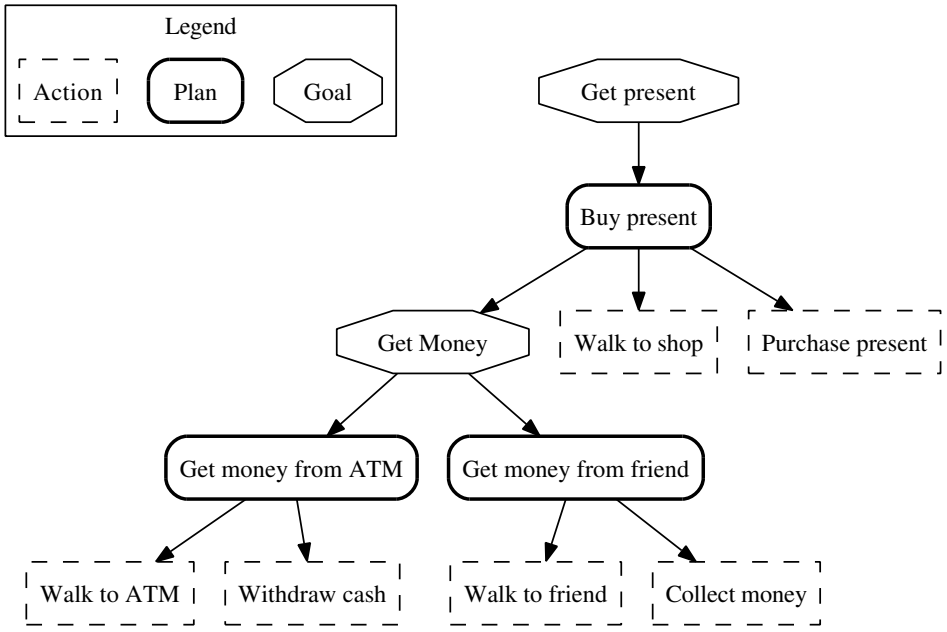


Figure 3.1: Goal-plan tree for a party scenario (adapted from [30])

'Collect money', 'Walk to shop', and 'Purchase present' represent basic actions.

The example nicely illustrates that there is an interesting correspondence between the behaviour tree concept and the BDI architecture. This is reflected in the different node types shown in Figure 3.1 which directly correspond to entities in the BDI architecture — goals, plans, and actions. Essentially, in the example, the selector nodes serve the purpose of goals and the sequence nodes serve the purpose of plans. We see that the agent has an initial goal, 'Get present', for which a single plan is available ('Buy present') which itself consists of a subgoal ('Get money'), and two consecutive actions ('Walk to shop' and 'Purchase present'). For goal 'Get money', two alternative plans are available, one that comprises walking to the ATM and withdrawing cash (both basic actions), and the other one that comprises walking to a friend and collecting money (also two basic actions). We can thus reformulate the program shown in the figure very elegantly using AgentSpeak (Jason) notation as shown in Listing 3.1. This leads to the following first design decision for the framework.

Feature 1 *The framework should allow for the formulation of agent programs in a declarative way, similar to that provided by AOP languages such as Jason.*

The example illustrates a further important aspect of simulation models, namely that of a clear separation between the *declarative* and the *procedural* parts. The given example is purely declarative in nature: we do not know *how* the agent walks to the ATM, *how* it withdraws the money, *how* it purchases the present, etc. Clearly, a concrete simulation model may need additional procedural logic that describes in more detail how those actions are to be performed. Depending on the level of granularity of those actions, they may themselves be described in a partly declarative, partly procedural way. This leads to the next important design decision:

Feature 2 *The framework should support both a declarative and a procedural way of describing the model logic. Both aspects are to be clearly separated. For example, whilst also representing building blocks of declarative plans, actions themselves are procedural in nature and*

Listing 3.1: Formulation of the party scenario in AgentSpeak notation

```
!getPresent.

// buy present
+!getPresent : true <-
  !getMoney;
  walkToShop;
  purchasePresent.

// get money from ATM
+!getMoney : true <-
  walkToATM;
  withdrawCash.

// get money from friend
+!getMoney : true  <-
  walkToFriend;
  collectMoney.
```

may contain arbitrary logic[1].

In the Jason code in Listing 3.1, we assume that both plans that handle the `getMoney` goal addition event are equally applicable (as denoted by the context simply being `true`). In most cases, however, a more complex selection process will be necessary. We may, for example, assume that the second plan (collecting money from a friend) is only applicable if the friend is at home. In this case, the criterion would be simple enough to be specified as part of the declarative agent logic. In most cases, however, the selection will be more complex and be, for example, based on *utility evaluation*. This leads to the following requirement:

Feature 3 *The framework should allow for the specification of a plan selection function in both a declarative (for simple criteria) and a procedural way (for more complex utility calculations).*

3.3 Scenario 2: Selecting a product

One of the main application areas of agent-based modelling is consumer behaviour modelling. A frequently recurring problem in this context is that of selecting from a (potentially large) range of products. Due to our still largely incomplete understanding of the mechanisms according to which people behave when selecting and purchasing products, models in the consumer behaviour domain are often learned from data; the actual decision making process is thus typically represented as a utility evaluation that operates upon the agent's knowledge (or belief) base and selects products *probabilistically*.

An example of a typical product selection function using utility evaluations is shown in Listing 3.2. Here, a set of products (each described by a textual string) is evaluated against two criteria: price and healthiness. The utility ut_i of product i is determined by calculating the normalised weighted sum of price and the healthiness factor:

[1] The same idea has been realised in Jason where actions may contain arbitrary Jason code.

Listing 3.2: Example of a product selection function based on utility evaluation

```
string selectProduct(
  set<string>& products,
  map<string,float>& price,
  map<string,int>& health
) {
  float maxPrice = 100.0, maxHealth = 10.0;

  // calculate and aggregate utility values for all products
  multimap<float,string> utility;
  float total = 0;
  for(auto const& p : products) {
    float ut = (0.7*price[p] + 0.3*health[p]) / (maxPrice + maxHealth);
    total += ut;
    utility.insert(pair<string,float>(total,i));
  }

  // perform utility evaluation and select best product
  uniform_real_distribution<> dist(0,1);
  float rand = dist(rng);
  for(auto it=utilityMap.begin(); it!=utilityMap.end(); ++it) {
    if(rand<(it->first/total))
    return it->second;
  }
  return "none";
}
```

$$ut_i = \frac{weight_p \cdot price_i + weight_h \cdot health_i}{price_{max} + health_{max}} \tag{3.1}$$

Basing the agents' decisions on exhaustive utility evaluations over sets of items (e.g. products) is a common approach in ABS. The utility-based approach is attractive because it is conceptually and mathematically simple, easily implementable, and computationally efficient. From a behavioural point of view, however, the approach is problematic since humans rarely act in such a way. Especially when the set of items under considerations is very large (certain supermarkets offer thousands of products in single categories), exhaustive evaluation is both unrealistic and inefficient. Instead, it would be more appropriate for the agent to successively narrow down its choice based on its personal preferences and accumulate the so-made choices into a *shortlist* which is then evaluated exhaustively. It is a natural consideration to look at the BDI architecture to realise the deliberation process.

This idea is best illustrated using a simple example scenario. We assume a product base of 1,000 products, each associated with three attributes: *availability* (fixed probability of 33%), *price* (random number between 0 and 100), and *healthiness* (random number between 0 (= unhealthy) and 3 (= healthy)). The agent follows a two-step decision-making process, as outlined below:

1. Shortlisting
   (a) Filtering out unavailable products
   (b) Filtering out expensive products (if price $\geqslant 10$)
   (c) Filtering out unhealthy products (if healthiness $\leqslant 1$)
2. Selecting the final product using a utility calculation

During the first step, the overall set of products is narrowed down based on (i) availability and (ii) the agent's preferences with respect to price and healthiness. The second step performs an exhaustive utility evaluation on the remaining shortlist of products.

A behaviour tree representation of the consumer behaviour scenario is shown in Figure 3.2. For simplicity, the first step is depicted as a basic action which hides the actual filtering substeps. The selection step is realised as a subgoal ('Select product') for which there are 1,000 relevant plans — one for each product. It becomes apparent that an explicit representation of products as individual plans may quickly become unmanageable. However, it is interesting to note that the plans only differ with respect to the product that they handle. This suggests the introduction of *parametrised goals, plans, and actions* in order to handle large domains of items in an elegant way.

Feature 4 *The framework should provide a generic way to formulate and manipulate parametrised goals, actions, and plans.*

Another feature that becomes apparent in the example is that the shortlists created as part of the 'Shortlist products' action need to be accessible from within the 'Buy product X' actions. The shortlists form part of the agent's belief base. In order to grant the modeller the highest level of flexibility, the framework should not impose too many constraints on the actual format of the belief base. It is common in ABS to use simple key-value stores (e.g. hash tables) of different type as data structures for the belief base. The framework should not interfere with this convention, but rather allow for the seamless integration of existing data structures.

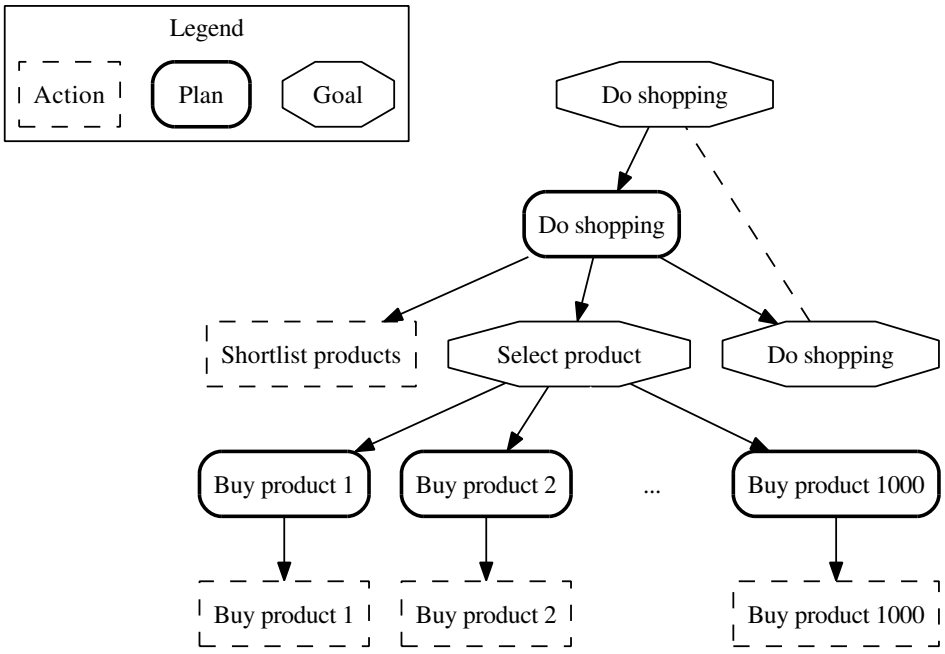


Figure 3.2: Goal-plan tree for a product selection and purchase scenario

Feature 5 *The framework should support the integration of (existing) key-value maps of different types as the data structures representing an agent's belief base into the deliberation process.*

The final characteristic of the consumer behaviour example is that of *proactiveness*. Rather than purely reacting to environmental stimuli, the agent should be able to initiate certain actions autonomously. In the example, this is reflected in the fact that the agent starts a new purchasing decision making cycle proactively every time a product has been purchased. In behaviour tree notation, this is not easily describable. In BDI terms, proactiveness is realised by means of *parallel goals*, i.e. goals that trigger separate intentions which are executed in parallel to the currently active ones. Intuitively, a parallel intention can be seen as being 'in competition' to the already active ones. Until now, all subgoals were such that they had to been satisfied in order for the current intention to be continued. This is not always appropriate. Consider again the example in Figure 3.2. If the agent is supposed to purchase products continuously, then a new purchasing cycle needs to be triggered once a product has been purchased. This can be done by defining a recursive plan, i.e. by making the goal 'Do Shopping' the last element in the body of the plan which itself handles the event 'Do Shopping' (as depicted by the dotted line between the two shopping goals). If the goal is strict (i.e. needs to be satisfied immediately), the result is an infinite loop and the execution of the plan will never end. A solution to this problem is to turn the shopping subgoal into a parallel one. Rather than trying to satisfy a parallel goal immediately once it has been reached during plan execution, a new intention is placed on the stack and interleaved with the current ones. In that way, the plan for the 'old' shopping goal can be finished whilst the plan for the new shopping goal can be started. We can thus formulate the following final requirement:

Feature 6 *The framework should support parallel handling of goals, i.e. the creation and execution of intentions which are executed in parallel to the currently active ones.*

3.4 Summary

In the area of ABS, the BDI framework has been largely neglected to date. The purpose of the previous two sections was to establish a basic correspondence between the ideas of BDI and the way in which simulation-based agents are designed and implemented by example. Due to the huge variety of domains in which ABS is being used successfully and the consequently large number of different simulation models, the analysis is clearly superficial and simplistic. However, rather than being exhaustive, the purpose of the examples is to emphasise certain *typical features* of agent-based simulation models which we believe are necessary for a BDI framework to be useful in a simulation context.

As illustrated in Sections 2.2 and 2.3.2, the BDI architecture is highly complex in nature and the re-implementation of a fully-fledged agent platform such as Jason is beyond the scope of this work. In order to clearly delineate the contribution of this work, we focus on *central aspects* of the agents' internal deliberation mechanism and neglect issues related with inter-agent communication (e.g. message exchange or speech act-based communication). We believe that, since the framework allows for the integration of procedural C++ code, messaging can be easily integrated; it does, however, not form an integral part of its internal semantics and is thus not further described in this thesis.

In the following two subsections, we briefly summarise the features informally introduced in the previous sections and provide an overview of particular technical challenges that need to be overcome in order to realise the features.

3.4.1 Desired features

The features introduced in the previous sections can be summarised as follows.

Modelling beliefs: It must be possible to model an agent's beliefs in a simple, yet flexible way. In particular, it should be easy to integrate key-value attributes of different types into the deliberation process such that they can be easily accessed from within an action.

Modelling goals: It must be possible to model an agent's individual goals in a declarative way. We distinguish between *parametrised* and *unparametrised*, conventional goals. Parametrised goals contain an additional type parameter together with a textual name that can be used to access the value of a particular sequence of values (e.g. a list of products), for each of which an instance of the particular goal is created internally. Unless stated as parallel (see further below), a goal needs to be satisfied immediately in order for the superordinate plan to continue successfully.

Modelling conventional plans: It must be possible to model an agent's available plans in a declarative way. Following the notation of Jason, this includes (i) the *triggering condition* (i.e. the goal), (ii) the *context*, and (iii) the *plan body* (containing subgoals as well as actions).

Modelling parametrised plans: In order to facilitate the formulation of large *option sets*, i.e. large sets of plans which only differ in one particular parameter, it must be possible to specify parametrised plans, i.e. plans that are parametrised with a container of possible options and instantiated automatically.

Modelling actions: It must be possible to model the actions that an agent is capable of performing. In order to provide a high level of flexibility, actions may contain arbitrary

procedural logic; on the other hand, they should be easily integratable into plans in a declarative way. Furthermore, in the presence of parametrised plans, actions themselves also need to be parametrisable.

In order to realise those features, a number of additional concepts are necessary:

Event store: In order to handle (i.e. add, remove, and access) events efficiently, the framework needs to provide an appropriate data structure for their storage.

Intention store: In order to handle (i.e. add, remove, and access) currently active intentions effectively, an appropriate data structure is required.

Selection mechanism: The framework should allow the developer to specify the way in which decisions during the deliberation process are to be made. In particular, we want to support the following three types of selection:

1. **Event selection:** The framework should allow the developer to specify an *event selection mechanism* that acts upon the event store and selects the next event for execution.
2. **Plan selection:** The framework should allow the user to specify a *plan selection mechanism*. This involves both the selection of *relevant plans* and the selection of *applicable plans*. Plan selection should also allow for the integration of *utility functions*, as commonly used in ABS.
3. **Intention selection:** The framework should allow the user to specify an *intention selection mechanism* that acts upon the intention store and selects the next intention for execution.

Parallel execution: The framework should allow for the annotation of plan body items as 'parallel'. Parallel plan body items create a separate intention on the intention stack that is executed concurrently (i.e. interleaved) to the one currently being pursued. In this case, the execution of the current plan can immediately continue.

Finally, the environment that agents are situated in and that they react to plays an important role in the development of an ABS. The environment itself is not part of the deliberation process and its dynamics are thus beyond the control of the BDI architecture. However, since agents need to perceive the environment as part of the BDI deliberation process, an appropriate interface needs to be provided by the framework.

Modelling the environment: It should be possible to integrate the environment in which the agents are situated into the deliberation process. The logic of the environment is fully provided by the developer; due to the high level of flexibility required in this case, the framework should not impose any constraints upon the nature of the environment and allow for its formulation in purely procedural C++ code. Nevertheless, it needs to provide an interface through which it can be perceived by the agents as part of their deliberation process.

Beside the actual features described above, there are a number of cross-cutting technical challenges that need to be overcome. They are described in more detail in the next section.

3.4.2 Technical challenges

In order to achieve the features described in the previous section, a number of technical challenges need to be overcome. In particular, we focus on four main challenges which are described in more detail below: (i) support for efficient, heterogeneous data storage, e.g. belief bases in which beliefs of different types (string, integer, float, etc.) are supported under a common interface, (ii) the design of an efficient declarative interface to the framework which does not compromise on the latter's efficiency, (iii) the integration of both declarative and procedural model logic into the framework, and (iv) the possibility to use both compile time and runtime plan selection in a unified manner. All challenges are described in more detail in the following sections.

Challenge 1: Support for heterogenous data storage

C++ is a statically typed languages and containers such as arrays, lists, or dictionaries are not allowed to contain values of different types. In a real-world simulation model, however, the beliefs of an agent may well be of various types: the agent may know about certain products (represented as text strings), about their prices (represented as floating point values), and about their quantity (represented as integer values). It may, for obvious reasons, encapsulate product-specific information into appropriate data structures such as structs or classes and, for the purpose of efficient retrieval, store them in appropriate containers such as hash maps. C++ is a statically typed language; the types of containers need to be determined at compile time and containers cannot, in general, hold heterogeneous elements. Although it is possible to circumvent this problem, e.g. by storing values in generic data types such as Boost.Any, this is only recommended if performance is not a critical issue (due to the frequent type conversions necessary in this case). As framework designers, we cannot anticipate the simulation models which the BDI framework may be integrated into; consequently, we should not impose any unnecessary constraints upon the nature of the agents' belief bases and allow for the highest level of flexibility. The challenge in this context is to allow a developer to integrate arbitrary data structures into the deliberation process, yet without requiring him to use any pre-defined data structures offered by the framework.

Challenge 2: Declarative interface

As described further above, one of the main benefits of AOP languages such as Jason or 2APL is their declarative nature. In those languages, agent programs can be written conveniently at a high level of abstraction without having to deal with the (fairly complicated) internals of the BDI reasoning cycle. C++ is, by its very nature, not a declarative language. C++ programs offer a high level of performance which, in general, comes at the expense of elegance and maintainability. Current AOP languages can be seen as being situated at the exact opposite of the spectrum: due to their declarative nature, they offer a high level of usability; on the other hand, they are typically not built with high performance in mind and thus largely unusable in a simulation context. The central challenge of this work is to bridge the gap between the two worlds by providing a declarative interface that is convenient to use for a developer, yet without compromising on performance.

Challenge 3: Combination of declarative and procedural logic

A declarative interface such as the one provided by Jason or 2APL is convenient, yet it also imposes strict constraints upon the programmer's flexibility w.r.t. the model logic. In order not

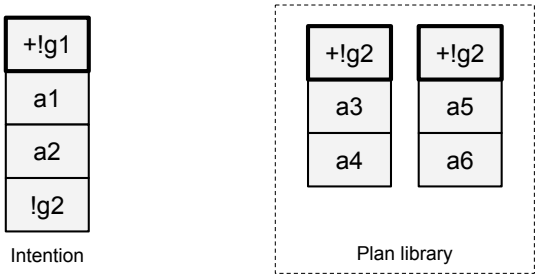


Figure 3.3: Example intention and plan library

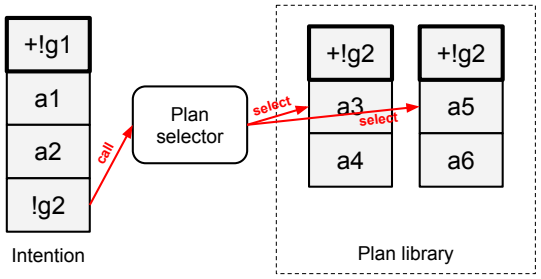


Figure 3.4: Plan selection at runtime

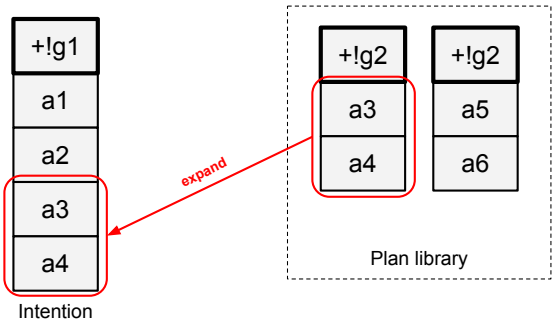


Figure 3.5: Plan expansion and selection at compile time

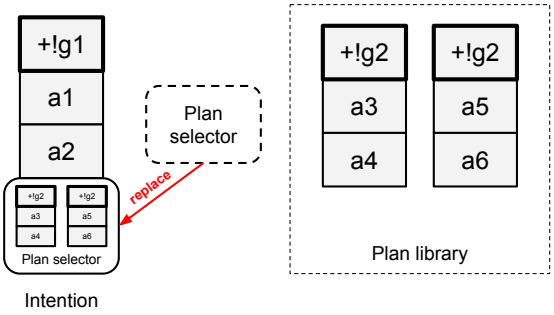


Figure 3.6: Plan expansion and selection at runtime

to limit the level of sophistication of the simulation model unnecessarily, the framework should follow an approach similar to that provided by Jason and allow for the seamless integration of both declarative and procedural code. This is further motivated by the fact that, depending on the particular simulation model to be developed, the focus may be either on declarativity (and simplicity) or on procedurality (and flexibility). In order to provide the highest level of flexibility, neither aspect should preclude the other one, rather should they coexist in a single simulation model — ideally to an easily adjustable extent.

Challenge 4: Support for both compile time and runtime plan selection

An important step during the BDI reasoning cycle is the selection of relevant and applicable plans whose actual shape is clearly dependent upon the problem domain. In some cases, the decision making will be very simple; for example, the developer may decide to always select the first plan; alternatively, he may decide to always select the one with the highest priority; or — which is most likely in the case of ABS — he may decide to perform a more comprehensive calculation in order to determine the utility of each plan, based on which the selection can then take place. Depending on the nature of plan selection, it can be either performed at compile time or at runtime. In the case of always selecting the first plan as well as when selecting plans based on pre-defined priority values, compile time selection is possible and, for efficiency reasons, advantageous. Consider, for example, the example in Figure 3.3. On the left side, the agent's current intention is displayed. It originates from a plan which handles the addition of goal `g1` and comprises within its body two actions, `a1` and `a2`, and a subgoal `g2`. On the right side, the plan library is shown; the agent has two possible plans for achieving `g2`. In conventional BDI interpreters such as Jason, plan selection happens at runtime. Upon encountering a subgoal

during the execution of an intention, a plan selection mechanism is triggered (see Figure 3.4). One of the goals of this work is to shift some of the computation towards compile time. If all the information necessary for plan selection is available at compile time (e.g. if always the first plan is to be chosen), then the subgoal can be replaced by the plan body of the chosen plan at compile time. This approach is shown in Figure 3.5. Here, the runtime overhead is incurred by the two levels of indirection (intention $\rightarrow$ selector $\rightarrow$ plan). In all other cases, i.e. if plan selection depends on runtime information, the selection itself cannot be made at compile time; we can, however, at least remove one level of indirection by replacing the subgoal with an appropriate selection mechanism at runtime. This approach is shown in Figure 3.6.

As soon as the decision depends upon information obtained at runtime, the selection also needs to take place at runtime. The framework should allow a developer to move seamlessly between the two selection modes. This is not trivial since the mechanisms for compile time and runtime selection differ significantly. In order to provide a convenient interface, an additional level of abstraction is therefore necessary.

4 A formal view on the BDI framework

An informal introduction to the BDI framework was given in Section 2.2 above. In order to clarify the concepts discussed in the following chapters and provide a sound basis for the implementation, a significantly extended and formal description of the BDI framework is provided in this section. It is important to note that the formalisation serves purely explanatory purposes and is not intended as a basis for formally deriving an implementation by means of refinement. The goal of the formalisation is to provide an abstract, but formally rigorous description of the BDI dynamics implemented in the framework and described informally in Sections 2.2 and 2.3 above. The formalisation focusses on those aspects that are under the explicit control of the framework; domain-specific aspects such as the internal workings of the environment are thus deliberately omitted. Customisation points are described on an abstract level with a particular focus on the interoperability between the framework and the custom components.

We use the Z notation [40] for the formalisation, primarily because of its rigour and syntactic closeness to a purely mathematical description. An overview of the notation is given in Appendix A. Z has been used extensively for the formal specification of multiagent systems [12, 13, 14, 29]. The work most closely related to ours is that of d'Inverno and Luck [13] who provide a formal specification of AgentSpeak(L). As opposed to d'Inverno and Luck, we focus here on the usage of BDI *in a simulation context* and omit certain BDI-related subtleties such as unification; furthermore, we try to keep the formalisation conceptually close to the subsequent implementation in C++. A formalisation of simulation-based agents has also been presented in [18]. It differs from the formalisation below in a number of aspects. First, the formalisation in [18] views agents on a higher level of abstraction and ignores mental notions such as beliefs, goals, desires, etc.; the formalisation here focusses on BDI-based agents. Second, the formalisation in [18] is intended to serve as a formal model for the *verification* of temporal logic properties; the formalisation below aims to serve as a specification for *implementation*. And, finally, in order to link the specification with a process algebra description, the specification in [18] was formulated in Object-Z [39] — an object-oriented extension of conventional Z.

4.1 States

We start the formalisation with the static components of the system. This comprises both the environment as well as the static components of an individual agent, i.e. its beliefs, goals, intentions, events, and plans.

4.1.1 Environment

As mentioned further above, the logic of the environment is beyond the control of the framework and can be integrated by the framework user in a purely procedural way using conventional C++ code. For the sake of clarity, we thus describe the environment abstractly as a pre-defined set of environmental states.

$[Env]$

4.1.2 Belief base

Beliefs can be of various types, depending on the particular domain that the simulation model is to operate in. In order to abstract away the manipulation of types (e.g. conversions), we introduce two general sets describing all possible belief names and values, respectively.

$$[BeliefName, BeliefValue]$$

An agent's belief base can then be defined as a partial function from possible belief names to possible belief values.

$$BeliefBase == BeliefName \pfun BeliefValue$$

4.1.3 Goals

Similar to belief names and values, goals can also be represented as an abstract set.

$$[Goal]$$

4.1.4 Actions

From the perspective of the BDI interpreter, actions represent atomic entities. Although they act as containers for (arbitrarily complex) procedural code, their internal workings are irrelevant for the dynamics described here. As a consequence, we describe the set of possible actions in a purely abstract manner as follows.

$$[Ac]$$

4.1.5 Events

The next important aspect is the *event*. We distinguish between two types of event: *goal addition* and *goal removal*. Events are represented as algebraic data types. As for the way in which an agent keeps track of multiple events, we use a simple sequence.

$$Event ::= Add\langle\langle Goal\rangle\rangle \mid Remove\langle\langle Goal\rangle\rangle$$
$$Events == \mathbb{F}\, Event$$

4.1.6 Plans

As described above, plans consist of smaller entities such as achievement goals, test goals, or actions, the *plan items*. A plan item is either a *goal addition*, a *goal removal*, or an *atomic action*, and can be represented formally as an algebraic data type. In order to allow for parallel goals, we further distinguish between *sequential* and *parallel goal addition*.

$$PlanItem ::= SeqAddition\langle\langle Goal\rangle\rangle \mid ParAddition\langle\langle Goal\rangle\rangle \mid Removal\langle\langle Goal\rangle\rangle \mid Action\langle\langle Ac\rangle\rangle$$

At this point, it is useful to briefly highlight the difference between goal addition and removal *plan items* and goal addition and removal *events* as described above. Plan items describe building blocks of a plan body; as such, strictly speaking, they do not directly *do* anything. Once chosen for execution, however, they *create* events which the agent can react to. Goal addition and removal plan items can thus be seen as wrappers for the respective event types. In the implementation, this distinction will be largely abandoned for reasons of efficiency; for the formal description, however, we believe that the distinction is useful.

Since the plan items are the entities that an agent is capable of executing at runtime, we also refer to them as *capabilities*. Albeit not directly necessary for the description of plans, the notion of capabilities will become particularly useful for the formulation of constraints on the agent state schema, as described further below. We define an agent's capabilities as follows.

$$Capabilities == \mathbb{F}\, PlanItem$$

A *plan* can then be defined as a (possibly empty) finite sequence of plan items. For convenience, we further introduce an alias name for finite sets of plans.

$$Plan == \mathrm{seq}\, PlanItem$$
$$PlanSet == \mathbb{F}\, Plan$$

An agent's *plan library* can then be defined as a relation between goals and plans. In particular, it is important to allow for there being multiple possible plans for a single goal; what we need is a one-to-many relationship between goals and plans. We can thus model the plan library as a total function from plans to goals.

$$PlanLib == (Plan \longrightarrow Goal)$$

4.1.7 Intentions

Once a plan has been chosen for execution, it becomes an *intention*. Technically, an intention is thus similar to a plan and can be described as a finite set of capabilities. At any point in time, an agent may have multiple intentions. In the literature, they are mostly kept in a stack — the *intention stack*. We choose to represent intentions as a finite set here; due to its LIFO (last in, first out) structure, a stack already prescribes a fixed access scheme which we believe is too implementation-specific at this point.

$$Intention == Plan$$
$$Intentions == \mathbb{F}\, Intention$$

4.1.8 Agent state

We now have all ingredients to describe the state, i.e. the static structure, of an agent. Technically, an agent's state is composed from its beliefs, events, intentions, plans, and capabilities. Furthermore, the agent comprises a number of functions which become relevant during the reasoning process described further below. The only two (intuitive) constraints here are that (i) each plan body may only comprise activities that the agent is capable of, and (ii) each intention may only contain activities that the agent is capable of. The important point to remark here is that plans are only indirectly related with intentions. This is due to the fact that, because of subgoal expansion, intentions are not strictly identical with plans. Nevertheless, intentions may only comprise those plan items that are also part of the original plan. The notion of capabilities comes in handy when formalising this requirement.

Agent

$beliefs : BeliefBase$
$events : Events$
$intentions : Intentions$
$plans : PlanLib$
$capabilities : Capabilities$
$perceive : Env \times BeliefBase \longrightarrow BeliefBase$
$applicablePlans : PlanSet \times BeliefBase \pfun PlanSet$
$selectPlan : PlanSet \times BeliefBase \longrightarrow Plan$
$executeAction : Ac \times BeliefBase \longrightarrow BeliefBase$

$\forall p : \mathrm{dom}\, plans \bullet \mathrm{ran}\, p \subseteq capabilities$
$\forall i : intentions \bullet \mathrm{ran}\, i \subseteq capabilities$

4.2 Operations

During the update step, a set of operations modifies the agent's state. Following the conceptual descriptions in 2.2 and 2.3.2, there are four major steps: (i) perception, (ii) event processing, (iii) plan selection, and (iv) intention execution. The following paragraphs provide a formalisation of those steps by means of operation schemas, along with additional necessary helper functions. All of those operations are then combined into the overall update cycle, as described in Section 4.2.5. For technical reasons, we follow a slightly different order and describe plan selection before event processing and intention selection. This is due to the fact that plan selection represents a building block for both of the latter steps and its formalisation is thus required first. Detailed precondition analyses for all state-changing operations provided below can be found in Appendix B.

4.2.1 Perception

Given the static structure of an agent, we can now define the individual operations that constitute its update cycle. We start with perception. As described above, the internals of the environment are not further specified here; the actual perception is thus denoted by function *perceive* that is part of schema *Agent* described above; it takes the environment and the belief base and produces an updated version of the latter. The operation shown below receives as an argument an instance of the environment which, we assume, is provided externally.

Perceive

$\Delta Agent$
$env? : Env$

$beliefs' = perceive(env?, beliefs)$
$events' = events$
$intentions' = intentions$
$plans' = plans$

4.2.2 Plan selection

An important part of both event processing and intention execution described further below is the selection of a plan for future execution. In order to simplify the description, it is therefore

useful to discuss plan selection first. For clarity, we subdivide the process of plan selection into a number of different schemas which are then combined using schema calculus.

We first define what it means for a plan to be relevant. Given a goal g, the set of relevant plans is the set of all those plans which have g as their triggering event. This is described by the following operation schema.

FindRelevantPlans

$\Xi Agent$
$g? : Goal$
$rp! : PlanSet$

$g? \in \mathrm{ran}\, plans$
$rp! = plans^{\sim}(\!|\{g?\}|\!)$

FindRelevantPlans requires plans for the goal to be handled. We thus also need to handle the case in which there is no plan. For simplicity, we assume there is no special treatment of this case — the agent will simply not do anything.

NoRelevantPlans

$\Xi Agent$
$g? : Goal$

$g? \notin \mathrm{ran}\, plans$

As described in Section 2.3.2 above, the set of applicable plans is obtained by checking the set of relevant plans for their applicability given the agent's current belief base. Since the actual check is custom to the respective simulation model, applicability is modelled on a purely abstract level as a function *applicablePlans* in schema *Agent* mapping from a given set of plans (the relevant plans) and the agent's current belief base to a new set of plans (the set of applicable plans). The actual mechanism according to which applicability is determined is left unspecified.

FindApplicablePlans

$\Xi Agent$
$rp? : PlanSet$
$ap! : PlanSet$

$(rp?, beliefs) \in \mathrm{dom}\, applicablePlans$
$ap! \subseteq rp?$
$ap! = applicablePlans(rp?, beliefs)$

In analogy to the selection of relevant plans, we need to handle the case of no applicable plan being available. For simplicity, we assume that there is no special further treatment of this case and the agent will simply remain passive.

NoApplicablePlans

$\Xi Agent$
$rp? : PlanSet$

$(rp?, beliefs) \notin \mathrm{dom}\, applicablePlans$

In the case of multiple plans being applicable, only a single plan can be chosen for execution. This is the purpose of the *plan selection function* which represents an important instance of *meta reasoning*. Similar to the applicability check, plan selection is a highly custom aspect that will most likely be implemented by the modeller in a procedural way. For example, in a simulation context, modellers may decide to implement utility evaluations in order to perform a final selection from the set of applicable plans. In order to keep things abstract, plan selection is represented by function *selectPlan* in schema *Agent*; it maps from the set of plans and the current belief base to a single plan. The actual mechanism according to which the selection takes place is left unspecified here.

SelectPlan

$\Xi Agent$
$ap? : PlanSet$
$p! : Plan$

$p! \in ap?$
$p! = selectPlan(ap?, beliefs)$

4.2.3 Event processing

After the environment has been perceived, events need to be processed. To this end, the agent picks the first element in the event store. If this element is a goal addition event, then a plan is searched and, if successful, added as a separate intention to the set of intentions. If the element is a goal removal event, then all currently pending addition events of the respective goal are removed from the event store.

As in the case of plan selection described further above, this fairly complex operation is split up into a number of individual steps. We start with the action of successfully picking an event, as described by the following operation schema.

PickEvent

$\Delta Agent$
$e! : Event$

$events \neq \varnothing$
$events' \subset events$
$beliefs' = beliefs$
$intentions' = intentions$
$plans' = plans$
$e! \in events$

The schema is conditional upon the event store not being empty; in order to totalise the resulting operation, we thus also have to handle empty stores. For simplicity, we decide that nothing should be done in this case. This is described by the following schema.

NoEvents

$\Xi Agent$

$events = \varnothing$

Now that an event has been picked, we can describe the actual checks. The next schema describes the check for a goal addition event. The schema takes as an input an event and outputs the goal g that is wrapped in the goal addition constructor $Add(g)$.

IsGoalAdditionEvent

$$e? : Event$$
$$g! : Goal$$
$$(\exists g : Goal \bullet e? = Add(g!))$$

Goal addition further requires the addition of an intention to the intention stack. This is described by the following operation schema.

AddIntention

$$\Delta Agent$$
$$i? : Intention$$
$$beliefs' = beliefs$$
$$intentions' = intentions \cup \{i?\}$$
$$events' = events$$
$$plans' = plans$$

We can now define the overall operation of handling a goal addition using schema calculus. For convenience, we start in reverse order by describing the selection of a plan which becomes an intention and gets added to the intention stack[1].

$$HGA0 \mathrel{\widehat=} SelectPlan[i!/p!] \gg AddIntention$$

The resulting operation is a total operation, i.e. no error case needs to be defined. In the next step, we prepend the search for applicable plans to the previously defined operation. In order to totalise the resulting operation, we also need to consider the case in which no applicable plans are available.

$$HGA1 \mathrel{\widehat=} (FindApplicablePlans \gg HGA0) \lor NoApplicablePlans$$

Similar to the previous step, we prepend the search for relevant plans to the previously defined operation. In order to totalise the operation, we also need to consider the case in which there are no relevant plans.

$$HGA2 \mathrel{\widehat=} (FindRelevantPlans \gg HGA1) \lor NoRelevantPlans$$

The overall operation to handle a goal addition event can then be defined by prepending two further operations — (i) picking an event and (ii) ensuring that the event is a goal addition event — to the previously defined operation as follows.

$$HandleGoalAddition \mathrel{\widehat=} (PickEvent \gg IsGoalAdditionEvent \gg HGA2)$$

This concludes the formalisation of the processing of goal addition events. We can now move on to the description of the goal removal event. Similar to the previous case, we first define a schema that checks whether the chosen event is a goal removal event.

[1] In order for the piping operator ($\gg$) to work, input and output arguments of the respective schemas need to match. To this end, the output argument $p!$ of schema *SelectPlan* is renamed to $i!$.

IsGoalRemovalEvent

$\Xi Agent$
$e? : Event$
$g! : Goal$

$(\exists g : Goal \bullet e? = Remove(g!))$

The operation schema is total, i.e. we do not need to handle any special error cases. We can now define the removal of all goal-related addition events from the overall set of events.

RemoveEvents

$\Delta Agent$
$g? : Goal$

$beliefs' = beliefs$
$intentions' = intentions$
$events' = events \setminus \{Add(g?)\}$
$plans' = plans$

The overall processing of goal removal events can then be described as the following sequence of operations:

$$HandleGoalRemoval \mathrel{\widehat=} (PickEvent \gg IsGoalRemovalEvent \gg RemoveEvents)$$

We now have all ingredients for the formalisation of the overall event processing operation. Intuitively, the processing of events constitutes (i) the processing of goal addition, (ii) the processing of goal removal, or (iii) no activity because of an empty event store. Using the operation schemas defined above, this can be formalised as follows.

$$ProcessEvents \mathrel{\widehat=} HandleGoalAddition \lor HandleGoalRemoval \lor NoEvents$$

4.2.4 Intention execution

During the next step, the agent first picks an intention i for execution, picks the first element e of i, and executes e. If there are no intentions, nothing happens. This is described by the following operation schema.

NoIntentions

$\Xi Agent$

$intentions = \varnothing$

If the set of intentions is not empty, then the agent first picks an intention. This may happen in a variety of ways: the agent could pick the first intention, it could make a random choice, use a more complicated prioritisation for selection, etc. In order to be as abstract as possible, we simply model intention selection as a function from the set of intentions to a single intention, and leave the actual selection process undefined.

PickIntention

$\Delta Agent$
$i! : Intention$

$intentions \neq \varnothing$
$beliefs' = beliefs$
$intentions' = intentions \setminus \{i!\}$
$events' = events$
$plans' = plans$
$i! \in intentions$

Once an intention has been chosen, the agent picks its first element (i.e. the first plan item). Since each intention is a sequence, this process is straightforward. The only requirement for the intention is to be non-empty.

PickFirstElement

$i? : Intention$
$i! : Intention$
$p! : PlanItem$

$i? \neq \varnothing$
$i? = \langle p! \rangle \frown i!$

If the intention chosen for execution is empty, then nothing happens. This is described by the following operation schema (empty intentions are removed in operation *CleanUp* defined further below).

IntentionEmpty

$i! : Intention$

$i! = \varnothing$

The most complex step during the execution of an intention is the execution of a plan item, of which there are four different types. In the case of an *atomic action*, the agent simply executes it. Since the internal structure of the action is not modelled explicitly, action execution is also modelled on an abstract level by means of function *executeAction* in schema *Agent*. Actions can be seen as operations upon the agent's knowledge; they can thus be modelled as a function from actions and belief bases to belief bases. The second case deals with *sequential goal addition*. Here, the plan item is simply replaced with the plan that has been chosen to handle the respective goal. Technically, this achieves the effect of *subgoal expansion*. The third case describes *parallel goal addition*. Here, rather than performing subgoal expansion, a new parallel intention needs to be created. This is achieved by creating a *goal addition event* and storing it in the agent's event store. As described further above, the goal addition event will then create a new intention. And, finally, the last case deals with *goal removal*. As with parallel goal addition, a *goal removal event* is created and placed in the agent's event store. We start by defining the execution of an atomic action.

ExecuteAction

$\Delta Agent$
$i? : Intention$
$p? : PlanItem$

$(\exists a : Ac \bullet$
$\quad p? = Action(a) \land$
$\quad beliefs' = executeAction(a, beliefs))$
$events' = events$
$intentions' = intentions$
$plans' = plans$

Executing a sequential goal addition is slightly more complex. It involves two steps: (i) finding a plan for the goal, and (ii) subgoal expansion, i.e. replacing the goal addition with the plan. For clarity, we start with the identification of the event.

IsSeqAddition

$\Xi Agent$
$i? : Intention$
$p? : PlanItem$
$g! : Goal$

$(\exists g : Goal \bullet p? = SeqAddition(g!))$

Finding a plan has already been described further above; we can thus reuse our previous operation schemes. However, we need to define the process of subgoal expansion. In order to do that, we distinguish between the cases where the intention being passed is empty and those where it is non-empty. This is described formally by the following two operation schemas and their conjunction.

*ExpandSubgoal*0

$\Delta Agent$
$i? : Intention$
$p? : Plan$

$i? \neq \varnothing$
$beliefs' = beliefs$
$events' = events$
$intentions' = (intentions \setminus \{i?\}) \cup \{p? \frown (tail\, i?)\}$
$plans' = plans$

*ExpandSubgoal*1

$\Delta Agent$
$i? : Intention$
$p? : Plan$

$i? = \varnothing$
$beliefs' = beliefs$
$events' = events$
$intentions' = (intentions \setminus \{i?\}) \cup \{p?\}$
$plans' = plans$

$$ExpandSubgoal \mathrel{\widehat=} ExpandSubgoal0 \lor ExpandSubgoal1$$

We can now define the overall operation of executing a sequential goal addition using schema calculus. Similar to above, we start in reverse order and define the sequence of identifying relevant and applicable plans, and selecting one plan from the resulting set.

$$FindPlan0 \mathrel{\widehat=} FindRelevantPlans \gg FindApplicablePlans \gg SelectPlan$$

The resulting operation schema can now be prepended with *IsSeqAddition* in order to describe the process of (i) identifying sequential goal addition, (ii) extracting the goal to be handled, and (iii) finding an appropriate plan.

$$ExecuteSeqAddition0 \mathrel{\widehat=} IsSeqAddition \gg FindPlan0$$

The resulting operation schema has one input argument, $p? : PlanItem$, and two output arguments $i! : Intention$ and $p! : Plan$. We can thus append subgoal expansion as follows:

$$ExecuteSeqAddition1 \mathrel{\widehat=} ExecuteSeqAddition0 \gg ExpandSubgoal$$

The resulting operation is complete with respect to its functionality, yet not totalised. In order to provide a totalised version, we need to consider the different error cases defined further above.

$$ExecuteSeqAddition \mathrel{\widehat=} ExecuteSeqAddition1 \lor NoRelevantPlans \lor NoApplicablePlans$$

This concludes the formalisation of sequential goal addition. We can now describe the next type of plan item, parallel goal addition. This case is fairly straightforward. All that needs to be done is to create a goal addition event and add it to the event store. This is described by the following operation schema.

ExecuteParAddition

$\Delta Agent$
$i? : Intention$
$p? : PlanItem$

$(\exists g : Goal \bullet$
$\quad p? = ParAddition(g) \land$
$\quad events' = events \cup \{Add(g)\})$
$beliefs' = beliefs$
$intentions' = intentions$
$plans' = plans$

The final step is to describe the execution of goal removal. Here, we simply need to create a dedicated goal removal event and add it to the event store.

ExecuteRemoval

$$
\begin{array}{l}
\Delta Agent \\
i? : Intention \\
p? : PlanItem \\
\hline
(\exists g : Goal \bullet \\
\quad p? = Removal(g) \land \\
\quad events' = events \cup \{Remove(g)\}) \\
beliefs' = beliefs \\
intentions' = intentions \\
plans' = plans
\end{array}
$$

The overall operation of executing a plan item can now be described succinctly by disjoining the previously defined suboperations using schema calculus.

$$ExecutePlanItem \mathrel{\widehat=} ExecuteAction \lor ExecuteSeqAddition \lor ExecuteParAddition \lor ExecuteRemoval$$

We can now combine the previous operations into the overall operation of executing an intention i. For a single update step of i, the agent first picks the first plan item p and executes it. This process can only fail if the chosen intention is empty.

$$StepIntention \mathrel{\widehat=} (PickFirstElement \gg ExecutePlanItem) \lor IntentionEmpty$$

The execution of an intention is then defined as a sequence of two steps: (i) selecting an intention for execution, and (ii) executing one step of the intention. This process can only fail if there are no intentions.

$$ExecuteIntention \mathrel{\widehat=} (PickIntention \gg StepIntention) \lor NoIntentions$$

Lastly, for the sake of completeness, we define a cleanup operation that removes from the set of intention all those intentions that are empty (i.e. those that have already been completed).

CleanUp

$$
\begin{array}{l}
\Delta Agent \\
\hline
beliefs' = beliefs \\
events' = events \\
intentions' = intentions \setminus \{\langle\rangle\} \\
plans' = plans
\end{array}
$$

This concludes the description of individual operations. In the next section, those operations are combined to form the overall agent update cycle.

4.2.5 The overall update cycle

The overall update step of an individual agent can now be defined as a simple sequence of four distinct, previously formalised steps: (i) perceiving the environment, (ii) processing an event,

(iii) executing an intention, and (iv) cleaning up the set of intentions. The selection of a plan happens as part of the event processing step.

$$Step \mathrel{\widehat{=}} Perceive \fatsemi ProcessEvents \fatsemi ExecuteIntention \fatsemi CleanUp$$

This sequence — which concludes the formalisation — corresponds with the informal description of the individual agents' update cycle given in Sections 2.2 and 2.3.

4.3 Customisation

The formal framework given in the previous sections contains a number of aspects that have been deliberately described on an abstract, not further specified level. Those aspects can be seen as dedicated *customisation points*, i.e. as aspects that the user can adapt to his circumstances. A summary is given below.

- The type of beliefs
- The internal dynamics of the environment
- The agents' perception of the environment
- The internal dynamics of agent actions
- Selection mechanisms:
  - Event selection
  - Plan selection
  - Intention selection

Customisation points have important implications on the software design, since the desired variability with respect to certain components has to be accounted for explicitly. In alignment with the requirements formulated in Section 3, the C++ framework described in Chapter 5 allows for the customisation of each of the points in the list above.

5 A C++ TMP-based BDI framework

The two major goals of the C++ framework developed during this project are *efficiency* and *usability*. The former is achieved by providing efficient algorithms and by shifting some of the computation from runtime to compile time. This involves both the *execution* of entire steps (e.g. the plan relevance check) as well as the *preparation* of efficient runtime structures using appropriate transformations at compile time. Both steps require a fair amount of template metaprogramming (TMP). The second goal is achieved by extending the C++ implementation with an embedded domain-specific language.

The C++ framework is implemented using a mix of purely functional and object-oriented code. As described in Section 2.4 above, C++ TMP represents a purely functional programming language; those computations in the framework that are to be performed at compile time are thus formulated in a purely functional way. On the other hand, the computations that are to be executed at runtime (e.g. event processing, intention execution) are inherently stateful in nature since they operate upon the agent's belief base, its current set of events, and its intentions; even though stateful computations can be perfectly well realised in a purely functional setting, we implemented them in an imperative, object-oriented way for reasons of efficiency.

One of the disadvantages of C++ TMP is its awkward syntax, due to which even simple computations become unreadable very quickly. For that reason, we looked for an appropriate way of describing the functionality of the framework in a manner that is abstract and succinct enough to be immediately comprehensible, yet still close enough to the ultimate implementation. With its schema calculus, the Z notation (which we used for the formal description of the framework in Section 4) provides a great toolbox for the description of stateful computations in a purely mathematical way. However, if recursive algorithms need to be described (as often the case in the purely functional world), schema calculus is no longer useful; one solution is to revert to axiomatic functions which, however, defeats the very purpose of the Z notation.

Haskell with its succinctness and its support for both purely and 'imperative-style' functional programs provides a powerful tool for the specification of both functional and stateful, imperative algorithms. Due to the functional nature of C++ TMP, compile time calculations can be described very elegantly using Haskell. In fact, due to the conceptual closeness between the two formalisms, Haskell algorithms translate into C++ TMP algorithms almost seamlessly. For that reason, we decided to use Haskell-style syntax for the specification of the algorithms and data structures in the sections to follow. It is important to note that, although typechecked and thus executable in general, the Haskell code fragments serve purely descriptive purposes and are, in their current state, not intended as executable implementations in their own right. The main reason for that is that the Haskell implementation omits some of the specific details of the BDI framework such as plan selection or applicability checking mechanisms. A description of necessary extensions is, however, beyond the scope of this work.

In analogy to the formalisation given in Section 4, we start with the static components of the framework in Section 5.1, followed by the operations in Section 5.2. In order not to clutter up the description of the general framework with too much implementation-specific de-

tails, we chose to move the description of concrete plan selection mechanisms into a separate Section (5.3). As indicated in Section 4.3 above, the framework contains a number of customisation points; their realisation in the C++ framework is described in Section 5.4. Finally, the framework components were designed such that they form an *embedded domain-specific language (EDSL)* that resembles the declarative interfaces used by AOP languages such as Jason or 2APL; this is described in further detail in Section 5.5. The chapter concludes with a summary in Section 5.6.

5.1 Components

For clarity, we follow the same structure as in Section 4 and start the description with the environment, followed by the belief base, goals, actions, events, plans, intentions, and, finally, the overall agent state.

5.1.1 Environment

In the context of multiagent systems, the environment is used to describe the world that the agents are situated in, which they are able to perceive and to act upon. Due to the infinite number of possible realisations, the internal workings of the environment are typically not under the explicit control of the agent-based framework and thus represented differently from the actual agent logic. In Jason, for example, in order to allow for the highest level of flexibility, environments are represented as conventional Java objects.

We follow this approach and allow for the formulation of the environment as a conventional C++ object. However, rather than expecting the environment to inherit from a base class or to implement a particular interface, we use static polymorphism: we only expect the environment object to provide a function `perceive` which accepts the agent state as a parameter, as shown below. What happens inside the function is beyond the control of the framework.

```
struct Environment {
  template <class AS>
  void perceive(AS& as) { ... }
};
```

5.1.2 Belief base

Conceptually, the belief base represents everything that the agent knows about itself as well as about the environment that it inhabits. For simplicity and efficiency, it is common in agent-based modelling to represent knowledge in terms of *attributes*, i.e. simple key-value pairs. Clearly, attributes may come in all flavours and thus need to be represented using a variety of different data types. For example, information about the agents' age may be represented as integer values, information about the agents' attitudes towards a given product may be represented as floating point values, and behavioural traits may be represented as simple Boolean flags. In order to keep the development efficient, associative containers such as dictionaries are frequently used.

For the abstract specification, we assume that belief names are strings, and belief values are integers (this simplifying assumption is dropped in the implementation described below).

type $BeliefName = String$
type $BeliefValue = Int$

An agent's belief base is then represented as a simple key–value store.

type $BeliefBase = BeliefName \rightarrow BeliefValue$

However, in general, any agent-based modelling framework should be fairly flexible with respect to the data types of agent attributes that it supports. Since we cannot anticipate the types that a modeller may want to use in his model, we have to provide a solution that is sufficiently flexible with respect to the handling of types, yet still performant enough. C++ is a statically typed language, so we cannot hope to represent values of different types within a single container (such as, for example, in Python). With the *Any* library, Boost offers a possible solution to this problem. Boost.Any represents a discriminated type that allows for the assignment of any value (`float`, `int`, `bool`, etc.). Although the type of each value is preserved, Boost.Any offers a uniform runtime interface. Despite the convenience, this approach comes at a comparatively high runtime cost since explicit conversions are necessary whenever the actual value is to be accessed. As a consequence, we chose not to use Boost.Any for the implementation of the belief base but to follow a different approach described further below.

In order to achieve the desired level of flexibility in C++, we let the user decide which types he would like to support in the knowledge base. From a developer's point of view, the belief base should be usable as follows:

```
// Declaration of belief base
BeliefBase<float,string,int> bb;

// Example beliefs
float fBelief = 42.0;
string sBelief = "foo";
int iBelief = 4711;

// Belief addition
bb.addBelief("float belief", fBelief);
bb.addBelief("string belief", sBelief);
bb.addBelief("int belief", iBelief);

// Belief query
bb.getBelief("float belief", &fBelief);
bb.getBelief("string belief", &sBelief);
bb.getBelief("int belief", &iBelief);
```

Here, `float`, `string`, and `int` are the types of the beliefs to be stored. The belief base should thus represent a union of different associative containers — one for each type. In terms of adding and querying beliefs, rather than providing different getter and setter functions depending on the type to be accessed, the belief base should provide a uniform interface. In the example above, this is shown under "Belief addition" and "Belief query", respectively, where one single function is used in either case.

In order to achieve the heterogeneity of the belief base, we make a conceptual subdivision between a *belief set* and a *belief base* and use a combination of inheritance and variadic templates in order to connect the two concepts and provide the appropriate level of flexibility. As the name suggests, a belief set represents a set of beliefs *of one particular type*, e.g. integer. A belief set can thus be seen as a simple wrapper around a dictionary that stores attribute keys (of type `string`) and their values (of whatever type is required by the user). The implementation

of the generic class `BeliefSet` is shown below.

```
template <class T>
class BeliefSet {
public:
  BeliefSet() {};

  void addBelief(std::string const& name, T const& val) {
    m_Beliefs[name] = val;
  }

  bool getBelief(std::string const& name, T& out) const {
    auto it = m_Beliefs.find(name)
    if(it != m_Beliefs.end()) {
      out = it->second;
      return true;
    }
    return false;
  }

  bool getBelief2(std::string const& name, T& out) {
    out = m_Beliefs[name];
    return true;
  }

  bool getBeliefP(std::string const& name, T*& out) {
    auto it = m_Beliefs.find(name);
    if(it != m_Beliefs.end()) {
      out = &it->second;
      return true;
    }
    return false;
  }

protected:
  std::unordered_map<std::string,T> m_Beliefs;
};
```

The class is parametrised with the type of belief (`int`, `double`, etc.) that this particular set is responsible for. It contains an unordered map (unordered for performance reasons) which serves as the associative container for the beliefs. It further provides several accessor functions that allow the developer to add and query beliefs. For performance reasons, belief query is possible in three different ways: (i) using a constant function `getBelief` which performs an existence check for the belief being queried (default mechanism), (ii) using a non-constant function `getBelief2` with better performance but without existence check, and, (iii) for situations where even higher performance is required, using a non-constant function `getBeliefP` which returns a pointer to the respective belief.

The belief base itself is more interesting. Its purpose is to allow a developer to specify an arbitrary set of types and combine the belief sets for each of those types under a single, convenient interface — including the accessor functions mentioned above. This can be achieved by

creating an appropriate *inheritance structure* where the belief base inherits from all individual belief sets. We start with the basic interface of the belief base shown below.

```
template <class... T>
struct BeliefBase;
```

The class is parametrised with a *parameter pack*, i.e. an arbitrary list of types, each of which represents a type of 'knowledge item' to be supported by the knowledge base. In order to be able to let `BeliefBase` inherit from each type in the parameter pack, we follow a recursive approach. Remember, as described in Section 2.4, the template system of C++ represents a purely functional language. Effectively, a template parameter pack represents a list and we can thus follow an approach similar to a recursive algorithm in any other functional programming language such as Haskell: the problem is described by means of a *base case* and one or more *recursive cases*.

The base case is shown below. Here, we consider a typelist that only contains a single element – the type `T1`. The logic in this case is trivial, we simply derive from the corresponding belief set class `BeliefSet<T1>`. In order not to mask the individual accessor functions of each belief set, they need to be made available explicitly by an appropriate `using` clause. This achieves the effect of providing the user with unique accessor functions as illustrated in the usage example in the beginning of this section.

```
template <class T1>
struct BeliefBase<T1> : public BeliefSet<T1> {
  using BeliefSet<T1>::addBelief;
  using BeliefSet<T1>::getBelief;

  BeliefBase()
  : BeliefSet<T1>() {}
};
```

For the following recursive case, we split up the list of types into a head (type `T1`) and a tail (parameter pack `Ts`). As in the base case, we derive from the belief set class for `T1`, but *also from the belief base parametrised with the tail* `BeliefBase<Ts...>`. This achieves the desired recursion effect which results in the construction of a class that contains all the individual belief sets of the types specified.

```
template <class T1, class... Ts>
struct BeliefBase<T1, Ts...>
: public BeliefSet<T1>, public BeliefBase<Ts...> {
  using BeliefSet<T1>::addBelief;
  using BeliefSet<T1>::getBelief;
  using BeliefBase<Ts...>::addBelief;
  using BeliefBase<Ts...>::getBelief;

  BeliefBase()
  : BeliefSet<T1>(), BeliefBase<Ts...>() {}
};
```

5.1.3 Goals

As motivated in Section 3.3 above, an important requirement for simulations is the possibility to formulate parametrised plans. Intuitively, a parametrised plan represents a template that can be instantiated for a given *option*, e.g. a particular product (as in the example given above). Options have a close correspondence with attributes. For the sake of simplicity, we assume here that both the option name and the option value are represented as strings. As described below, in the C++ representation, the option type can be defined explicitly by the user. An agent's options library can then be represented as a mapping from option names to option values.

$$\begin{aligned}&\textbf{type } \mathit{OptionName} = \mathit{String}\\&\textbf{type } \mathit{Option} \quad\;\; = \mathit{String}\\&\textbf{type } \mathit{Options} \quad\;\; = \mathit{OptionName} \rightarrow [\mathit{Option}]\end{aligned}$$

In the C++ framework, options are not represented as types but as parameters to goals, actions, and plans. This is described in more detail further below.

A goal is uniquely characterised by its name which we assume is represented as a simple string. A goal itself is either a *basic goal* or a *parametrised goal* (in order to serve as the basis for parametrised plans). In both cases, the goal comprises a name; in the parametrised case, the goal also comprises an *option name* as described above.

$$\begin{aligned}&\textbf{type } \mathit{BGoalName} = \mathit{String}\\&\textbf{type } \mathit{PGoalName} = \mathit{String}\\&\textbf{data } \mathit{Goal} = \mathit{BGoal}\,\{\,\mathit{goal\_gn} :: \mathit{BGoalName}\,\} \mid\\&\qquad\qquad \mathit{PGoal}\,\{\,\mathit{goal\_gn} :: \mathit{PGoalName},\\&\qquad\qquad\qquad\quad \mathit{goal\_on} :: \mathit{OptionName}\,\}\ \textbf{deriving}\ (\mathit{Eq})\end{aligned}$$

Goals can be translated into their C++ equivalents in a straightforward way. Like in Haskell, we represent them as parametrised types. A basic goal accepts one parameter: the user-defined goal that it is supposed to represent.

```
template <class Goal>
struct BGoal {};
```

A parameterised goals accept three parameters: (i) the user-defined goal type that it is supposed to represent, (ii) the option name (as described above), and (iii) the option type.

```
template <
  class Goal,
  const char* Param,
  class Type
>
struct PGoal {
   const char* param = Param;
};
```

5.1.4 Actions

An action is either a *basic action* or a *parametrised action*. Both types comprise a name which is a simple string; in the latter case, the action also comprises an option name.

$$\begin{aligned}
&\textbf{type } BActionName = String\\
&\textbf{type } PActionname\ = String\\
&\textbf{data } Action = BAction\ \{\, action_name :: BActionName\,\} \mid\\
&\qquad\qquad PAction\ \{\, action_name :: PActionName,\\
&\qquad\qquad\qquad\qquad option_name :: String\,\}\ \textbf{deriving}\ (Eq)
\end{aligned}$$

In C++, similar to goals, actions are represented as types. However, in addition to their static representation, actions also have an important role at runtime. In the C++ framework, actions have a slightly wider meaning than they have in the formal description above. Everything that is executable — be it a basic action, a plan, a parallel goal addition, etc. — is viewed as an action. In order to manipulate those different action instances at runtime, we extend the basic action type with two member functions, `done` and `advance`. The former denotes whether the executable entity has already been finished. In the case of basic actions, this is clearly irrelevant; the function thus always returns *true*. The latter function advances the executable entity by one step. This is only relevant for non-basic executable entities, i.e. plans; the function is thus left empty.

```
struct BAction {
    bool done() { return true; }
    void advance() {}
};
```

Parametrised actions further comprise a member variable holding the parameter value, as well as a corresponding setter function.

```
template <class ParamType>
struct PAction {
    PAction() {}

    bool done() { return true; }
    void advance() {}

    void setParam(ParamType const& _param) { param = _param; }
    ParamType param;
};
```

5.1.5 Events

Following the description in Section 4.1.5, two types of events are supported: goal addition and goal removal. Similar to the previous concepts, events can thus be represented as an algebraic data type.

$$\textbf{data } Event = Add\ Goal \mid Remove\ Goal\ \textbf{deriving}\ (Eq)$$

Again, the translation into C++ is straightforward. We represent events as parametrised types that inherit from a common base class `Event`. By analogy with the Haskell type class *Eq*, base class `Event` provides as its only element the comparison operator which is, for example, required when storing and accessing instances of types in containers.

```
template <class Goal>
```

```
struct Add : Event {};

template <class Goal>
struct Remove : Event {};
```

An agent's event store can then be defined as a simple list of events.

type $Events = [Event]$

Events are to be manipulated at runtime, so we need an appropriate container for their storage. In order to allow for the greatest level of flexibility, we decide for a std::vector. However, what type should the actual events have? C++ is a statically typed language, so we cannot hope to have containers holding elements of different type. An obvious solution would be to use Boost.Any. However, as described above, Boost.Any is designed for runtime access and requires frequent type conversion which we consider too expensive here.

The solution that we choose is to define an appropriate *variant type*. The set of all possible event types can be seen as the Cartesian product of the set of all possible user-defined goal types and the set of all possible *kinds of events* (i.e. goal addition and goal removal). If we assume that the set of possible goal types as well as the set of event *kinds* are known at compile time, then the set of possible *event types* (the Cartesian product) can also be computed at compile time. Intuitively, for each goal, there are two concrete event types: addition and removal. The resulting set of event types is thus twice as large as the original set of goal types.

The compile time calculation can be realised as a sequence of steps, wrapped within a single metafunction class. Given a set *goals* of user-defined goal types, the first step comprises the computation of the set of possible *basic goals*. Intuitively, given a list of user-defined goals *[GetMoney, GetPresent]*, we want to apply them to the type constructors *Add* and *Remove* in order to produce the list *[BGoal GetMoney, BGoal GetPresent]*. This can be achieved by means of a simple map operation:

let $bgoals = map\ (\lambda\ g \rightarrow BGoal\ g)\ goals$

The same can be expressed in C++ TMP using `mpl::transform`, the MPL equivalent of *map*.

```
typedef typename boost::mpl::transform<
  Goals,
  BGoal<boost::mpl::_1>
>::type BGoals;
```

In the next step, the set of all possible combinations of basic goals and events needs to be computed. Using set *bgoals* defined above, this can be done as follows.

let $additionEvents = map\ (\lambda\ g \rightarrow Add\ g)\ bgoals$

In C++, this can be done as in the previous step.

```
typedef typename boost::mpl::transform<
  BGoals,
  Add<boost::mpl::_1>
>::type AdditionEvents;
```

In the next step, we define the set of removal events accordingly.

let $removalEvents = map\ (\lambda\ g \rightarrow Rem\ g)\ bgoals$

The C++ implementation is equivalent to the previous one.

```
typedef typename boost::mpl::transform<
  BGoals,
  Add<boost::mpl::_1>
>::type RemoveEvents;
```

The final list of possible events can now be produced by concatenating the two previously computed lists.

let $events = additionEvents^{+} removalEvents$

In C++, this can be realised by using `mpl::copy`, a compile time function that accepts as its first parameter a typelist and as its second parameter an insertion algorithm. The result is again a typelist.

```
typedef typename boost::mpl::copy<
  RemovalEvents,
  boost::mpl::front_inserter<AdditionEvents>
>::type type;
```

For better usability, all previous compile time operations are wrapped into a static metafunction class `expandEventTypes` which is partially shown below.

```
struct expandEventTypes {
  template <class Goals>
  struct apply {
    BOOST_MPL_ASSERT(( boost::mpl::is_sequence<Goals> ));

    ... // individual transformation steps go here

    typedef typename boost::mpl::copy<
      RemovalEvents,
      boost::mpl::front_inserter<AdditionEvents>
    >::type type;
  };
};
```

Compile time function `expandEventTypes` can now be 'called' by instantiating its inner template with the typelist of goals and accessing the resulting type definition.

```
typedef typename boost::mpl::apply<expandEventTypes,Goals>::type expanded_events1;
```

The result of the compile time call is a list of all possible events. There is no runtime equivalent of this type. In order to mimic a heterogeneous vector that can hold elements of all types represented by `expanded_events`, we thus need to turn `expanded_events` into a Boost.Variant. This can be done by calling the static metafunction `boost:make_variant_over` which accepts a typelist parameter `T` and returns a Boost.Variant over all types in `T`.

```
typedef typename boost::make_variant_over<expanded_events1>::type event_type
```

The resulting type can then be used to define the desired runtime event store.

```
std::vector<event_type> events;
```

5.1.6 Plans

A *plan item* is either an action, a (sequential) goal addition, or a parallel goal addition.

data $PlanItem = Ac\ Action \mid Seq\ Goal \mid Par\ Goal$

In C++, the plan items are represented as simple types, parametrised with the user-defined goal that they are supposed to handle. Actions have already been described above; they can be reused here. It thus remains to describe sequential and parallel goal addition which are also represented as simple parametrised types.

```
template <class Goal>
struct Par {};

template <class Goal>
struct Seq {};
```

A *plan body* can now be defined as a simple list of plan items.

type $PlanBody = [PlanItem]$

A *plan* is then either a *basic plan* which consists of a name and a plan body, or a *parametrised plan* which consists of a name, an option name, and a plan body.

$$\begin{aligned}\textbf{data}\ Plan = \ & BPlan\ \{plan\_gn :: BGoalName,\\ & \quad plan\_pb :: PlanBody\}\ \mid\\ & PPlan\ \{plan\_gn :: PGoalName,\\ & \quad plan\_on :: OptionName,\\ & \quad plan\_pb :: PlanBody\}\\ \textbf{type}\ Plans = & [Plan]\end{aligned}$$

In C++, a plan is represented as a type parametrised with (i) its name, (ii) the goal it promises to handle, and (iii) its body. Note that the option parameter is not represented explicitly. This is due to the fact that, in the C++ implementation, the option parameter is held as part of the global state being operated upon. As a consequence, there is only a single plan type.

```
template <
  int Name,
  class Goal,
  class Body
>
struct Plan {
  typedef Body body;
  typedef Goal goal;
  const int name = Name;
};
```

5.1.7 Intentions

As formally described in Section 4.2.4, we denote with an intention the executable, i.e. *expanded* version of a plan. An intention is a recursive data structure representing (i) a basic action, (ii) a parallel goal addition, or (iii) a sequence of the aforementioned items.

$$\begin{aligned}\textbf{data}\ \mathit{Intention} = {} & \mathit{BasicAc}\ \mathit{Action}\ | \\ & \mathit{Rem}\ \mathit{Goal}\ | \\ & \mathit{ParAdd}\ \mathit{Goal}\ | \\ & \mathit{Sequence}\ [\mathit{Intention}]\ \textbf{deriving}\ (\mathit{Eq})\end{aligned}$$

In analogy to previous sections, in C++, intention items are also represented as parametrised types. Actions have already been described above; they can be reused here. It thus remains to describe *goal removal*, *parallel goal addition*, and *sequence*.

Goal removal and parallel addition are represented as parametrised types. Intentions represent executable entities and are thus intuitively specialisations of basic actions (realised by means of inheritance). Furthermore, in order to make instances of intentions executable, they are realised as *functors*[1] or *function objects* (i.e. they provide an implementation of the function call operator). The content of the function call operator is described in further detail in the context of intention execution in Section 5.2.4 below.

```
template <class Goal>
struct Rem : public BAction {
  template <class Env, class AS>
  bool operator()(Env& env, AS& as) const { ... }
};

template <class Goal>
struct Par : BAction {
  template <class Env, class AS>
  bool operator()(Env& env, AS& as) const { ... }
};
```

It remains to discuss the recursive case, i.e. sequences of intentions. Informally, a sequence simply acts as a wrapper to a set of sub-intentions which, at runtime, get executed in the order in which they were stored. A sequence is one of a number of *control flow nodes*. Conceptually, a control flow node can be seen as a function from one set of executable entities to another. Intuitively, given a set $\mathscr{E}$ of entities available for execution, a control flow node defines how the members of $\mathscr{E}$ are to be executed. A sequence executes all the members of $\mathscr{E}$ in order[2]. In C++, sequences are implemented as a hierarchy of classes shown in Listings C.2–C.4.

The agent's set of intentions is then defined as a list of intention items:

$$\textbf{type}\ \mathit{Intentions} = [\mathit{Intention}]$$

In C++, in order to provide a high level of flexibility with respect to access, we represent the set of intentions as a `std::vector`. However, as in the case of events describe above, we face

[1] In C++, the term 'functor' is used differently from that in category theory and functional programming. In the last two cases, a functor is typically understood as a type that can be mapped over, whereas in C++, a functor refers to a function object. When we talk about functors here, we always mean C++ functors.

[2] *Selectors* as the second type of control flow node will become particularly important in the context of *meta reasoning*; they are thus described in further detail in Section 5.3.

the problem that containers in C++ can only be homogeneous, whereas different intentions are represented as different types. The set of intentions is defined as the set of all possible expanded plans. In order to store instances of all those different types of intentions in a common container, they need to be unified. We can thus follow an approach similar to that used in the case of events.

In order to define an appropriate Boost.Variant type, we perform again a number of steps at compile time. First, the set of possible goal types is translated into the set of possible event types using metafunction `expandEventTypes`, as described in Section 5.1.5 above.

```
typedef typename boost::mpl::apply<expandEventTypes,Goals>::type expanded_events1;
```

In the next step, `expanded_events1` is used as the basis for the definition of all possible courses of action that may arise throughout the simulation. For each possible event (i.e. each possible goal–addition/removal combination), all applicable plans are determined and expanded recursively, as described in more detail in Section 5.3 below. This transformation process is triggered as follows and performed by metafunction `expandEvent` which is described in Section 5.2.3 below.

```
typedef typename boost::mpl::transform<
  expanded_events1,
  expandEvent<PL,Selector>
>::type expanded_events;
```

The resulting typelist `expanded_events` is then wrapped inside a Boost.Variant in order to be storable in a homogeneous container.

```
typedef typename boost::make_variant_over<expanded_events>::type intention_type;
```

5.1.8 Agent state

Using the components defined above, an *agent state* can then be defined as a simple union of beliefs, plans, intentions, events, and options.

$$\begin{aligned}\textbf{data}\ AgentState = AS\ \{\ & bb :: BeliefBase,\\ & pl :: Plans,\\ & is :: Intentions,\\ & eq :: Events,\\ & os :: Options\}\end{aligned}$$

Technically, the agent state is the most central component in the C++ framework since it represents the point where compile time and runtime computations meet. Some of the compile time transformation were described above, further descriptions follow in Section 5.2 below. The basic structure of the `AgentState` class is shown below.

```
template <                                                    1
  class BB,                                                   2
  class PL,                                                   3
  class Goals,                                                4
  class InitEvents,                                           5
```

```
  class Applicability=RTApplicability,                                    6
  class Selector=RandomSelector                                           7
>                                                                         8
struct AgentState {                                                       9
  BOOST_MPL_ASSERT(( boost::mpl::is_sequence<InitEvents> ));              10
  BOOST_MPL_ASSERT(( boost::mpl::is_sequence<PL> ));                      11
  BOOST_MPL_ASSERT(( boost::mpl::is_sequence<Goals> ));                   12
                                                                          13
  typedef PL pl;                                                          14
                                                                          15
  /* Compile-time transformations go here */                              16
                                                                          17
  AgentState() { initEvents(); }                                          18
  AgentState(BB const& _bb) : bb(_bb) { initEvents(); }                   19
                                                                          20
  BB& getBB() { return bb; }                                              21
                                                                          22
  void addEvent(event_type const& ev) { events.push_back(ev); }           23
  void remEvent(event_type const& ev) {                                   24
    events.erase(                                                         25
      std::remove(events.begin(), events.end(), ev),                      26
      events.end()                                                        27
    );                                                                    28
  }                                                                       29
  void addIntention(intention_type const& in) { intentions.push_back(in); } 30
                                                                          31
  void initEvents() {                                                     32
    boost::mpl::for_each<InitEvents>([this](auto const& ev){              33
      events.push_back(ev);                                               34
    });                                                                   35
  }                                                                       36
                                                                          37
  std::vector<event_type> events;                                         38
  std::vector<intention_type> intentions;                                 39
  BB bb;                                                                  40
};                                                                        41
```

The state is parametrised with six types: a *belief base*, a *plan library*, a set of *goals*, a set of *initial events*, the *applicability checking mechanism*, and the *plan selection mechanism*. The first four concepts are now briefly described in turn, the applicability checking mechanism and the plan selection mechanism are described in Sections 5.2.2 and 5.3.

Belief base: As described in Section 5.1.2, the purpose of the belief base is to provide access to the agent's knowledge whenever necessary. For that reason, the user can either pass an existing instance of the belief base to the state class upon construction (as shown in Line 19), or leave the construction of a default instance to the state class. If the belief base has to be accessed, function `getBB` which returns a non-const reference can be used.

Plan library: The plan library represents the agent's *capabilities*, i.e. the set of plans that it can choose from. As described in Sections 4.1.6 and 5.1.6 above, each plan is represented as a

separate type which results in the plan library representing a heterogeneous collection. In order to handle this heterogeneity efficiently, the plan library is required to be an MPL sequence (as denoted by the MPL assertion), the actual type of which (list, vector) is not further prescribed. MPL sequences allow for efficient manipulation of type lists at compile time. As shown in Line 14, the type of the plan library is stored for further reference.

Goal set: In order for the state class to be able to react to events, e.g. to find plans for goals, it needs to have access to the available list of goal types. Remember that goals are represented as types. We thus need to pass to the state class a list of goal types. This is the purpose of the third template parameter. As with the plan library, the goal set is expected to be an MPL sequence.

Initial events: The starting point for the reasoning process of an individual agent is the processing of initial events (if there are any). As described in Sections 4.1.5 and 5.1.5 above, there are two types of events in the framework — goal addition and goal removal — each of which is represented by its own generic type. An agent can have an arbitrary number of initial events so, as with the previous two parameters, the set of initial events is expected to be an MPL sequence. In the constructor of the state class, each initial goal type is instantiated and added to the STL vector `events` in order to be processed at runtime. As described above, runtime data structures such as `std::vector` cannot contain heterogeneous types; the original types thus need to be wrapped into Boost.Variants, as described in Sections 5.1.5 above. In order to add events to and remove events from the vector, the agent state class further provides two functions `addEvent` and `removeEvent`.

Apart from events, *intentions* also play an important role in the reactive planning process. The management of intentions is inherently dependent upon the circumstances that an agent finds itself in and thus cannot be performed at compile time. In order to allow for the processing of events at runtime, the state class contains a vector `intentions`; the definition and management of intentions is described in further detail in Section 5.2.4.

In order to allow an agent to influence both its own state as well as the state of the environment during its update step, we combine both states into a single data structure, the *system state*, which is defined as a simple tuple.

$$\textbf{data}\ SystemState = SS\ \{\, as :: AgentState,\\ es :: Environment \,\}$$

5.2 Operations

The previous sections dealt with the static aspects of the C++ framework, i.e. with types representing central components as well with basic transformations between types. This section describes the implementation of the BDI operations in C++. We follow the same order as in the formal description in Section 4 above and start with perception, followed by plan selection, event processing, and intention execution. We finish by describing the overall update cycle.

5.2.1 Perception

Conceptually, perception represents a mapping from the environment to a new agent state. Since both the agent state and the state of the environment is wrapped into the system state, as described above, the mapping happens implicitly. The actual nature of perception depends on the content of the environment and is thus deliberately left unspecified here.

$$perceive :: State\ SystemState\ ()$$

There are countless ways in which the perception function can be realised in C++: as a parametrised function, as a member function of the agent class accepting a reference to the environment, or, conversely, as a member function of the environment class accepting a reference to the agent class. We decided to follow the last idea. As part of the perception process, the environment typically has to make a number of decisions, e.g. as to which environmental information is to be perceivable by which agent. In general, this involves a fair amount of agent-independent logic. We thus believe that it is, in most cases, more practical for the perceive function to part of the environment object.

5.2.2 Plan selection & expansion

As described in various places above, plan selection involves a number of steps — finding the set of relevant plans, determining the set of applicable plans, and selecting a plan for execution. In the Z notation, this was represented by means of *schema piping*, i.e. by concatenating functions and passing values between them. For performance reasons, we try to avoid passing around values in the C++ implementation whenever possible and aim to operate on common data structures instead. Nevertheless, modularity should be preserved. For all steps except the first one (relevance checking, which is a pure compile time computation), this is achieved by employing the idea of *policy-based class design*, a combination of generic classes and inheritance as described in Section 2.4.2. By decomposing the overall plan selection functionality into different classes which are then assembled into a (linear) inheritance hierarchy, operation upon common data structures is made possible whilst still preserving modularity and exchangeability. We describe each of the substeps in the following paragraphs, again following the Haskell–C++ scheme.

Relevance check

The first step involves finding relevant plans for a given goal. Formally, this can be represented as a function from a given goal to a set of plans. However, reading access to the agent's state (in particular, its plan library) is required. Rather than passing the agent state as a simple parameter to the function, we use the Reader monad to emphasise the side-effectness of the computation. We believe this nicely describes the motivation for the way in which the compile time and runtime operations described in the following paragraphs are wrapped within the agent state class in the C++ implementation.

Given a goal g to find a plan for, the function below filters the agent's plan library for those plans which have g as their triggering goal. The resulting set of relevant plans is returned. In the case of parametrised plans, each of the relevant plans is further passed the option parameter.

$$\begin{aligned}
&findRelevantPlans :: Goal \to Reader\ AgentState\ Plans \\
&findRelevantPlans\ (BGoal\ g) \quad = \mathbf{do}\ as \leftarrow ask \\
&\qquad\qquad\qquad\qquad\qquad\quad return\,\$\,filter\ (\lambda(BPlan\ g'\ _)\ g \equiv g')\ (pl\ as) \\
&findRelevantPlans\ (PGoal\ g\ o) = \mathbf{do}\ as \leftarrow ask \\
&\qquad\qquad\qquad\qquad\qquad\quad \mathbf{let}\ rps = filter\ (\lambda(PPlan\ g'\ _\ _) \to g \equiv g')\ (pl\ as) \\
&\qquad\qquad\qquad\qquad\qquad\quad \mathbf{let}\ rps' = map\ (\lambda\, p \to [p\ \{plan_on = o\}])\ rps \\
&\qquad\qquad\qquad\qquad\qquad\quad return\,\$\,foldr\ (^+)\ [\,]\ rps'
\end{aligned}$$

In C++, since the goal that a plan handles is known at compile time, the test for relevance of a given plan can also be performed at compile time. To that end, we first define a metafunction

class `is_relevant` which performs the check. It uses `boost::is_same` to check whether the goal being passed as a parameter corresponds with the triggering goal of the plan.

```
template <class Goal>
struct is_relevant {
  template <class Plan>
  struct apply {
    typedef typename boost::mpl::bool_<
      boost::is_same<Goal, typename Plan::goal>::value
    > type;
  };
};
```

`is_relevant` can be seen as representing the lambda function that is passed to the filter operation in the Haskell code above. The equivalent of filtering in Boost.MPL is `copy_if`; the overall filtering operation can thus be described as the following compile time calculation.

```
typedef typename boost::mpl::copy_if<
  PL,
  is_relevant<G>
>::type relevantPlans;
```

Applicability check

In the next step, the set of relevant plans is filtered with respect to applicability. Similar to the relevance check, the applicability check often requires information from an agent's current state, for example its belief base; the applicability check for a single plan can thus be represented as a function that accepts a plan and returns a Boolean value whilst also using information from the agent's current state.

$$isApplicable :: Plan \rightarrow Reader\ AgentState\ Bool$$

Determining the set of applicable plans is now straightforward. We just need to filter the set of relevant plans by applicability[3].

$$findApplicablePlans :: Plans \rightarrow Reader\ AgentState\ Plans$$
$$findApplicablePlans\ ps = filterM\ isApplicable\ ps$$

Applicability is a process in which the agent can decide whether its current *context* as defined by its knowledge allows for the execution of a given plan. If the context information is available at compile time (which will, for obvious reasons, rarely be the case), then the applicability check can also be performed at compile time using metaprogramming. But even in those cases where runtime information is necessary, metaprogramming can be usefully employed to provide the appropriate infrastructure for making an efficient decision at runtime, i.e. to assemble the respective types such that they allow for efficient runtime execution.

We start with the simplest possible type of applicability check: the identity. Here, all relevant plans are considered applicable. For obvious reasons, this type of applicability check can be easily performed at compile time. Nevertheless, since we do not know what subsequent selection mechanism the class is to be combined with, we also need to provide a runtime version

[3]Since *isApplicable* happens in the Reader monad, *filterM* has to be used instead of *filter*.

alongside the static one. The implementation is shown below. The class contains three different type definitions: (i) `RelevantPlans`, the input set of relevant plans, (ii) `CApplicablePlans`, the set of resulting applicable plans to be used by a subsequent *compile time* plan selection mechanism, and (iii) `RApplicablePlans`, the set of resulting applicable plans to be used by a subsequent *runtime* plan selection mechanism. The compile time version of the applicability check is implicit in the equivalent definition of `RelevantPlans` and `CApplicablePlans`; the runtime version of the check takes place in constructor `apply()` where, for each element in the input typelist of relevant plans, a concrete instance is created and stored in the runtime vector `applicablePlans`.

```
struct CTAllApplicable {
  template <class Plans, class, const char*>
  struct apply : CtrlFlow<Plans, void, g_sEmpty> {

    typedef Plans RelevantPlans;
    typedef Plans CApplicablePlans;
    typedef typename boost::make_variant_over<Plans>::type RApplicablePlans;

    std::vector<RApplicablePlans> applicablePlans;

    apply() {
      boost::mpl::for_each<Plans>([this](auto plan) {
        this->applicablePlans.push_back(plan);
      });
    }

    template <class AS>
    void computeApplicablePlans(AS& as) {}
  };
};
```

The example shown above supports both compile time and runtime applicability checking. In order to allow for efficient applicability checking at runtime, we subdivide the responsibilities in a hierarchy of classes. The topmost class is responsible for providing the basic static infrastructure. As described above, C++ only allows for homogeneous runtime containers; since each plan of the input list is represented by its own distinct type, plans thus have to be wrapped into a Boost.Variant. The class comprises two vectors, one for the input set of applicable plans and one for the resulting set of applicable plans. As described further below, those sets that pass the applicability check will then, at runtime, be moved from vector `relevantPlans` to vector `applicablePlans`. Also note that there is no type definition for subsequent compile time selection mechanisms. The reason is that, in the case below, we have already committed to a runtime-dependent solution, so there is no possibility to return to the static world.

```
struct RTApplicabilityBase {
  template <class Plans, class Type, const char* Param>
  struct apply : CtrlFlow<Plans,Type,Param> {

    typedef typename boost::make_variant_over<Plans>::type RelevantPlans;
    typedef RelevantPlans RApplicablePlans;
```

```
    apply() {
      boost::mpl::for_each<Plans>([this](auto plan) {
        this->relevantPlans.push_back(plan);
      });
    }

    apply(Plans plans)  {
      boost::fusion::for_each(plans, [this](auto plan) {
        this->relevantPlans.push_back(plan);
      });
    }

    std::vector<RelevantPlans> relevantPlans;
    std::vector<RApplicablePlans> applicablePlans;
  };
};
```

The previous class introduces the basic static skeleton for applicability checking. In the next step, the basic functionality for filtering relevant plans for applicability is introduced. This is, again, realised as a metafunction class. However, from this point, we need to start treating parametrised and non-parametrised plans differently. This is realised by means of template specialisation, as shown in the following high-level view on struct `RTApplicability2`.

```
struct RTApplicability2 {
  template <class Plans, class ParamType, const char* Param>
  struct apply : RTApplicabilityBase::template
    apply<Plans,ParamType,Param> { ... };

  template <class Plans, const char* Param>
  struct apply<Plans,void,Param> : RTApplicabilityBase::template
    apply<Plans,void,Param> { ... };
};
```

The parametrised case is detailed below. It contains a function `computeApplicable-Plans_` which filters the set of relevant plans by applicability. The function accepts two parameters: a reference to the agent state and an option parameter. The latter is passed on to each plan. Since the set elements are of type Boost.Variant, a *visitor mechanism* needs to be used for both setting the option parameter and checking for applicability. The visitor classes themselves are described further below.

```
template <class Plans, class Type, const char* Param>
struct apply : RTApplicabilityBase::apply<Plans,Type,Param> {
  apply() : RTApplicabilityBase::apply<Plans,Type,Param>() {}
  apply(Plans plans) : RTApplicabilityBase::apply<Plans,Type,Param>(plans) {}

  void setParam(Type const& _param) {
    for(auto& ap : this->relevantPlans)
      apply_visitor(setParameter<Type>(_param),ap);
  }
```

```
  template <class AS>
  void computeApplicablePlans_(AS& as, auto& f)
  {
    std::copy_if(
      this->relevantPlans.begin(),
      this->relevantPlans.end(),
      std::back_inserter(this->applicablePlans),
      [&as,&f,this](auto& el){
        apply_visitor(setParameter<Type>(f),el);
        return apply_visitor(isApplicable<AS,Type>(as,f),el);
    });
  }
};
```

The non-parametrised case is shown below. It differs from the parametrised case only with respect to the option parameter which is now missing.

```
template <class Plans, const char* Param>
struct apply<Plans,void,Param> : RTApplicabilityBase::apply<Plans,void,Param> {
  apply() : RTApplicabilityBase::apply<Plans,void,Param>() {}
  apply(Plans plans) : RTApplicabilityBase::apply<Plans,void,Param>(plans) {}

  template <class AS>
  void computeApplicablePlans_(AS& as)
  {
    std::copy_if(
      this->relevantPlans.begin(),
      this->relevantPlans.end(),
      std::back_inserter(this->applicablePlans),
      [&as,this](auto& el){
        return apply_visitor(isApplicable<AS,void>(as),el);
    });
  }
};
```

The bottommost class of the hierarchy realises the overall applicability check by inheriting from and utilising the aforementioned intermediate classes. Function `computeApplicablePlans` determines the set of all options in the agent's belief base. For each option, helper function `computeApplicablePlans_` (described above) is called. This achieves the desired plan instantiation effect.

```
template <class Plans, class Type, const char* Param>
struct apply : RTApplicability2::apply<Plans,Type,Param> {
  apply() : RTApplicability2::apply<Plans,Type,Param>() {}
  apply(Plans plans) : RTApplicability2::apply<Plans,Type,Param>(plans) {}

  template <class AS>
  void computeApplicablePlans(AS& as) {
    std::unordered_set<Type> foo;
    as.getBB().getBelief(Param,foo);
```

```
    for(auto& f : foo) {
      this->computeApplicablePlans_(as,f);
    }
  }
};
```

The non-parametrised case is trivial. Since the notion of options does not exist in this case, the computation of applicable plans reduces to a call of the aforementioned function `computeApplicablePlans_`.

```
template <class Plans, class Type>
struct apply<Plans,Type,g_sEmpty>
: RTApplicability2::apply<Plans,Type,g_sEmpty> {
  apply() : RTApplicability2::apply<Plans,Type,g_sEmpty>() {}
  apply(Plans plans) : RTApplicability2::apply<Plans,Type,g_sEmpty>(plans) {}

  template <class AS>
  void computeApplicablePlans(AS& as) {
    this->applicablePlans.reserve(this->relevantPlans.size());
    this->computeApplicablePlans_(as);
  }
};
```

It remains to briefly discuss the applicability visitor used in class `RTApplicability2`. As mentioned above, instances of Boost.Variant can only be accessed by means of compile time visitors. Consequently, if a plan is to be checked for applicability, an appropriate visitor is needed. Both parametrised and non-parametrised plans can be handled by the same visitor. We start with the parametrised case.

```
template <class AS, class ParamType>
struct isApplicable : boost::static_visitor<bool> {
  isApplicable(AS const& _as, ParamType const& _param)
  : as(_as),
    param(_param) {}

  template <class P>
  bool operator()(P const& plan) const {
    return plan._applicable(as,param);
  }

  AS const& as;
  ParamType param;
};
```

`isApplicable` accepts as type parameters the type of the state object as well as the type of an option parameter. The struct consists of a constructor which accepts instances of both the agent state and the parameter type and stores them in member variables. It also contains an implementation of the function call operator and thus represents a function object. The function call operator is parametrised with the type of plan to be handled and accepts an instance of

this plan type. The instance of the agent state as well as the parameter are forwarded to this plan instance object by calling its member function `_applicable`. Although not shown in the listing above, the member function `_applicable` forwards its two parameters to a global function `applicable` for which the framework provides a default implementation (returning *true*) that the user can overwrite in order to implement his own applicability check. The process of overwriting for the purpose of customisation is described in more detail in Section 5.4 below.

The applicability visitor for non-parametrised plans can now be realised as a simple specialisation of the template given above. It is equivalent to the previous one except that it does not allow for the existence of an option parameter.

```
template <class AS>
struct isApplicable<AS,void> : boost::static_visitor<bool> {
  isApplicable(AS const& _as)
  : as(_as) {}

  template <class P>
  bool operator()(P const& plan) const {
    return plan._applicable(as);
  }

  AS const& as;
};
```

Plan selection

Finally, from the set of applicable plans, a single plan needs to be selected for execution. There is a wide variety of possible plan selection mechanisms, so we start on an abstract level and represent, in Haskell, the selection as a function from a set of plans to a single plan.

$$selectPlan :: Plans \rightarrow Plan$$

In C++, selection is performed by means of a dedicated *selector* component which itself represents an executable component. To this end, it is implemented as a functor, i.e. it provides an implementation of the function call operator along with two functions `done` and `advance` (just like the basic actions described in Section 5.1.4).

Depending on (i) the type of information necessary for making a decision, and (ii) the nature of plans to be selected from, plan selection can be performed either at compile time or at runtime. As described in the previous section, in order to ensure compatibility, each compile time applicability checking mechanism needs to provide both compile time-processable and runtime-processable output. A subsequent compile time selection mechanism can thus use the former, a runtime selection mechanism the latter. A runtime applicability checking mechanism can only produce runtime-processable output and is thus only composable with a subsequent runtime selection mechanism.

By analogy with the applicability check, we start the description with the compile time case. One possible way of selecting a plan at compile time is to always choose the first element of a given list of plan types. An example implementation is shown below. Here, the idea of policy-based class design becomes apparent. The template accepts as its first parameter the type of the applicability check to be used. It then uses inheritance to incorporate its functionality and operate upon the same data structures. In this way, passing around values can be avoided.

```
struct CTHeadSelector {
  template <
    class Applicability,
    class Plans,
    class Type=void,
    const char* Param = g_sEmpty
  >
  struct apply : Applicability::template apply<Plans, Type, Param> {
    BOOST_MPL_ASSERT(( boost::mpl::is_sequence<Plans> ));
    typedef typename boost::mpl::deref<
      typename boost::mpl::begin<
        typename Applicability::template apply<Plans, Type, Param>::CApplicablePlans
      >::type
    >::type SelectedPlan;

    SelectedPlan plan;

    bool done() const { return plan.done(); }

    void advance() {
      if(!done())
        plan.advance();
    }

    template <class Env, class AS>
    bool operator()(Env& env, AS& as) {
      if(!done()) {
        std::cout << "Executing child plan" << std::endl << std::flush;
        bool b = plan(env, as);
        if(!b) {
          typedef Remove<BGoal<typename SelectedPlan::goal>> RemoveEvent;
          as.addEvent(RemoveEvent());
          return false;
        }
        return true;
      }
      return true;
    }
  };
};
```

The most interesting things happen in the type definition at the top of the listing. Here, the type of the resulting plan is set to be type of the first plan in the given input list of plans. At this point, we are still in the static world, i.e. there is no notion of an executable plan. This changes in the next line where a new instance of `SelectedPlan` is created. At this point, the bridge between compile time and runtime has been crossed. The instance is then used in the usual way, i.e. it is advanced using function `advance` or checked for completeness using function `done`.

Similar to the applicability check, plan selection may also happen entirely at runtime. There may be two reasons for that. First, due to its reliance upon runtime information, the applica-

bility may have already been performed at runtime; in this case, we have already crossed the bridge into the runtime world from which there is no escape. Or, second, the plan selection process itself may be dependent upon runtime information, e.g. using utility calculations which take into account the agent's knowledge.

The basic structure of a runtime selector is shown in the listing below. It has a strong resemblance with its compile time equivalent shown above. The only notable difference is in the way in which the type of the resulting selected plan is defined. Rather than defining a new type at compile time, the variant type of the members of the set of applicable plans inherited from the applicability mechanism is reused.

```
struct RTSelectorBase {
  template <
    class Applicability,
    class Plans,
    class Type=void,
    const char* Param = g_sEmpty
  >
  struct apply : Applicability::template apply<Plans,Type,Param> {

    apply()
    :   chosenPlan(-1), Applicability::template apply<Plans,Type,Param>() {}
    apply(Plans plans)
    :   chosenPlan(-1),
      Applicability::template apply<Plans,Type,Param>(plans) {}

    void advance() {
      if(!done())
        apply_visitor(advancePlan(), this->applicablePlans[this->chosenPlan]);
    }

    bool done() const {
      return(apply_visitor(isDone(), this->applicablePlans[this->chosenPlan]));
    }

    typedef typename Applicability::template
      apply<Plans,Type,Param>::ApplicablePlans SelectedPlan;
    int chosenPlan;
  };
};
```

The classes described above provide the basic infrastructure for runtime plan selection, but they do not yet allow for sophisticated selection strategies. A wide variety of concrete mechanisms are thinkable: random selection, utility-based selection, etc. A number of mechanisms has been implemented in the framework; they are described in more detail in Section 5.3.

It remains to discuss the overall process of finding a plan for a given goal. Following the descriptions above, it can be defined as a sequence of three steps: (i) determining the relevant plans, (ii) determining the applicable plans, and (iii) selecting a plan for execution.

$$\begin{aligned} &\mathit{findPlan} :: \mathit{Goal} \rightarrow \mathit{Reader}\ \mathit{AgentState}\ \mathit{Plan} \\ &\mathit{findPlan}\ g = \mathbf{do}\ \mathit{as} \leftarrow \mathit{ask} \end{aligned}$$

$$
\begin{aligned}
&rp \leftarrow \mathit{findRelevantPlans}\ g \\
&ap \leftarrow \mathit{findApplicablePlans}\ rp \\
&\mathit{return}\,\$\,\mathit{selectPlan}\ ap
\end{aligned}
$$

In C++, the overall process of finding a plan can now be implemented as a combination of compile time and runtime computation. In the first step, the set of relevant plans is determined at runtime; in the second step, each of the resulting plans is *expanded* at compile time (described in more detail in the next section); in the third and final step, the set of expanded plans is then passed to the user-defined plan selector, along with the equally user-defined applicability checking mechanism. This is shown below.

```
typedef typename boost::mpl::copy_if<
  PL,
  is_relevant<G>
>::type relevantPlans;
BOOST_MPL_ASSERT(( boost::mpl::is_sequence<relevantPlans> ));

typedef typename boost::mpl::transform<
  relevantPlans,
  expandPlanItem<PL,Applicability,Selector,Type>
>::type expRelevantPlans;
BOOST_MPL_ASSERT(( boost::mpl::is_sequence<expRelevantPlans> ));

typedef typename Selector::template apply<
  Applicability,
  expRelevantPlans,
  OptParamType,
  OptParamName
> type;
```

Plan expansion

As motivated in Chapter 3, C++ metaprogramming can be usefully employed to replace subgoals in plans with more concrete entities — either entire plans if the selection can be performed at runtime, or problem-specific function objects that are tailored to efficient selection at runtime. Plan expansion is a recursive process; we start the description with the expansion of a single plan item. Three cases need to be distinguished: if the plan item is an action, nothing is to be done; if the plan item is a sequential goal addition, a parallel addition action is created; if the plan item is a sequential goal addition, then a plan for the given goal is found and expanded recursively, as shown below.

$$
\begin{aligned}
&\mathit{expandPlanItem} :: \mathit{PlanItem} \rightarrow \mathit{Reader}\ \mathit{AgentState}\ \mathit{Intention} \\
&\mathit{expandPlanItem}\ (\mathit{Ac}\ a) \ = \mathit{return}\,\$\,\mathit{BasicAc}\ a \\
&\mathit{expandPlanItem}\ (\mathit{Par}\ g) = \mathit{return}\,\$\,\mathit{ParAdd}\ g \\
&\mathit{expandPlanItem}\ (\mathit{Seq}\ g) \ = \mathbf{do}\ as \leftarrow \mathit{ask} \\
&\qquad\qquad\qquad\qquad\quad plan \leftarrow \mathit{findPlan}\ g \\
&\qquad\qquad\qquad\qquad\quad \mathit{expandPlan}\ plan
\end{aligned}
$$

In C++, plan expansion happens at compile time. The function shown above can thus be realised as a metafunction class. For clarity, the logic is split up into two nested functions. The

outer function (shown below) deals with the expansion of the three types of plan items, as described above.

```
template <class PL, class Applicability, class Selector, class PParamType=void>
struct expandPlanItem {
  BOOST_MPL_ASSERT(( boost::mpl::is_sequence<PL> ));

  template <class P>
  struct apply {
    typedef P type;
  };

  template <int Name, class Goal, class Body>
  struct apply<Plan<Name,Goal,Body>> {
    typedef typename boost::mpl::transform<
      Body,
      expandPlanItem<PL,Applicability,Selector,PParamType>
    >::type NewBody;
    BOOST_MPL_ASSERT(( boost::mpl::is_sequence<NewBody> ));

    typedef Sequence<Name, Goal, NewBody, PParamType> type;
  };

  // sequential goal => expand immediately
  template <class Goal>
  struct apply<Seq<Goal>> {
    typedef typename expandPlanItem_<
      PL,
      Applicability,
      Selector,
      PParamType
    >::template apply<Goal>::type type;
  };

  // parallel goal => turn into action
  template <class Ev>
  struct apply<Par<Ev>> {
    typedef ParAdd<Ev> type;
  };
};
```

The inner function deals with the special case of sequential goal addition, where a distinction between basic and parametrised goals needs to be made. In both cases, relevant goals are identified and then passed as a parameter to a user-defined plan selector, along with the user-defined applicability checking mechanism. In the case of parametrised plans, the option parameter further needs to be passed to the plan selector.

```
template <class PL, class Applicability, class Selector, class PParamType=void>
struct expandPlanItem_ {
  BOOST_MPL_ASSERT(( boost::mpl::is_sequence<PL> ));
```

```
  template <class P>
  struct apply {
    typedef P type;
  };

  // sequential basic goal => expand immediately
  template <class G>
  struct apply<BGoal<G>> {
    typedef typename boost::mpl::copy_if<
      PL,
      is_relevant<G>
    >::type relevantPlans;
    BOOST_MPL_ASSERT(( boost::mpl::is_sequence<relevantPlans> ));

    typedef typename boost::mpl::transform<
      relevantPlans,
      expandPlanItem<PL,Applicability,Selector,PParamType>
    >::type newRelevantPlans;
    BOOST_MPL_ASSERT(( boost::mpl::is_sequence<newRelevantPlans> ));

    typedef typename Selector::template apply<
      Applicability,
      newRelevantPlans,
      PParamType
    > type;
  };

  // sequential parametrised goal => expand immediately
  template <class G, const char* Param, class Type>
  struct apply<PGoal<G,Param,Type>> {
    typedef typename boost::mpl::copy_if<
      PL,
      is_relevant<G>
    >::type relevantPlans;
    BOOST_MPL_ASSERT(( boost::mpl::is_sequence<relevantPlans> ));

    typedef typename boost::mpl::transform<
      relevantPlans,
      expandPlanItem<PL,Applicability,Selector,Type>
    >::type newRelevantPlans;
    BOOST_MPL_ASSERT(( boost::mpl::is_sequence<newRelevantPlans> ));

    typedef typename Selector::template apply<
      Applicability,
      newRelevantPlans,
      Type,
      Param
    > type;
  };
};
```

Once the expansion of single plan items is defined, the expansion of entire plans can be described in a straightforward way. We simply need to iterate over the plan body, expand each item separately, and return the resulting sequence of intentions[4].

$$
\begin{aligned}
&\mathit{expandPlan} :: \mathit{Plan} \to \mathit{Reader}\ \mathit{AgentState}\ \mathit{Intention} \\
&\mathit{expandPlan}\ p = \mathbf{do}\ \mathit{as} \leftarrow \mathit{ask} \\
&\qquad\qquad\qquad\quad \mathit{ep} \leftarrow \mathit{mapM}\ \mathit{expandPlanItem}\ (\mathit{plan\_pb}\ p) \\
&\qquad\qquad\qquad\quad \mathit{return}\,\$\,\mathit{Sequence}\ \mathit{ep}
\end{aligned}
$$

As mentioned above, plan expansion happens at compile time in C++. To that end, function `expandPlan` is realised as the following type transformation upon the agent's plan library `PL`.

```
typedef typename boost::mpl::transform<
   PL,
  expandPlanItem<PL,Selector>
>::type transformed_plans;
```

5.2.3 Event processing

With respect to the processing of events, two cases need to be handled: goal addition and goal removal. In the case of goal addition, we first determine a plan for the given goal using the functions described further above and perform plan expansion; the resulting intention is then added to the agent's intention store. In the case of goal removal, we simply filter out those addition events from the event store which correspond with the goal to be removed.

$$
\begin{aligned}
&\mathit{processEvent} :: \mathit{Event} \to \mathit{State}\ \mathit{SystemState}\ () \\
&\mathit{processEvent}\ (\mathit{Add}\ g) = \mathbf{do}\ \mathit{ss} \leftarrow \mathit{get} \\
&\qquad\qquad\qquad\qquad\qquad\quad \mathbf{let}\ \mathit{plan} = \mathit{runReader}\ (\mathit{findPlan}\ g)\,\$\,\mathit{as}\ \mathit{ss} \\
&\qquad\qquad\qquad\qquad\qquad\quad \mathbf{let}\ \mathit{res} = \mathit{runReader}\ (\mathit{expandPlan}\ \mathit{plan})\,\$\,\mathit{as}\ \mathit{ss} \\
&\qquad\qquad\qquad\qquad\qquad\quad \mathbf{let}\ \mathit{is'} = (\mathit{is}\,\$\,\mathit{as}\ \mathit{ss})^{+}[\mathit{res}] \\
&\qquad\qquad\qquad\qquad\qquad\quad \mathbf{let}\ \mathit{as'} = (\mathit{as}\ \mathit{ss})\,\{\mathit{is} = \mathit{is'}\} \\
&\qquad\qquad\qquad\qquad\qquad\quad \mathit{put}\ \mathit{ss}\,\{\mathit{as} = \mathit{as'}\} \\
&\mathit{processEvent}\ (\mathit{Remove}\ g) = \mathbf{do}\ \mathit{ss} \leftarrow \mathit{get} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathbf{let}\ \mathit{eq'} = \mathit{filter}\ (\boldsymbol{\lambda}\ i \to i \not\equiv \mathit{Add}\ (g))\ (\mathit{eq}\,\$\,\mathit{as}\ \mathit{ss}) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathbf{let}\ \mathit{as'} = (\mathit{as}\ \mathit{ss})\,\{\mathit{eq} = \mathit{eq'}\} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{put}\ \mathit{ss}\,\{\mathit{as} = \mathit{as'}\}
\end{aligned}
$$

In C++, the processing function described above can be formulated as a metafunction class, as shown in the listing below.

```
template <
  class PL,
  class Applicability=RTApplicability,
  class Selector=RandomSelector
>
struct expandEvent {
  template<class Ev>
  struct apply {};
```

[4]Since *expandPlanItem* takes place in the Reader monad, *mapM* needs to be used instead of *map*.

```
  template<class Goal>
  struct apply<Add<BGoal<Goal>>> {
    BOOST_MPL_ASSERT(( boost::mpl::is_sequence<PL> ));

    typedef typename boost::mpl::copy_if<
      PL,
      is_relevant<Goal>
    >::type relevantPlans;
    BOOST_MPL_ASSERT(( boost::mpl::is_sequence<relevantPlans> ));

    typedef typename boost::mpl::transform<
      relevantPlans,
      expandPlanItem<PL,Applicability,Selector>
    >::type newRelevantPlans;

    typedef typename Selector::template apply<
      Applicability,
      newRelevantPlans
    > type;
  };

  template<class Goal>
  struct apply<Remove<Goal>> {
    typedef Rem<Goal> type;
  };
};
```

The overall processing of events can then be described as a simple mapping of the individual processing function over the event store.

$$
\begin{aligned}
&\mathit{processEvents} :: \mathit{State}\ \mathit{SystemState}\ () \\
&\mathit{processEvents} = \mathbf{do}\ \mathit{ss} \leftarrow \mathit{get} \\
&\qquad\qquad\qquad\quad \mathit{mapM}\ \mathit{processEvent}\ (\mathit{eq}\,\$\,\mathit{as}\ \mathit{ss}) \\
&\qquad\qquad\qquad\quad \mathit{return}\ ()
\end{aligned}
$$

In C++, event processing is an inherently runtime-dependent operation, so we can realise `processEvents` as a simple functor as shown below. The function call operator is parametrised with the agent state as well as with the event selection mechanism to be used. The mapping operation in the Haskell code above is replaced with a conventional while-loop. In order to access events of type Boost.Variant, a visitor mechanism is used.

```
struct processEvents {
  template <class AS, class EventSelector>
  void operator()(AS& as, EventSelector es) {
    while(!as.events.empty()) {
      auto event = es(as);
      apply_visitor(eventVisitor<AS>(as), *event);
      as.events.erase(event);
    }
  }
```

```
};
```

The event visitor itself is shown in the listing below. Its only functionality is to create a new intention of the overall intention type and adds it to the agent's intention store.

```
template <class AS>
struct eventVisitor : boost::static_visitor<> {
  eventVisitor(AS& _as)
  : as(_as)
  {}

  template <class Ev>
  void operator()(Ev const& ev) const {
    typedef typename AS::expand_event::template apply<Ev>::type new_intention;
    as.addIntention(new_intention());
  }

  AS& as;
};
```

One possible implementation of an event selection mechanism is shown below. It follows the FIFO principle, i.e. it simply creates an iterator to the first element in the vector and returns it to the calling function.

```
struct FIFO_EventSelector {
  template <class AS>
  auto operator()(AS& as) {
    auto itEvent = as.events.begin();
    return itEvent;
  }
};
```

5.2.4 Intention execution

Similar to the previous steps, we start with the treatment of an individual intention. We need to distinguish between four cases: (i) basic action, (ii) goal removal, (iii) parallel goal addition, and (iv) sequences. In the case of a basic action, we simply perform the state changes required by the action. This is a custom action and is thus left unspecified here. In the case of a goal removal, we iterate over the set of events and filter out the goal addition events that correspond with the goal to be removed. In the case of parallel goal addition, we simply add a goal addition event to the agent's event store. And, finally, in the case of a sequence, we recursively execute the intentions wrapped within the sequence.

$$
\begin{aligned}
&\mathit{executeIntention} :: \mathit{Intention} \to \mathit{State}\ \mathit{SystemState}\ () \\
&\mathit{executeIntention}\ (\mathit{BasicAc}\ a) \quad = \textbf{do}\ \mathit{return}\ () \\
&\mathit{executeIntention}\ (\mathit{Rem}\ g) \qquad = \textbf{do}\ ss \leftarrow \mathit{get} \\
&\qquad\qquad\qquad\qquad\quad \textbf{let}\ eq' = \mathit{filter}\ (\lambda\, x \to x \not\equiv \mathit{Add}\ (g))\ (eq\,\$\,as\ ss) \\
&\qquad\qquad\qquad\qquad\quad \textbf{let}\ as' = (as\ ss)\,\{\, eq = eq' \,\} \\
&\qquad\qquad\qquad\qquad\quad \mathit{put}\ ss\,\{\, as = as' \,\}
\end{aligned}
$$

$$
\begin{aligned}
&executeIntention\ (ParAdd\ g) \quad = \textbf{do}\ ss \leftarrow get \\
&\qquad\qquad\qquad\qquad\qquad\quad \textbf{let}\ eq' = (eq\,\$\,as\ ss)^{+}[Add\ (g)] \\
&\qquad\qquad\qquad\qquad\qquad\quad \textbf{let}\ as' = (as\ ss)\,\{eq = eq'\} \\
&\qquad\qquad\qquad\qquad\qquad\quad put\ ss\,\{as = as'\} \\
&executeIntention\ (Sequence\ is) = \textbf{do}\ mapM\ executeIntention\ is \\
&\qquad\qquad\qquad\qquad\qquad\quad return\ () \\
\end{aligned}
$$

The overall execution of intentions can thus be described as a simple mapping of the individual processing function over the agent's intention store.

$$
\begin{aligned}
&executeIntentions :: State\ SystemState\ () \\
&executeIntentions = \textbf{do}\ ss \leftarrow get \\
&\qquad\qquad\qquad\qquad mapM\ executeIntention\ (is\,\$\,as\ ss) \\
&\qquad\qquad\qquad\qquad return\ () \\
\end{aligned}
$$

In C++, a distinction between different types of intentions has been made implicit by means of dynamic polymorphism. It is common to all intentions that they are represented as functors, i.e. they provide a function call operator which allows them to be executed. The execution logic is wrapped inside the function call operator.

In the case of goal removal, all addition events referring to the respective goal are removed from the agent's event store by calling member function `remEvent` on the agent state.

```
template <class Goal>
struct Rem : public BAction {
  template <class Env, class AS>
  bool operator()(Env&, AS& as) const {
    typedef Add<Goal> GoalAddition;
    as.remEvent(GoalAddition());
    return true;
  }
};
```

In the case of parallel goal addition, an addition event for the respective goal is added to the agent's event store by calling member function `addEvent` on the agent state.

```
template <class Goal>
struct ParAdd : BAction {
  template <class Env, class AS>
  bool operator()(Env&, AS& as) const {
    as.addEvent(Add<Goal>());
    return true;
  }
};
```

Finally, in the case of a sequence, the first element of the wrapped set of intentions is executed recursively. Again, since intentions are stored in a container of type Boost.Variant, a visitor mechanism is to be used.

```
template <int Name, class Goal, class Plans, class ParamType>
struct SequenceBase : CtrlFlow<Plans, ParamType> {
```

```
  ...

  template <class Env, class AS>
  bool operator()(Env& env, AS& as) {
    if(!done()) {
      return apply_visitor(executePlan<Env,AS>(env,as), vec.front());
    }
  }

  ...
};
```

The visitor itself is shown below. The logic is straightforward: it simply calls the respective plan's function call operator and thus triggers the execution of the plan.

```
template <class Env, class AS>
struct executePlan : boost::static_visitor<bool> {
  executePlan(Env& _env, AS& _as)
  : as(_as), env(_env) {}

  template <class Plan>
  bool operator()(Plan& plan) const {
    return plan(env,as);
  }

  AS& as;
  Env& env;
};
```

The overall operation of execution the intentions can now be defined as the following functor. Similar to the event processing function described above, the function call operator is parametrised with the agent state as well as with the intention selection mechanism to be used.

```
struct executeIntentions {
  template <class Env, class AS, class IntentionSelector>
  void operator()(Env& env, AS& as, IntentionSelector is) {
    auto intention = is(as);
    apply_visitor(executePlan<Env,AS>(env,as), *intention);
    apply_visitor(advancePlan(), *intention);
    if(apply_visitor(isDone(), *intention))
      as.intentions.erase(intention);
  }
};
```

One possible implementation of an intention selection mechanism is shown below. Similar to the case of event processing described in Section 5.2.3 above, it follows the FIFO principle, i.e. it simply picks the first element from the vector (an iterator to it, to be precise) and returns it to the calling function.

```
struct FIFO_IntentionSelector {
```

```
  template <class AS>
  auto operator()(AS& as) {
    auto itIntention = as.intentions.begin();
    return itIntention;
  }
};
```

5.2.5 Overall update cycle

Using the functions defined in the previous sections, a single update step can now be described succinctly as a sequence of three steps operating upon the agent state: perception, event processing, and intention execution. The integer parameter in the `step` function denotes the current time step but is currently not used.

$$
\begin{aligned}
&\mathit{step} :: \mathit{Int} \rightarrow \mathit{State}\ \mathit{SystemState}\ () \\
&\mathit{step}\ t = \mathbf{do}\ \mathit{perceive} \\
&\qquad\qquad\quad \mathit{processEvents} \\
&\qquad\qquad\quad \mathit{executeIntentions}
\end{aligned}
$$

In C++, this functionality can be realised in a straightforward way by implementing `step` as a functor that is parametrised with (i) the agent state, (ii) the environment, (iii) the event selection mechanism, and (iv) the intention selection mechanism.

```
struct step {
  template <
    class AS,
    class Env,
    class EventSelector,
    class IntentionSelector
  >
  void operator()(
    AS& as,
    Env& env,
    EventSelector es,
    IntentionSelector is
  ) const {
    env.perceive(as);
    processEvents()(as, es);
    executeIntentions()(env, as, is);
    as.tick();
  }
};
```

The overall simulation of a single agent for a number of ticks can then be realised as a simple mapping operation over a list of time steps.

$$
\begin{aligned}
&\mathit{simulateAgent} :: \mathit{State}\ \mathit{SystemState}\ () \\
&\mathit{simulateAgent} = \mathbf{do}\ \mathbf{let}\ \mathit{ticks} = [1\,..\,100] \\
&\qquad\qquad\qquad\qquad\quad \mathit{mapM}\ \mathit{step}\ \mathit{ticks} \\
&\qquad\qquad\qquad\qquad\quad \mathit{return}\ ()
\end{aligned}
$$

In C++, the mapping operation is replaced with a for-loop.

```
void simulateAgent() {
  for(auto i: irange(0,100)) {
  step()(
    state,
    env,
    FIFO_EventSelector(),
    FIFO_IntentionSelector()
  );
  }
}
```

This concludes the description of the operative side of the framework. The C++ implementation illustrates the gradual shift from compile time to runtime computation during the agent's update cycle. For example, relevance checking was realised as a pure compile time function; applicability checking and plan selection were represented as hybrid solutions that allow for both compile time and runtime execution; event processing and intention execution were realised as pure runtime computations.

5.3 Plan selection mechanisms

Plan selection and transformation as described in the previous sections can be seen as realising the *practical reasoning* that the BDI architecture has been designed for: pursuit of long-term goals combined with reactive behaviour and re-planning. However, in some situations, a higher-order form of reasoning will be required. Consider, for example, the case when there is more than one relevant and applicable plan. Which of those plans should be chosen for execution? A higher-order decision-making process is required to solve this problem.

Plan selection: an overview

Before delving into the details of plan selection, it is useful to emphasise that there is a strong correspondence between plan selection and the notion of *control flow nodes* in the context of behaviour trees, as described in Section 3.1. In a behaviour tree, control flow nodes describe how the children of a node are to be executed. One might want, for example, to execute them in fixed or random order, or one might want to pick one particular element (e.g. based on a utility calculation as is very common in agent-based modelling) and execute it. There are clearly countless ways to perform a selection on a given set of plans; it is, however, common to all of them that they act as *functions* that accept a set of plans as an input, performs a selection, and produces another set of output (typically a unary set) for subsequent execution.

This leads to the notion of a *selector* which has been described briefly in the context of behaviour trees in Section 3.1 and elaborated upon in Section 5.2.2. The notion of a selector in the context of the BDI framework is based upon the idea of selecting a plan *at runtime*. If the selection criterion is based on static information (e.g. predetermined priority values), then there is no reason for a dedicated selection mechanism; in this case, the selection can be performed as part of the compile-time plan transformation process described in Section 5.2.2. However, if runtime information is required to make the decision (e.g. by means of particular beliefs obtained during the course of the simulation), then the decision has to be postponed until the simulation is executed.

The structural basis for the realisation of concrete runtime selectors has already been described in Section 5.2.2 by means of base class `RTSelectorBase`. In the framework, three different selectors have been implemented: one random selector and two utility-based selectors. They are described in the following paragraphs.

Random selector

Random selection represents the simplest selection mechanism. Given a set of plans, the random selector picks one plan with uniformly distributed probability. The code of the selector is shown in Listing 5.1. It is realised as a functor. Unless execution has already finished (as denoted by the return value of member function `done`), the randomly chosen plan is advanced by one step. As described further above, this happens by calling the `advance` function of the plan using a visitor mechanism. The Boolean return value of the advance visitor denotes success or failure during the execution; in the case of failure, the triggering goal associated with the plan is removed from the agent's event store using the `removeGoal` visitor. The visitor itself is shown in the listing below. Its only purpose is to create a goal removal event and add it to the agent's event store.

```
template <class AS>
struct removeGoal : boost::static_visitor<> {
  removeGoal(AS& _as)
  : as(_as)
  {}

  template <class Plan>
  void operator()(Plan& plan) const   {
    typedef RGoal<typename Plan::event> RemoveEvent;
    as.addEvent(RemoveEvent());
  }

  AS& as;
};
```

Utility selectors: core functionality

Choosing a plan using a simple probability evaluation is often not sufficient in a simulation context. Instead, as described in Section 2.1, it is often necessary to base the decision upon which plan to pursue on more complex utility calculations. For example, in the context of the party scenario described in Section 3.2, deciding whether to get the money from the ATM or from a friend may depend on the agent's evaluation as to which of the two options is more convenient in the current situation or, alternatively, whether the friend is currently at home at all. When assigning different utility values to different plan options, a number of scenarios is thinkable: we may *always* pick the plan with the highest utility, we may give the plans with higher utility *more priority* during the selection, etc. The choice of mechanism depends on the concrete scenario to be modelled.

In order to represent those different mechanisms, a further level of abstraction is added to the class hierarchy by encapsulating the common core functionality of different utility selection mechanisms into a dedicated base class. The code is shown in Listing 5.2. The class is parametrised with (i) the applicability selection mechanism to incorporate, (ii) the set of plans

Listing 5.1: The random selector class

```
struct RandomSelector {
  template <
    class Applicability,
    class Plans,
    class Type=void,
    const char* Param = g_sEmpty
  >
  struct apply : RTSelectorBase::template apply<Applicability, Plans,Type,Param> {
    typedef typename RTSelectorBase::template
      apply<Applicability,Plans,Type,Param>::SelectedPlan SelectedPlan;

    apply() : RTSelectorBase::template apply<Applicability, Plans,Type,Param>() {}
    apply(Plans plans)
    : RTSelectorBase::template apply<Applicability, Plans,Type,Param>(plans) {}

    template <class AS>
    int pickBestPlan(AS& as) const {
      this->getApplicablePlans(as);
      std::uniform_int_distribution<int> dist(0,this->applicablePlans.size()-1);
      return dist(rng);
    }

    template <class AS>
    bool operator()(AS& as) {
      if(this->chosenPlan == -1)
        this->chosenPlan = pickBestPlan(as);
      assert(this->chosenPlan != -1);

      if(!this->done()) {
        auto& cp = this->applicablePlans[this->chosenPlan];
        bool b = apply_visitor(executePlan<AS>(as), cp);
        if(!b) {
          // remove failed plan from list of applicable plans
          this->removePlanFromApplicable(cp);
          // if there are still applicable plans left, pick one randomly
          if(this->applicablePlans.size() > 0) {
            std::uniform_int_distribution<int> dist(0,this->applicablePlans.size()-1);
            this->chosenPlan = dist(rng);
            operator()(as);
          }
          // else, throw failure event
          else {
            // enqueue event for plan failure
            apply_visitor(removeGoal<AS>(as),cp);
            return false;
          }
        }
        return true;
      }
      return true;
    }
  };
};
```

Listing 5.2: Utility selector base class

```
template <
  class Applicability,
  class Plans,
  class Type,
  const char* Param
>
struct UtilitySelectorBase
: RTSelectorBase::template apply<Applicability, Plans,Type,Param> {
  typedef typename RTSelectorBase::template
   apply<Applicability,Plans,Type,Param>::SelectedPlan SelectedPlan;

  UtilitySelectorBase() :
    RTSelectorBase::template apply<Applicability, Plans,Type,Param>() {}
  UtilitySelectorBase(Plans plans) :
    RTSelectorBase::template apply<Applicability, Plans,Type,Param>(plans) {}

  template <class AS>
  float _utility(AS& as, auto& plan) const {
    return apply_visitor(getUtility<AS,Type>(as,this->param), plan);
  }
};

template <
  class Applicability,
  class Plans,
  const char* Param
>
struct UtilitySelectorBase<Applicability, Plans,void,Param>
: RTSelectorBase::template apply<Applicability, Plans,void,Param> {
  typedef typename RTSelectorBase::template
   apply<Applicability, Plans,void,Param>::SelectedPlan SelectedPlan;

  UtilitySelectorBase()
  : RTSelectorBase::template apply<Applicability, Plans,void,Param>() {}
  UtilitySelectorBase(Plans plans)
  :  RTSelectorBase::template apply<Applicability, Plans,void,Param>(plans) {}

  template <class AS>
  float _utility(AS& as, auto& plan) const {
    return apply_visitor(getUtility<AS,void>(as), plan);
  }
};
```

to be selected from, (iii) the option parameter type, and (iv) the option parameter name (the last two parameters only apply in the case of parametrised plans). Similar to each other plan, the base class provides a method `_utility` which visits the plan wrapped within the selector and calls the plan's own `_utility` function. This recursive delegation process ends as soon as function `_utility` inside a sequence has been reached. As described further above, sequences represent the runtime equivalents of declarative plans and can thus be considered the leaf nodes in the plan hierarchy; as shown at the bottom of Listing C.3 in the Appendix, function `_utility` calls functor `utility` which contains the actual utility evaluation for the respective plan and can be customised by the user. This is described in further detail in Section 5.4.

Maximum utility selector

The first strategy when selecting plans based on a utility evaluation is always to pick the plan with the highest utility. This is the purpose of class `MaxUtilitySelector` shown in Listing 5.3. In addition to those functions inherited from `UtilitySelectorBase`, the class contains a function `pickBestPlan` which performs the utility comparison and returns the plan with the highest utility value. This plan is then stored as the current plan and used as the basis for subsequent execution.

Utility evaluation selector

Rather than always strictly selecting the plan with the highest utility, it is often more useful to just give those plans with higher utility *higher priority*, yet without categorically disallowing the execution of plans with lower utility. This is the purpose of the second type of utility selector represented by class `UtilityEvalSelector`. The code is shown in Section 5.4. It differs from `MaxUtilitySelector` only with respect to the selection logic in method `pickBestPlan`. Rather than simply returning the plan with the highest utility, two steps are performed: (i) all plans including their utility values are thrown into a temporary data structure, and (ii) from the data structure, a plan is picked randomly based on its relative utility. This increases the likelihood of those plans with higher utility being chosen, yet it still leaves open the possibility for plans with lower utility to be selected.

Clearly, in addition to the different utility evaluation *mechanisms* described above, concrete *evaluation functions* must be defined by the user and integrated into the framework. This is described in further detail in Section 5.4 below.

5.4 Customisation

The framework described in the previous sections provides a number of customisation points by means of which the user can insert custom logic. In particular and corresponding with the description in Section 4.3, the framework can be customised with respect to the following aspects:

Beliefs: Beliefs of arbitrary type can be integrated into the reasoning process.

Environment: The internal dynamics of the environment can be formulated in a high-level programming language (C++).

Perception: The agents' perception of the environment can be formulated as a function of the environment in a high-level programming language (C++).

Listing 5.3: Maximum utility selector

```
struct MaxUtilitySelector {
  template <
    class Applicability,
    class Plans,
    class Type=void,
    const char* Param=g_sEmpty
  >
  struct apply : UtilitySelectorBase<Applicability, Plans,Type,Param> {
    typedef typename UtilitySelectorBase<Applicability, Plans,Type,Param>
      ::SelectedPlan SelectedPlan;

    apply() : UtilitySelectorBase<Applicability, Plans,Type,Param>() {}
    apply(Plans plans)
    : UtilitySelectorBase<Applicability, Plans,Type,Param>(plans) {}

    template <class AS>
    int pickBestPlan(AS& as) const {
      int _chosenPlan=-1;
      float maxSoFar=-1;
      for(int idx=0; idx<this->applicablePlans.size(); idx++) {
        float ut = this->_utility(as, this->applicablePlans[idx]);
        if(ut > maxSoFar) {
          _chosenPlan = idx;
          maxSoFar = ut;
        }
      }
      return _chosenPlan;
    }

    template <class AS>
    bool operator()(AS& as) {
      if(this->chosenPlan==-1) {
        this->computeApplicablePlans(as);
        this->chosenPlan = pickBestPlan(as);
      }
      assert(this->chosenPlan != -1);

      if(!this->done()) {
        auto& cp = this->applicablePlans[this->chosenPlan];
        bool b = apply_visitor(executePlan<AS>(as), cp);
        if(!b) {
          // enqueue event for plan failure
          apply_visitor(removeGoal<AS>(as),cp);
          return false;
        }
        return true;
      }
      return true;
    }
  };
};
```

Listing 5.4: Utility evaluation selector

```
struct UtilityEvalSelector {
  template <
    class Applicability,
    class Plans,
    class Type=void,
    const char* Param=g_sEmpty
  >
  struct apply : UtilitySelectorBase<Applicability, Plans,Type,Param> {
    typedef typename UtilitySelectorBase<Applicability, Plans,Type,Param>
      ::SelectedPlan SelectedPlan;

    apply() : UtilitySelectorBase<Applicability, Plans,Type,Param>() {}
    apply(Plans plans)
    : UtilitySelectorBase<Applicability, Plans,Type,Param>(plans) {}

    template <class AS>
    int pickBestPlan(AS& as) {
      // aggregate utility values
      std::map<float,int> utilityMap;
      float total = 0;
      for(int i=0; i<this->applicablePlans.size(); i++) {
        float ut = this->_utility(as, this->applicablePlans[i]);
        total += ut;
        utilityMap[total] = i;
      }
      // perform utility evaluation
      std::uniform_real_distribution<> dist(0,1);
      float rand = dist(rng);
      for(auto it=utilityMap.begin(); it!=utilityMap.end(); ++it) {
        if(rand<(it->first/total)) {
          return it->second;
        }
      }
      return -1;
    }

    template <class AS>
    bool operator()(AS& as) {
      if(this->chosenPlan==-1) {
        this->computeApplicablePlans(as);
        this->chosenPlan = pickBestPlan(as);
      }
      assert(this->chosenPlan != -1);

      if(!this->done()) {
        auto& cp = this->applicablePlans[this->chosenPlan];
        bool b = apply_visitor(executePlan<AS>(as), cp);
        if(!b) {
          // enqueue event for plan failure
          apply_visitor(removeGoal<AS>(as),cp);
          return false;
        }
        return true;
      }
      return true;
    }
  };
};
```

Actions: The internal dynamics of agent actions can be formulated in a high-level programming language (C++).

Event selection: The mechanism based on which events from the agent's event vector are selected for execution can be chosen by the user.

Plan selection: Customisation with respect to plan selection can be further subdivided into three subproblems: (i) the evaluation function for plan applicability, (ii) the actual plan selection mechanism, and (iii) the utility calculations for plans. All of them are described in further detail in the paragraphs below.

Intention selection: The mechanism based on which intentions from the agent's intention vector are selected for execution can be chosen by the user.

Applicability evaluation: As part of the deliberation process, available plans need to be checked for relevance and applicability. The relevance check is straightforward and solely based upon the equivalence of the goal to be handled and the triggering event of the plan under consideration. The applicability check is slightly more complex. Rather than being based on static aspects such as type information, applicability is dependent upon the current context that an agent finds itself in. It is obvious that, in general, such a decision cannot be made at compile time. It is thus necessary to include a mechanism into the framework, according to which runtime applicability checks can be injected by the user. As described in Section 5.2.2 above, the applicability check is performed by calling a plan-specific function `_applicable`. For sequences (the leaf nodes in the plan hierarchy), this function further applies visitor `applicable` to the plan wrapped within the sequence. In order to customise the applicability check, the visitor is parametrised with the plan that it is referring to and can thus be provided by the user.

An example of a custom applicability check in the context of the product purchasing example is given in Listing 5.5. Here, the applicability of a particular purchase plan `BuyProductP` is determined by the value of three attributes of the respective product: *availability*, *price*, and *healthiness*. Information relating to the attributes is read from the agent's belief base and used to create a shortlist of products for subsequent utility evaluation; since the belief base itself is influenced by the agent's environment, this information is inherently runtime-specific.

Plan selection mechanism: Plan selection as a way to perform meta reasoning has been discussed extensively in Section 5.3 above. It allows a user to specify the way in which an agent can resolve potential conflicts or ambiguities in plan selection. The customisation of plan selection is realised by means of dynamic polymorphism: the framework contains a set of base classes that concrete mechanisms must inherit from in order to inherit the core functionality. Examples of concrete plan selection mechanisms were given in Section 5.3. The plan selection can then be activated by passing it as a type parameter to class `AgentState`, as described in Section 5.5 below.

Utility evaluation: The customisation of utility evaluation is realised by a combination of (i) dynamic polymorphism in the form of a set of base classes as described in the previous section, and (ii) static polymorphism in the form of a parametrised functor that realises the mapping between a plan and a utility calculation. An example utility functor is shown in Listing 5.6. It is parametrised with plan `DoShoppingP` which represents one possible option for the agent to go shopping. The code is borrowed from the product purchase example introduced informally in Section 3.3 and described in more detail as part of the case study in Section 6.2 below.

Listing 5.5: Example applicability check

```
template <>
struct applicable<BuyProductP, int> {
  template <class AS>
  bool operator()(AS& as, int const& i) {
    if(!init) {
      auto& bb = as.getBB();
      bool b = bb.getBelief2(g_sPrice, price);
      assert(b);
      b = bb.getBelief2(g_sHealth, health);
      assert(b);
      init=true;
    }

    // check availability
    std::uniform_real_distribution<> rDist(0,10);
    if(rDist(rng) <= 3)
      return false;

    // check price
    if(price[i] >= 6) {
      std::set<int> *ep, *ep2;
      bool b = as.getBB().getBeliefP(g_sExpensiveProducts, ep);
      if(!b) {
        std::set<int> expProducts;
        expProducts.insert(i);
        as.getBB().addBelief(g_sExpensiveProducts, expProducts);
      }
      else
        ep->insert(i);
      return false;
    }

    // check healthiness
    if(health[i] <= 1) {
      std::set<int>* up;
      bool b = as.getBB().getBeliefP(g_sUnhealthyProducts, up);
      if(!b) {
        std::set<int> unhealthyProducts;
        unhealthyProducts.insert(i);
        as.getBB().addBelief(g_sUnhealthyProducts, unhealthyProducts);
      }
      else
        up->insert(i);
      return false;
    }

    return true;
  }
};
```

Listing 5.6: An example utility functor

```
template <>
struct utility<DoShoppingP, int> {
  template <class AS>
  float operator()(AS const& as, int const& param) {
    return 1.0;
  }
};
```

5.5 Programmer interface

Due to the complexity of the underlying reasoning mechanism, agent-oriented programming can be difficult. In order to facilitate the formulation of AOP applications, it is thus crucial to relieve the programmer from having to deal with the intricacies of the underlying mechanisms. Both Jason and 2APL are nice examples of how a subdivision between a convenient and easy-to-use interface and the complex dynamics of the reasoning process can be realised by extending the actual reasoner with a domain-specific language for the formulation of the agent logic (see Section 2.3 for examples).

The goal of the frontend described in this section is to mimic some of the declarative aspects of AOP languages like Jason or 2APL, yet without compromising on the efficiency of the underlying C++ implementation. In particular, the user should be enabled to formulate beliefs, actions, events, plans, and the agent's internal state (as the union of the previous components) in an easy and intuitive way. Technically, the declarative frontend realises an *embedded domain-specific language (EDSL)*. Starting with the belief base, the framework elements that constitute the EDSL are described in the following sections. Each section starts with an introduction and a schematic description of what a convenient programmatic formulation of the respective components should look like from a user's point of view, followed by the technical realisation.

Environment

As described in Section 5.1.1 above, the environment can be realised as an arbitrary C++ object. In that way, the programmer is given maximum flexibility with respect to the logic. The only requirement is that the environment class has to provide a function `perceive` that the framework uses during the perception stage.

An example implementation of the environment class for the product purchasing scenario is shown in Listing 5.7. It contains as its only function the aforementioned `perceive` function which is parametrised with the agent state class. The logic is straightforward: in the first tick of the simulation, initial percepts about the products are being created and added to the belief base; in order to represent uncertainty and stochastic variation in the agent's perception, randomly chosen products are then added to the 'cheap products' and 'healthy product' data structures, respectively. In each tick, the agent will thus perceive 100 different products as being healthy or 100 different products as being cheap.

The function does itself make use of static polymorphism by assuming the existence of method `getBB` (for accessing the belief base) in the agent state class. In order to ensure that all changes made as part of the perception process are actually reflected in the original belief base, pointers to the original data structures are obtained and used as the basis for modifications

Listing 5.7: Example implementation of the environment class

```
struct Environment {
  template <class AS>
  void perceive(AS& as) {
    auto& beliefs = as.getBB();

    // creating initial beliefs
    if(as.getTick() == 0) {
      std::set<int> sHealthyProducts, sCheapProducts;
      std::set<int> sExpensiveProducts, sUnhealthyProducts;
      std::unordered_set<int> sProducts;
      std::unordered_map<int,float> sPrice;
      std::unordered_map<int,int> sHealth;

      beliefs.addBelief(g_sProducts, sProducts);
      beliefs.addBelief(g_sPrice, sPrice);
      beliefs.addBelief(g_sHealth, sHealth);
      beliefs.addBelief(g_sCheapProducts, sCheapProducts);
      beliefs.addBelief(g_sHealthyProducts, sHealthyProducts);
      beliefs.addBelief(g_sExpensiveProducts, sExpensiveProducts);
      beliefs.addBelief(g_sUnhealthyProducts, sUnhealthyProducts);
    }

    // creating uniform probability distributions
    std::uniform_real_distribution<> rDist(0,10);
    std::uniform_real_distribution<> iDist1(0,1000);
    std::uniform_real_distribution<> iDist2(0,3);

    std::set<int> *sHealthyProducts, *sCheapProducts;
    std::unordered_map<int,float> *sPrice;
    std::unordered_map<int,int> *sHealth;

    // creating references to belief data structures
    bool b = beliefs.getBeliefP(g_sHealthyProducts, sHealthyProducts);
    assert(b);
    b = beliefs.getBeliefP(g_sCheapProducts, sCheapProducts);
    assert(b);
    b = beliefs.getBeliefP(g_sPrice, sPrice);
    assert(b);
    b = beliefs.getBeliefP(g_sHealth, sHealth);
    assert(b);

    // perceive 100 random products as being cheap
    // and 100 random products as being healthy
    for(auto i: irange(0,100)) {
      int prod = iDist1(rng);
      (*sPrice)[prod] = rDist(rng);
      (*sHealth)[prod] = iDist2(rng);
      sCheapProducts->insert(prod);
      sHealthyProducts->insert(prod);
    }
  }
};
```

(using method `getBeliefP` instead of `getBelief`, as described in Section 5.1.2).

Beliefs

As described in Section 5.1.2, it is common in agent-based modelling to support a wide range of different beliefs. In Jason, beliefs can be specified irrespective of their type which is handled automatically through type inference, as shown below.

```
name("fritz").
age(42).
isBoss(true).
```

In order to support an equivalent level of flexibility, we implemented a heterogeneous belief base (see Section 5.1.2). In the first step, the user needs to define the belief base by passing it the data types that it is supposed to handle.

```
BeliefBase<int,string,bool> bb;
```

Individual beliefs can then be created as conventional variables.

```
string belName = "fritz";
int belAge = 42;
bool belBoss = true;
```

Due to the implementation of the belief base as a heterogeneous container, all beliefs can now be added through a common interface, irrespective of their type.

```
bb.addBelief("name", belName);
bb.addBelief("age", belAge);
bb.addBelief("boss", belBoss);
```

Similarly, if beliefs are to be queried, this can be done in a unified and convenient way.

```
bb.getBelief("float belief", &fBelief);
bb.getBelief("string belief", &sBelief);
bb.getBelief("int belief", &iBelief);
```

Goals

In Jason, goals are represented as literals and prepended as required in the specific context (e.g. to describe goal addition or removal). In C++, since goals are to be processed at compile time, they need to be represented as types. Goals do not exhibit any functionality on their own, so there is no need for a concrete goal to inherit from a base class. Goals can thus be represented as simple types, as shown below for the party scenario:

```
struct DoShopping {};
struct Buy {};
```

The agent's goal library can then be described as a simple Boost.MPL typelist.

```
typedef mpl::list<
  DoShopping,
  Buy
> Goals;
```

Actions

In Jason, actions are implemented in plain Java. To that end, the Jason-internal environment classes that need to be subclassed in order to implement a custom environment provide a function `executeAction` which can be used to implement the action logic. An example implementation for a custom action `considerProducts` is shown below.

```
public boolean executeAction(String agName, Structure action) {
    if(action.getFunctor().equalsIgnoreCase("considerProducts")) {
      try {
        /* custom action logic goes here */
        return true;
      }
      catch(Exception e) {
        System.err.println("Error: " + e.getMessage());
      }
    }
    logger.info("executing: "+action+", but not implemented!");
      return true;
}
```

The so-defined action can then be integrated into existing plans by means of its name. In the following Jason plan, action `considerProducts` is used as the first item of the plan body.

```
+!doShopping : true <-
  considerProducts;
  !shop.
```

As described in Section 5.1.4, actions in C++ represent executable entities and are thus implemented as functors (function objects). The framework distinguishes between basic and parametrised actions. Instances of basic actions can be created by simply inheriting from class `PAction` and overwriting the function call operator. Using the example from above, this can be achieved as follows.

```
struct ConsiderProducts : public BAction {
  template <class AS>
  bool operator()(AS& as) const {
    std::set<int> *cp,*hp,*ep,*up;
    std::unordered_set<int> *p,p2;
    as.getBB().getBeliefP(g_sCheapProducts, cp);
    as.getBB().getBeliefP(g_sExpensiveProducts, ep);
    /* additional logic goes here */
    return true;
  }
};
```

As opposed to Jason, the C++ framework further supports the notion of parametrised action. Concrete instances can be created by inheriting from class `PAction` and passing the respective option parameter type, as shown below.

```
struct BuyProduct : public PAction<int> {
  template <class AS>
  bool operator()(AS& as) const {
    cout << "Buy product: " << this->param << endl << flush;
    return true;
  }
};
```

Similar to Jason, actions can be referred to from within plans by means of their name. Plans are described further below but, for the sake of illustration, we anticipate its definition and show the integration of the action defined above into a plan below.

```
typedef Plan<
  DoShoppingP,
  DoShopping,
  mpl::list<
    ConsiderProducts
    AGoal<Shop>
  >
> PDoShoppingP;
```

Events

In Jason, events that may appear in the body of a plan are described by simply prepending the goal with the appropriate event specifier, as shown below.

```
!g    // achievement goal
!!g   // parallel achievement goal
?g    // test goal
```

In C++, as with goals described above, events are represented as types. For performance reasons, test goals are not explicitly supported; if the belief base is to be queried, this is best done within a basic action. The framework supports both goal addition and goal removal. In addition, a distinction between sequential and parallel (i.e. background) execution is made. Events are described as simple types, as shown in the example below.

```
typedef AGoal<BuyProduct> AddBP; // addition of goal 'BuyProduct'
typedef RGoal<BuyProduct> RemBP; // removal of goal 'BuyProduct'
typedef Par<AddBP> ParAddBP;     // parallel addition of goal 'BuyProduct'
typedef Par<RemBP> ParRemBP;     // parallel removal of goal 'BuyProduct'
```

The definition becomes even more intuitive if the new `using` directive is used instead of `typedef`.

```
using AddBP = AGoal<BuyProduct>; // addition of goal 'BuyProduct'
```

```
using RemBP = RGoal<BuyProduct>; // removal of goal 'BuyProduct'
using ParAddBP = Par<AddBP>;     // parallel addition of goal 'BuyProduct'
using ParRemBP = Par<RemBP>;     // parallel removal of goal 'BuyProduct'
```

The resulting types can then be used within plan bodies, as shown in the following paragraphs.

Plans

In Jason, plans are described in a purely declarative way by specifying the triggering event, the context, and the plan body. The plan body itself is simply a sequence of formulae, i.e. actions or subgoals.

```
+!getMoney       // triggering event
 : true <-       // context
  walkToATM;     // basic action
  withdrawCash;  // basic action
  !shop.         // subgoal
```

In C++, given the definition of goals, actions, and events, plans can be formulated in a way that is as declarative as in Jason.

```
typedef Plan<
  GetMoneyFromATM, // plan name
  GetMoney,        // triggering event
  list<            // plan body
    WalkToATM,     // basic action
    WithdrawCash   // basic action
    AGoal<Shop>    // subgoal
  >
> PLGetMoneyFromBank;
```

The agent's plan library can then be described as a simple Boost.MPL typelist.

```
 typedef mpl::list<
   PDoShoppingP,
   PBuyProductP
 > PlanLibrary;
```

Agent state

Finally, the overall agent state needs to be described. In Jason, this happens implicitly by means of the content of the *asl* file, i.e. the file that contains the agent logic. In the C++ framework, the agent state needs to be described explicitly. As with the definition of plans, the framework allows for a purely declarative description, as shown below.

```
 typedef State<
   Beliefs,              // belief base
     PlanLibrary,        // plan library
     Goals,              // all goals that the agent can handle
```

```
    list<                 // list of initial goals
     AGoal<DoShopping>
    >,
    RTApplicability       // applicability checking mechanism
  UtilityEvalSelector     // plan selection mechanism
> AgentState;
```

5.6 Summary

In the previous sections, a comprehensive description of the C++ framework was given. Starting with the static components, we continued by describing the implementation of the central steps of the reasoning cycle. An important design principle underlying the implementation is the combination of compile time and runtime computations in order to achieve both good performance and convenience of use. For that reason, three kinds of operations are used: pure compile time, hybrid compile time/runtime, and pure runtime.

For the purpose of plan selection, a number of concrete mechanisms tailored to agent-based modelling were implemented and described. The framework provides a number of customisation mechanisms which allow a programmer to adapt the functionality to its specific requirements. Finally, the framework was designed with a particular focus on usability. In particular, the central, user-facing framework components were designed in such a way that they form an embedded domain-specific language that resembles the DSL integrated into AOP languages such as Jason or 2APL. In that way, the performance of C++ can be combined with the convenience of a higher-level, declarative programming interface.

The previous section contained a number of simple examples of how the framework can be used. The next chapter contains the description of more complex case studies. Besides usability aspects, the experiments will also emphasise performance implications of different approaches being used (e.g. compile time vs runtime plan selection).

6 Case studies

In this chapter, we return to the two example scenarios introduced informally in Chapter 3. We provide examples of how the scenarios can be implemented using the C++ framework. Furthermore, we evaluate their performance in terms of both compile time and runtime consumption.

6.1 Scenario 1: Party scenario

For the first case study, we return to the party scenario introduced in Section 3.2. We keep the example deliberately simple and restrict the focus to basic, i.e. unparametrised actions. In this example, the agent follows a fairly simple behavioural rule which is shown again in Figure 6.1. The agent has one major goal, namely to buy a present. In order to achieve this goal, three tasks need to be accomplished. First, the agent needs to get money. This is realised as a subgoal for which two options are available: (i) getting money from the ATM and (ii) getting money from a friend. Both cases require further substeps. Second, the agent needs to walk to the shop. And, third, the present needs to be purchased. The last two tasks are realised as simple actions.

Implementation

In this simple example, the environment does not require any internal logic, so we can leave the perception function empty.

```
struct Environment {
  template <class AS>
  void perceive(AS const& as) {
  }
};
```

We further assume that the agent is 'stateless', i.e. that its knowledge base is empty. However, the agent state class requires a type parameter as well as an instance of the belief base upon construction, so we simply define an empty belief base as follows.

```
typedef BeliefBase<int> Beliefs;
Beliefs beliefs;
```

In order to implement the agent's behavioural logic, we first need to define the goals as basic types. This can be done by simply providing appropriate structs, as shown below.

```
struct GetPresent {};
struct GetMoney {};
```

We continue with the individual actions. All actions in the example are basic actions; we thus need to define classes (or structs) that inherit from class `BAction` and provide a function

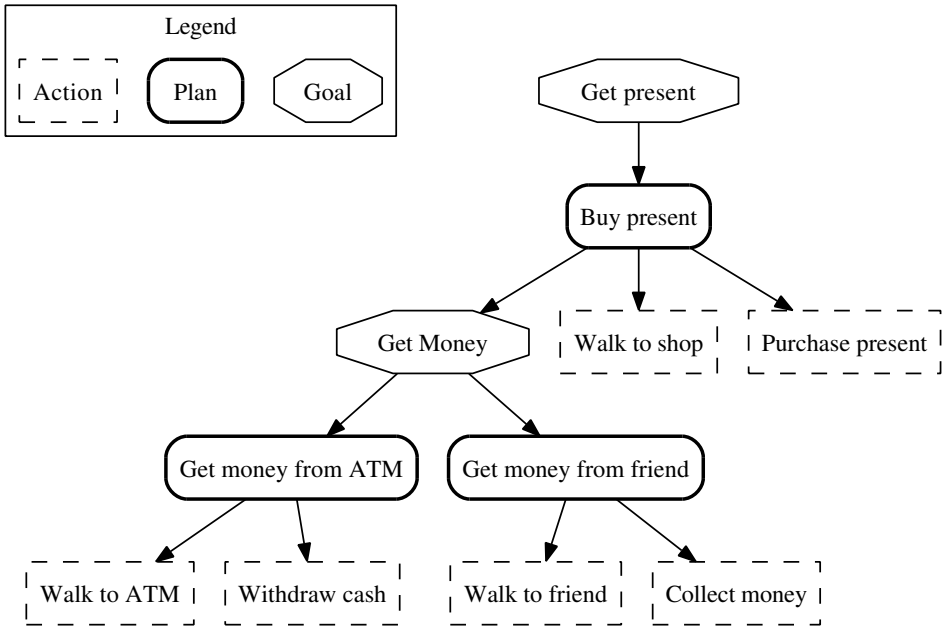


Figure 6.1: Goal-plan tree for a party scenario (adapted from [30])

call operator that (i) is parametrised with the agent state and the environment and (ii) accepts a reference to the agent state as an argument. We exemplify the implementation using the 'Walk to ATM' action. For simplicity, we restrict the internal logic of the actions to a simple output to the console.

```
struct WalkToATM : public BAction {
  template <class AS>
  bool operator()(AS& as) const {
    cout << "Walk to ATM" << endl << flush;
    return true;
  }
};
```

All other actions can be implemented accordingly. In the case study, all actions have the same utility of 1.0. Furthermore, all actions are by default applicable. The description of the utility and applicability structs is thus omitted here.

We continue with the definition of the necessary plans. There are two goals in the example: *GetPresent* and *GetMoney*. For the first goal, the agent has one plan; it is shown below.

```
typedef Plan<
  BuyPresent,
  GetPresent,
  mpl::list<
    Seq<BGoal<GetMoney>>,
    WalkToShop,
    PurchasePresent
  >
> PBuyPresent;
```

The plan definition tells us that (i) *BuyPresent* is the name of the plan, (ii) *GetPresent* is the goal to be handled, and (iii) the plan body contains three elements: a basic subgoal *GetMoney* which is to be handled sequentially, and two basic actions, *WalkToShop* and *PurchasePresent*.

For the second goal, *GetMoney*, there are two plan options that the agent can select from. The first plan is responsible for getting money from the bank. It is shown below.

```
typedef Plan<
  GetMoneyFromATM,
  GetMoney,
  mpl::list<
    WalkToATM,
    WithdrawCash
  >
> PGetMoneyFromATM;
```

The plan body contains two basic actions: *WalkToATM* and *WithdrawCash*. Alternatively, the agent can get money from his friend. This is realised by the plan shown below.

```
typedef Plan<
  GetMoneyFromFriend,
  GetMoney,
  mpl::list<
    WalkToFriend,
    CollectMoney
  >
> PGetMoneyFromFriend;
```

Similar to the previous plan, this plan contains two basic actions: *WalkToFriend* and *CollectMoney*.

Given the definition of goals, actions, and plans, we can now define the agent's plan library as a simple Boost.MPL typelist.

```
typedef mpl::list<
  PBuyPresent,
  PGetMoneyFromATM,
  PGetMoneyFromFriend,
> PlanLibrary;
```

For the second goal *GetMoney*, the agent has two plan options to choose from; we thus need to think about the plan selection mechanism next. Plan relevance is determined based on triggering goals, so there is nothing else to do.

For applicability checking, we need to distinguish between compile time and runtime checking. In order to be able to compare compile time and runtime checking with respect to performance, we need to provide the basic infrastructure for both mechanisms. For compile time checking, no further logic is required; for runtime checking, we need to provide appropriate functors. For simplicity, we assume that all plans are applicable. The respective functors can thus be implemented as exemplified below.

```
template <>
struct applicable<GetMoneyFromFriend> {
```

```
  template <class AS>
  bool operator()(AS const& as) {
    return true;
  }
};
```

The final step in the plan selection process concerns the selection from a set of applicable plans. For compile time selection, no additional information is required; for runtime selection (using a utility mechanism), we need to assign appropriate utility values to the individual plans. For simplicity, we give each plan a utility of 1.0 which results in alternative plans having equal probability of being selected. As described in Section 5.3 above, this requires the implementation of a utility functor which is exemplified for plan *GetMoneyFromATM* below[1]. All other utility calculations are implemented accordingly.

```
template <>
struct utility<GetMoneyFromATM> {
  template <class AS>
  float operator()(AS const& as) {
    return 1.0;
  }
};
```

Given the previous definitions as well as two chosen mechanisms (for applicability checking and plan selection), the agent's state can now be defined as follows.

```
AgentState<
  Beliefs,
  PlanLibrary,
  Goals,
  mpl::list<
    Add<BGoal<GetPresent>>,
  >,
  RTAllApplicable,
  MaxUtilitySelector
> state(beliefs);
```

Evaluation

For the first experiment, we focus on the impact of compile time and runtime computation on the performance of the resulting application. We compare the following three scenarios:

1. Pure compile time (compile time applicability checking + compile time plan selection)
2. Hybrid (compile time applicability checking + runtime plan selection)
3. Pure runtime (runtime applicability checking + runtime plan selection)

[1] In a more realistic example, the function call operator would, of course, contain a more meaningful utility calculation, probably utilising the agent's current knowledge. In order not to complicate the performance evaluation, this has been kept deliberately simple.

#agents	real	user	system
10	6.073	5.253	0.304
100	5.601	5.287	0.297
1000	5.803	5.498	0.282
10000	5.532	5.226	0.293

#agents	real	user	system
10	0.0	0.0	0.0
100	0.002	0.0	0.0
1000	0.011	0.01	0.0
10000	0.09	0.088	0.0

Table 6.1: Influence of population size on compilation time (left) and execution time (right) with **compile time applicability checking** and **compile time plan selection**

For each scenario, we run the simulation for 1000 ticks and with 10, 100, 1000, and 10000 agents, respectively. All experiments were performed on a Dell Latitude E6420 with 2 Intel® Core™ i7-2620M CPUs (2.7 GHz each), 8 GB of memory, and Linux Mint 17.1 Cinnamon 64-bit (kernel version 3.13.0-37-generic) as operating system. For profiling, the Linux tool `time` was used. It reports three different aspects of the runtime of an application: (i) the elapsed real time between invocation and termination ('real'), (ii) the user CPU time ('user'), and (iii) the system CPU time ('system'). The resulting numbers are averages of 10 independent runs.

The results for Scenario 1 are shown in Table 6.1. We can see that the compilation is, of course, independent of the number of agents in the population. Furthermore, due to the heavy focus on runtime computation in this example, the execution time is close to 0 for all population sizes.

#agents	real	user	system
10	5.969	5.657	0.292
100	5.99	5.679	0.292
1000	5.974	5.658	0.3
10000	6.026	5.697	0.31

#agents	real	user	system
10	0.041	0.039	0.0
100	0.381	0.379	0.0
1000	3.669	3.663	0.0
10000	36.671	36.646	0.0

Table 6.2: Influence of population size on compilation time (left) and execution time (right) with **compile time applicability checking** and **runtime time plan selection**

The results for Scenario 2 are shown in Table 6.2. Here, the compilation time is slightly higher which is due to the more complex class structure in the case of runtime plan selection. We can also see that the execution time is significantly higher now. As expected, it scales linearly with the population size.

#agents	real	user	system
10	6.131	5.797	0.317
100	6.161	5.829	0.312
1000	6.177	5.861	0.3
10000	6.197	5.865	0.311

#agents	real	user	system
10	0.062	0.061	0.001
100	0.57	0.567	0.0
1000	5.532	5.523	0.001
10000	55.056	54.98	0.033

Table 6.3: Influence of population size on compilation time (left) and execution time (right) with **runtime applicability checking** and **runtime time plan selection**

The results for Scenario 3 are shown in Table 6.3. Compilation is slightly more time-consuming than in the previous case because of the additional classes required for runtime applicability checking. Also, as expected, the execution time is slightly higher than before.

#agents	Scenario	real	user	system
10	Baseline	0.353	0.287	0.06
	BDI	6.131	5.797	0.317
	Factor	*17.37*	*20.2*	*5.3*
100	Baseline	0.346	0.273	0.065
	BDI	6.161	5.829	0.312
	Factor	*17.8*	*21.4*	*4.8*
1000	Baseline	0.353	0.282	0.065
	BDI	6.177	5.861	0.3
	Factor	*17.5*	*20.8*	*4.6*
10000	Baseline	0.353	0.282	0.065
	BDI	6.197	5.865	0.311
	Factor	*17.6*	*20.8*	*4.8*

#agents	Scenario	real	user	system
10	Baseline	0.0	0.0	0.0
	BDI	0.062	0.061	0.001
	Factor	*n/a*	*n/a*	*n/a*
100	Baseline	0.018	0.018	0.0
	BDI	0.57	0.567	0.0
	Factor	*31.7*	*31.5*	*n/a*
1000	Baseline	0.132	0.13	0.0
	BDI	5.532	5.523	0.001
	Factor	*41.9*	*41.8*	*n/a*
10000	Baseline	1.311	1.305	0.003
	BDI	55.056	54.98	0.033
	Factor	*42*	*42.1*	*11*

Table 6.4: Impact of BDI framework usage on compilation time (left) and runtime (right) for different population sizes, with runtime applicability checking and runtime time plan selection

The second experiment concerns the general compilation time and runtime overhead incurred by the framework. For that purpose, we developed a baseline scenario which contains the same functionality with respect to the agent's behaviour, yet without using the BDI framework. The code of the baseline scenario can be found in Appendix D. The results of the experiment are shown in Table 6.4. It is apparent that usage of the framework increases compilation time by a constant factor of $\approx$ 17–18 and execution time by a constant factor of $\approx$ 30–40. Especially in the latter case, there is certainly room for improvement. We are planning to investigate further optimisations as part of our future work.

	Jason (1 core)			Jason (4 cores)			C++ framework (1 core)		
#agents	real	user	system	real	user	system	real	user	system
10	6.642	1.72	0.12	6.138	1.998	0.102	0.067	0.065	0.0
100	11.516	5.91	0.668	7.72	5.324	0.908	0.568	0.565	0.0
1000	26.59	13.7	7.62	23.05	27.1	12.59	5.55	5.54	0.001
10000	315.3	131.5	177.9	249.9	240.7	239.5	55.056	54.98	0.033

Table 6.5: Execution time of the Jason implementation in comparison with that of the C++ framework for different population sizes, with runtime applicability checking and runtime plan selection in the latter case

The final experiment compares the runtime performance of the C++ framework with that of Jason. For that purpose, we have implemented the party scenario using the declarative agent language of Jason, as shown in Appendix E. In order to keep the evaluation focussed on the core functionality of the frameworks, the basic actions (`walkToShop`, `walkToATM`, etc.) have been left completely empty. For the evaluation, we ran the Jason implementation in 'headless' mode, i.e. without graphical interface. Since Jason is multi-threaded, we ran experiments both on a single and on four cores in order to show the impact of parallelism. The results for different numbers of agents are shown in Table 6.5. The numbers indicate that Jason scales particularly well in the presence of parallelism; it also becomes apparent that operations in kernel mode consume an increasingly significant portion of time as the population grows. This could be either due to garbage collection or to the management of threads on the operating

system level, although these are just speculations that would have to be investigated by means of more complex profiling experiments. The C++ framework scales linearly with the number of agents and is, in terms of total time, significantly faster than Jason.

6.2 Scenario 2: Selecting a product

It is often necessary in agent-based modelling to apply a common behavioural pattern to a wide range of options. For example, in order to be able to perform a selection from a (possibly large) range of available products in a shop, the agent may need to apply an evaluation function to each of the products prior to the actual selection operation. Both product evaluation and product selection can be seen as generic operations parametrised by the respective product. Since those operations correspond with actions in the BDI framework and actions form part of larger plans, it is natural to consider the implementation of generic actions as plans, as described in Section 3.3. In our second case study, we focus on the evaluation of those parametrised plans and actions.

For the evaluation, we return to the product purchasing scenario introduced in Section 6.2. Here, the agent is supposed to pick a product from a large range of options. The selection of a product is split up into two successive steps. In the first step, a shortlist of products is created. The products on the shortlist can be seen as those options that satisfy the agent's basic criteria and are thus going to be considered seriously for purchase. The agent bases its shortlisting process upon two criteria: price and healthiness. In the second step, the agent performs a comparison of the shortlisted products by means of a utility evaluation. Similar to the previous case study, we start with a description of the model implementation, followed by a performance evaluation.

Implementation

We start with the environment. As opposed to the previous case study, the environment has an important purpose here: it is responsible for providing the agent with a set of percepts about the available products, about their price as well as about their healthiness. For simplicity, the perception process is uniformly randomised, i.e. in each tick, each agent is presented with a random selection of products. The code of the environment class is shown below.

```
struct Environment {
  template <class AS>
  void perceive(AS& as) {
    auto& beliefs = as.getBB();

    // in the first tick, present the agent with basic information
    // about the products and their attributes
    if(as.getTick() == 0) {
      std::set<int> sHealthyProducts,
              sCheapProducts,
              sExpensiveProducts,
              sUnhealthyProducts;
      std::unordered_set<int> sProducts;
      std::unordered_map<int,float> sPrice;
      std::unordered_map<int,int> sHealth;

      beliefs.addBelief(g_sProducts, sProducts);
```

```
      beliefs.addBelief(g_sPrice, sPrice);
      beliefs.addBelief(g_sHealth, sHealth);
      beliefs.addBelief(g_sCheapProducts, sCheapProducts);
      beliefs.addBelief(g_sHealthyProducts, sHealthyProducts);
      beliefs.addBelief(g_sExpensiveProducts, sExpensiveProducts);
      beliefs.addBelief(g_sUnhealthyProducts, sUnhealthyProducts);
    }

    std::uniform_real_distribution<> rDist(0,10);
    std::uniform_real_distribution<> iDist1(0,1000);
    std::uniform_real_distribution<> iDist2(0,3);

    // data structures for the storage of percepts
    std::set<int> *sHealthyProducts, *sCheapProducts;
    std::unordered_map<int,float> *sPrice;
    std::unordered_map<int,int> *sHealth;

    // determine information about healthy and cheap products,
    // as well as about price and healthiness of all products
    bool b = beliefs.getBeliefP(g_sHealthyProducts, sHealthyProducts);
    assert(b);
    b = beliefs.getBeliefP(g_sCheapProducts, sCheapProducts);
    assert(b);
    b = beliefs.getBeliefP(g_sPrice, sPrice);
    assert(b);
    b = beliefs.getBeliefP(g_sHealth, sHealth);
    assert(b);

    // create random percepts
    for(auto i: irange(0,100)) {
      int prod = iDist1(rng);
      (*sPrice)[prod] = rDist(rng);
      (*sHealth)[prod] = iDist2(rng);
      sCheapProducts->insert(prod);
      sHealthyProducts->insert(prod);
    }
  }
};
```

Products are represented as simple integer values. The following data structures are further used to store information about the products:

- `std::unordered_set<int>` for representing the products.
- `std::set<int>` for representing the shortlists, i.e. certain attribute-based subsets of products (e.g. healthy products, cheap products, etc.).
- `std::unordered_map<int, T>` for representing the products' attribute of type `T`.

Knowing the types of data structures necessary for storing the product information, we can now parametrise the agent's belief base as follows.

```
typedef BeliefBase<
  std::unordered_set<int>,
  std::set<int>,
  std::unordered_map<int,float>,
  std::unordered_map<int,int>
> Beliefs;
```

In order to implement the agent's behavioural logic, we first need to define the goals as basic types. For the product purchasing scenario, we only need the following two goals as well as the subsequent definition of the goal library.

```
struct DoShopping;
struct Buy;

typedef mpl::list<
  DoShopping,
  Buy
> Goals;
```

We continue with the individual actions. Inspecting the behaviour tree in Figure 3.2, we need two actions: `ConsiderProducts` (nonparametrised) and `Buy` (parametrised). Intuitively, `ConsiderProducts` contains the shortlisting process. Here, the agent uses its knowledge about the products and their attributes in order to classify them according to the criteria that it considers relevant (in our case price and healthiness).

```
struct ConsiderProducts : public BAction {
  template <class AS>
  bool operator()(AS& as) const {
    std::set<int> *cp,*hp,*ep,*up;
    std::unordered_set<int> *p,p2;
    // determine products labelled as cheap
    bool b = as.getBB().getBeliefP(g_sCheapProducts, cp);
    assert(b);
    // determine products labelled as expensive
    b = as.getBB().getBeliefP(g_sExpensiveProducts, ep);
    assert(b);
    // determine products considered _really_ cheap by the agent
    std::set<int> rcp;
    std::set_difference(
      cp->begin(),
      cp->end(),
      ep->begin(),
      ep->end(),
      std::inserter(rcp, rcp.begin())
    );
    // determine products labelled as healthy
    b = as.getBB().getBeliefP(g_sHealthyProducts, hp);
    assert(b);
    // determine products labelled as unhealthy
    b = as.getBB().getBeliefP(g_sUnhealthyProducts, up);
```

```
    assert(b);
    // determine products considered _really_ healthy by the agent
    std::set<int> rhp;
    std::set_difference(
      hp->begin(),
      hp->end(),
      up->begin(),
      up->end(),
      std::inserter(rhp, rhp.begin())
    );
    // store all products considered both really healthy
    // and really cheap by the agent
    b = as.getBB().getBeliefP(g_sProducts, p);
    p->clear();
    assert(b);
    std::set_intersection(
      rcp.begin(),
      rcp.end(),
      rhp.begin(),
      rhp.end(),
      std::inserter(*p, p->begin())
    );
    return true;
  }
};
```

The second basic action, Buy, is trivially simple. All it does is to output the product that has been chosen for purchase.

```
struct BuyProduct : public PAction<int> {
  template <class AS>
  bool operator()(AS& as) const {
    cout << "Buy product: " << this->param << endl << flush;
    return true;
  }
};
```

We continue with the definition of the necessary plans. There are two goals in the example: DoShopping and Buy. For the first goal, the agent has one plan; it is shown below.

```
typedef Plan<
 DoShoppingP,
 DoShopping,
 mpl::list<
   ConsiderProducts,
   Seq<
     PGoal<
       Buy,
       g_sProducts,
       int
     >
```

```
    >,
    Par<
      BGoal<DoShopping>
    >
  >
> PDoShoppingP;
```

The body of the plan consists of three items: (i) a basic action `ConsiderProducts` (described above), (ii) a sequential addition of parametrised goal `Buy` (parametrised with the product index of type `int`, all of which are stored in a data structure whose name is represented by `g_sProducts`), and (iii) the recursive parallel addition of goal `DoShopping` in order to realise an infinite purchasing loop. Goal `Buy` is parametrised with belief `g_sProducts`, where `g_sProducts` represents the name of a container with elements of type `int`.

The plan for handling goal `Buy` is very simple. It contains as its only item the basic action `BuyProduct`.

```
typedef Plan<
 BuyProductP,
 Buy,
 mpl::list<
   BuyProduct
 >
> PBuyProductP;
```

As shown above, goal `Buy` which plan `BuyProductP` is supposed to handle was a parametrised goal. At runtime, plan `BuyProductP` will thus be instantiated once for each product in the product list. Most of the selection logic happens within the applicability evaluation of the plan. This can be understood as follows. Each product option is represented as a separate plan that — if performed successfully — satisfies the agent's `Buy` goal. As described further above, in order to avoid exhaustive utility evaluation for all products, the agent first creates a shortlist of products worth considering for purchase. This shortlist creation process is realised by means of applicability evaluation: each product option represented by a plan is considered applicable if and only if it satisfies the basic price and healthiness criteria of the agent.

For clarity, we break up the description of the applicability checking mechanism into small bits. The first part is only executed once, in the beginning of the simulation. Here, the price and health information for each product is obtained and stored in temporary containers `price` and `health`.

```
if(!init) {
  auto& bb = as.getBB();
  bool b = bb.getBelief2(g_sPrice, price);
  assert(b);
  b = bb.getBelief2(g_sHealth, health);
  assert(b);
  init=true;
}
```

In the next step, the availability of the product under consideration is determined. For simplicity, we assume that each product has an availability of 30%. If the product turns out to

be unavailable, the applicability checking function is left unsuccessfully.

```
std::uniform_real_distribution<> rDist(0,10);
if(rDist(rng) <= 3)
  return false;
```

The next step concerns the evaluation of the product price. If the product has a price higher than 6, then it is considered expensive. The agent memorises this newly gained information by adding the product to its list of expensive products and decides that the product is not applicable.

```
if(price[i] >= 6) {
  std::set<int> *ep, *ep2;
  bool b = as.getBB().getBeliefP(g_sExpensiveProducts, ep);
  if(!b) {
    std::set<int> expProducts;
    expProducts.insert(i);
    as.getBB().addBelief(g_sExpensiveProducts, expProducts);
  }
  else
    ep->insert(i);
  return false;
}
```

The same mechanism is applied to the healthiness of the product. If the product has a health value less than or equal to 1, then it is considered unhealthy. The agent memorises each unhealthy product by adding it to the list of unhealthy products and decides that the product is not applicable.

```
if(health[i] <= 1) {
  std::set<int>* up;
  bool b = as.getBB().getBeliefP(g_sUnhealthyProducts, up);
  if(!b) {
    std::set<int> unhealthyProducts;
    unhealthyProducts.insert(i);
    as.getBB().addBelief(g_sUnhealthyProducts, unhealthyProducts);
  }
  else
    up->insert(i);
  return false;
}
```

If the product passes all of the aforementioned checks (i.e. if the product is cheap enough and healthy enough), then then plan to buy the product is considered applicable and the applicability evaluation returns `true`. Conceptually, the product has now been added to the shortlist.

Given the definition of goals, actions, and plans, we can now define the agent's plan library as a simple Boost.MPL typelist.

```
typedef mpl::list<
  PDoShoppingP,
  PBuyProductP
```

```
> PlanLibrary;
```

Due to the dependence upon runtime information, runtime applicability checking mechanism `RTApplicability` needs to be chosen. Furthermore, since products on the shortlist are to be compared with respect to their utility, the `UtilityEvalSelector` is a good choice for plan selection. Given the previous definitions as well as the two chosen mechanisms for applicability checking and plan selection, the agent's state can now be defined as follows (the belief base is initially empty, so there is no need to pass it to the constructor of the agent state).

```
AgentState<
  Beliefs,
  PlanLibrary,
  Goals,
  mpl::list<
    Add<BGoal<DoShopping>>
  >,
  RTApplicability,
  UtilityEvalSelector
> state;
```

This concludes the description of the implementation. In the next paragraphs, we describe a set of experiments with the overall goal of evaluating compile time and runtime performance.

Evaluation

Similar to the previous case study, we present the results from the performance evaluation in the following paragraphs.

#agents	real	user	system
10	5.47	5.184	0.273
100	5.482	5.191	0.276
1000	5.501	5.188	0.289
10000	5.593	5.227	0.343

#agents	real	user	system
10	0.175	0.173	0.0
100	1.701	1.698	0.0
1000	17.111	17.095	0.001
10000	170.033	169.913	0.003

Table 6.6: Compilation time (left) and execution time (right) for 1000 products, 100 percepts, 100 ticks, and a varying number of agents

In the first experiment, we investigate the performance of the framework with respect to the population size. In order to keep the experiment manageable, the number of simulated ticks is reduced to 100. The results are shown in Table 6.6. As expected, the compilation time is, again, independent of the size of the population. The runtime increases linearly with the number of agents which is also intuitively correct.

In the second experiment, we investigate the efficiency of the framework with respect to a growing number of products to be selected from. To that end, we first increase the number of available products, while leaving the number of percepts fixed (100). The numbers indicate that compilation time remains stable and execution time increases linearly with the number of products to be chosen from. For this and the following experiments, the number of simulated ticks is increased to 1,000.

#products	real	user	system
100	5.577	5.197	0.36
1000	5.643	5.303	0.32
10000	5.623	5.223	0.387

#products	real	user	system
100	0.357	0.353	0.0
1000	1.79	1.78	0.003
10000	17.487	17.45	0.017

Table 6.7: Compilation time (left) and execution time (right) for 10 agents, 1000 ticks, a varying number of products, and a fixed number of percepts (100)

#percepts	real	user	system
100	5.62	5.267	0.343
1000	5.873	5.453	0.4
10000	5.59	5.203	0.363

#percepts	real	user	system
100	16.94	16.767	0.153
1000	24.763	24.053	0.677
10000	63.077	62.61	0.38

Table 6.8: Compilation time (left) and execution time (right) for 10 agents, 1000 ticks, a fixed number of products (10,000) and a varying number of percepts

We now reverse the experiment and vary the number of percepts while keeping the number of products fixed at a level of 10,000. The results are shown in Table 6.8. There is a much stronger increase in runtime now, yet it is still sublinear in the number of products.

#prod & perc.	real	user	system
100	5.86	5.39	0.443
1000	5.683	5.31	0.343
10000	5.687	5.337	0.323

#prod & perc.	real	user	system
100	0.387	0.373	0.013
1000	4.383	4.373	0.003
10000	62.703	61.97	0.663

Table 6.9: Compilation time (left) and execution time (right) for 10 agents, 1000 ticks, a varying number of products, and a varying number of percepts

In the final experiment, we vary both the number of products and the number of percepts *at the same time*. The numbers shown in Table 6.9 indicate a slightly superlinear increase in runtime. This is most likely due to the combination of both products **and** percepts being varied. In fact, making the number of products equal to the number of percepts forces the agent to check each product for applicability. In that case, the number of parametrised plans being executed is increased which, in turn, also increases the number of products that a utility evaluation is to be performed over.

7 Reflection

This chapter is dedicated to a reflection on the work described in the previous chapters. We start in Section 7.1 by revisiting the technical challenges posed in Section 3.4.2 further above and describing our experience in coming up with a solution. This is then followed in Section 7.2 by a more general reflection on the use of template metaprogramming in the context of the developed framework. The chapter concludes with a brief summary.

7.1 Technical challenges revisited

In Section 3.4.2, we described the following four major technical challenges:

1. Support for heterogeneous data storage
2. Declarative interface
3. Combination of declarative and procedural logic
4. Support for both compile time and runtime plan selection

In the context of the framework, we have developed solutions for each of the challenges. They are briefly revisited in the following paragraphs.

Support for heterogeneous data storage

As a matter of convenience, it is import for a framework user not to be unreasonably constrained with respect to the data types he can use to store simulation-specific data. This concerns in particular the management of beliefs, but also the management of events and intentions. As mentioned in various places above, C++ is a statically typed language which does not allow for heterogeneous containers. If values of different types have to be stored in a container, they need to be wrapped explicitly into a container type such as Boost.Any which, however, impacts the runtime performance by requiring frequent type conversions. In order to allow for heterogeneous storage, we followed two different approaches. The first approach — which we used for the implementation of the belief base — is a technique called policy-based class design. Here, generic programming is used in combination with inheritance to mimic the behaviour of heterogeneous containers. The advantage of this technique is that the complexity of the actual composition mechanism is completely hidden from the user, to whom the belief base appears as one single component that provides a unified interface for adding and accessing beliefs of different type. The second technique that we used to realise heterogeneous storage of events and intentions was Boost.Variant. As opposed to Boost.Any which allows for storing and accessing arbitrarily typed values at runtime, type conversions necessary for manipulating instances of Boost.Variant happen at compile time. Manipulation is thus cheap. The disadvantage is that Boost.Variants cannot be accessed directly; they have to be accessed through appropriate *visitors* which have the form of simple structs (or classes) that provide a parametrised function call operator.

Declarative interface

The second important technical challenge was the development of a declarative interface in order to facilitate the accessibility of the framework. We aimed to follow the philosophy of Jason which provides a declarative interface for the formulation of the agent logic (beliefs, goals, plans), but allows for the formulation of the environment logic in a high-level general purpose programming language (Java).

Our initial goal was to develop an external domain-specific language (DSL) on top of the developed framework. However, having worked with TMP, we realised that providing an *embedded* DSL would be even more advantageous: first, due to the capability of TMP to produce embedded DSLs almost 'for free', this seemed like the obvious solution; second, due to its seamless integration into the high-level host language, an embedded DSL offers more flexibility than an external one since it allows for arbitrarily mixing DSL and non-DSL code.

By designing the types, the metafunctions, and the metafunction classes carefully enough, an embedded DSL results almost for free. The subset of types that the user needs to specify in order to describe his application logic forms the syntax of the DSL. In our example, the primary language elements are actions, goals, events, plans, and the agent state. The desired agent logic can be described conveniently by plugging together and composing types in the appropriate way. It would certainly be beneficial to further improve the convenience of the interface, e.g. by hiding the still required references to Boost.MPL, or by replacing the *typedef* notation with an even more abstract symbolic interface. We aim to address this point as part of our future work.

Combination of declarative and procedural logic

Whereas a DSL provides a high level of convenience with respect to the respective domain under consideration, it also constrains the developer's freedom by providing him with a well-defined set of language elements that he is allowed to use. Everything that is beyond the expressiveness of the DSL cannot be expressed by the developer. A high-level general purpose language like C++ or Java, on the other hand, provides a high level of freedom with respect to the formulation of application logic, yet it is, in general, inferior to a tailored DSL with respect to succinctness and elegance. It is obvious that neither approach is superior to the other and different problems require different solutions. This is not just true across but also even within projects. Whereas the agent logic is clearly defined and may thus be best described using a DSL, the logic of the environment may be entirely arbitrary and should thus be formulable in a generic, non-constrained way. As described above, embedded DSLs provide the advantage of combining domain-specific and general code. In the BDI framework, it is thus perfectly possible to mix and match both paradigms flexibly. In that way, the agent-specific bits can be specified in a declarative, minimalistic way (e.g. the plans), whereas the developer is free to formulate the logic of the environment, of basic actions, or of utility and applicability mechanisms using conventional C++ ad libitum.

Support for both compile time and runtime plan selection

The final challenge concerns the selection of plans which — depending on the criterion used — may be performed at compile time or at runtime. Especially at runtime, a plethora of concrete selection mechanisms are conceivable, some of which have been described in Section 5.3 above. Rather than anticipating and hardwiring all possible mechanisms in the framework, we tried to provide a generic structure that deals with the problem of compile time and runtime plan selection in an abstract way and allows the user to plug in his own mechanisms. This is

achieved by a mixture of compile time, hybrid, and runtime computations using Boost.MPL and Boost.Fusion as described in Sections 5.2.2, 5.3, and 5.4. The structure allows for mixing and matching different concrete selection mechanisms including applicability checking and actual selection, yet without losing track of important constraints. For example, a compile time computation may be succeeded by either a compile time or a runtime computation, whereas a runtime computation may only be succeeded by another runtime computation. As described in detail in Sections 5.2.2 and 5.3, this has been accounted for explicitly.

7.2 General reflection on C++ TMP

Following the solution of concrete technical challenges described in the previous section, we now reflect on the more general experience with C++ TMP in the context of the dissertation project. As a start, we emphasise three of the main benefits of using generic programming techniques and, in particular, template metaprogramming. First, one of the main problems of most software code bases is that they are cluttered with repetitive boilerplate code. In most cases, this is a natural consequence of the evolutionary nature of the software development project: functionality is added or removed, developers with different levels of expertise leave and join the team, requirements change, etc. Implications on the overall software design are often neglected when new functionality is added — especially in the presence of resource constraints. Cut-and-paste as a means of replicating existing functionality is common and likely to lead to software erosion.

Generic programming represents just one of many techniques to deal with this problem. By facilitating the implementation of generic data structures and algorithms, i.e. those that separate functionality from the types that are to be operated upon, developers are enabled to focus on the *reusability* of their code. The idea of writing generic, type-independent code — also referred to as static or parametric polymorphism — encourages developers to embed some of the semantics of the application into the type system. By coming up with a meaningful set of types on the one hand, and data structures and algorithms that are to operate upon those types on the other hand, the developer realises an important separation of concerns with respect to data and functionality which can help to reduce the amount of boilerplate control logic in the code significantly. One could refer to this paradigm as *type-oriented programming*.

Template metaprogramming (TMP) goes even further. Using generic programming techniques, it allows for the development of complex *computations* that are themselves executed at compile time. TMP has been shown to be extremely powerful for the design and implementation of both custom applications and, in particular, libraries [2, 1]. Revisiting the idea of type-oriented programming mentioned above, TMP can be seen as an enhanced version thereof. Whereas conventional generic programming aims to abstract away some of the type-specific details from the underlying code, TMP allows for the abstraction of entire type-related computations.

For the implementation of the BDI framework, this view proved very helpful. By separating those aspects of the framework that the user should have explicit control of (plans, goals, actions, etc.) from those that are part of the internal ‘mechanics’, usability could be enhanced significantly. Hiding irrelevant details from the user helps him to focus on the essentials; doing so in a type-oriented way using TMP further provides the additional benefit of enhancing the overall performance of the application.

In the following paragraphs, we focus on three main criticisms regarding TMP and reflect on our experiences.

Argument 1: C++ and, in particular, TMP is not a good choice for a framework

Compared to widespread programming paradigms such as object-orientation and associated mainstream languages such as Java, TMP in combination with C++ certainly seems like a slightly esoteric choice for the implementation of a framework. It is true that the number of programmers with experience or even proficiency in C++ and, even more so, in TMP is certainly significantly lower than that of Java programmers. This, in turn, can be expected to reduce the number of developers willing to use the developed framework as part of their code. Nevertheless, we believe that C++ in combination with TMP is the right choice of technology for our problem, for reasons laid out below.

First, in the realm of high-level general purpose programming languages, C++ still produces the most efficient machine code. Despite impressive advances in the context of Just-in-Time (JIT) compilation, binaries produced by modern C++ compilers are still several steps ahead. Even though the difference may not be relevant for a general purpose word processing application, it certainly is for applications that involve a fair amount of numeric computation. Furthermore, the compilation stage in the case of C++ creates a binary file that can be executed over and over again, without the need of recompilation (unless the code changes, of course). This is particularly attractive for the implementation of stochastic simulation models that need to be executed a large number of times in order to cope with the random variance. Had we implemented the simulation in Java, JIT compilation would have been performed over and over again, every time the simulation is executed, which is clearly a significant waste of computation power and thus runtime.

A second argument in favour of C++ as a language for the implementation of the framework concerns the frequent recent and future changes to the standard. With C++11, C++14, and C++17, C++ is in a process of significant transformation with respect to its look-and-feel. Features such as generic lambdas, enhanced type inference (e.g. return type deduction and the `auto` keyword), variadic templates, or range-based for-loops make C++ significantly easier to use and the resulting code significantly less verbose and cryptic. Updates to the Standard Template Library (STL) as well as rapid evolution of the Boost libraries further contribute to this trend. In general it can be said that, whilst still being largely downwards compatible, modern C++ code is rarely comparable with that of 10 years ago.

Regarding the second point (the popularity of TMP), it is certainly true that developers cannot be expected to be experienced or even proficient in TMP. However, in our opinion, the important point is *that they do not have to be*. As mentioned above, while TMP is perfectly useful for the development of custom applications, we believe that its particular strength lies in the design and implementation of libraries. With its capability to abstract away type-specific computations and to implement embedded domain-specific languages, TMP is a particularly useful technique for developing convenient interfaces to complex framework. In a sense, due to its particular characteristics, TMP is capable of 'hiding itself'. As such, we believe that TMP should be used as a means to an end (e.g. to achieve high performance or to hide some of the complex computations behind a convenient type interface) but should, on the other hand, be largely hidden from the user. For a user of the BDI framework developed in the context of this work, TMP is almost entirely invisible. The only 'hint' is the encoding of typelists (e.g. the plan bodies or the plan library itself) by means of Boost.MPL. If desired, this could be easily improved further by replacing MPL lists with variadic templates.

In summary, we fully agree that TMP represents a highly specialised technology in which conventional users (even technical users) cannot be expected to have experience. But we also believe that the two biggest strengths of TMP are (i) its power of abstraction, and (ii) its capability to hide itself behind a convenient, domain-specific interface. The experience we gained

in the project is thus that TMP should — with all its complexity and eccentricity — be used to layer out as much of the application logic as possible into the type system and type-specific computations. In doing so, the computations should be defined in such a way that they eventually disappear behind the interface. The result was described in Section 5.5 further above.

Argument 2: Not all computations can be performed at compile time

Discerning those computations that can be performed at compile time from those that have to be performed at runtime is, in our experience, an essential part of the development. Similar to purely functional programming in Haskell where a significant amount of time has to be spent on the design of data types and functions prior to the implementation of the actual functionality, TMP also requires a substantial amount of conceptual work in advance. This involves, in particular, the identification of compile time computations. Some computations may be more or less obvious (e.g. plan selection based on a statically known criterion such as 'always pick the first applicable plan'); others may be harder to identify. Consider, for example, the idea of subgoal expansion performed at compile time as described in Section 3.4.2. Even in cases where a definite decision about which concrete plan to substitute the goal with cannot be made at compile time, the decision logic can, at least, be inlined by replacing the goal with an appropriate selection mechanism. Although this seems like a small change, it helps to remove one or even several layers of indirection which, when running the same simulation thousands of times, can certainly have a significant impact on the overall performance.

On the other hand, it is clear that pure compile time computation will rarely be sufficient if problems more complex than the calculation of factorials need to be solved. As described at various points in this thesis, at some point, the bridge between compile time and runtime will invariably have to be crossed. Whereas Boost.MPL is of great help at compile time, Boost.Fusion has proven to be particularly useful in the hybrid world. With its capability to support both compile time and runtime calculations in parallel, it can be usefully employed for the purpose of multistage programming. Nevertheless, at some point, the static world will have been left ultimately and the utility of Boost.Fusion diminishes. This is the point where dynamic polymorphism starts to become useful.

Our experience in the project has shown that no particular technique is entirely sufficient, but that a right combination needs to be found. In particular, the conceptual subdivision of computation into compile-time, hybrid, and runtime has proven useful. An example of a pure compile time calculation is the expansion of plans described in Section 5.2.2. For the expansion, no runtime information is necessary. An example of a hybrid computation is the applicability checking mechanism, also described in Section 5.2.2. The reason for the hybrid nature of the computation is that it is succeeded by plan selection which may itself happen both at compile time and at runtime. As a consequence, applicability checking needs to be designed such that it supports any subsequent plan selection mechanism, regardless of whether it happens at compile time or at runtime. This is the purpose of the Boost.Fusion containers and algorithms used within the applicability checking mechanism. An example for pure runtime computations are the plan selection mechanisms described in Section 5.3. Here, inheritance hierarchies and dynamic polymorphism are used to achieve a good balance between performance, separation of concerns, and reusability. In particular, inheritance helps to avoid code duplication and to hide away some of the common aspects of different runtime selection mechanisms; dynamic polymorphism allows a user to plug in custom selection mechanisms and use them within the context of the framework.

Argument 3: The larger the problem gets, the less maintainable the TMP code becomes

It is perfectly true that the syntax of C++ templates is anything but user-friendly and elegant. As a consequence, even conceptually simple computations quickly become unwieldy. This is further complicated by the fact that the compiler output with respect to template instantiation errors is famously incomprehensible[1].

On the other hand, TMP has shown to be a purely functional, Turing-complete programming language. As such, it shares two important similarities with other purely functional programming languages like Haskell. With its extreme focus on succinctness and elegance, Haskell can, in a sense, be seen as the diametrally opposite purely functional counterpart of TMP. It is thus not surprising that the relationship between the two languages has sparked considerable interest in recent years [32]. Due to the close correspondence between the two languages, the translation of code from one language into the other is surprisingly trivial; in fact, it can almost be done automatically. Haskell can thus serve as a specification language for TMP code — an idea that is also used by Bartosz Milewski in his forthcoming book "Category theory for programmers"[2]. Once the correspondence between Haskell and C++ is clear and Haskell is used as a sketching language for TMP algorithms, the TMP implementation reduces to an almost trivial, mechanical translation. Clearly, the translation still has to be done manually and, although tool support for round-trip engineering of Haskell and C++ TMP code would be highly desirable, this is still an open technical problem. Nevertheless, the common purely functional nature provides a strong unifying conceptual framework which facilitates the implementation of TMP code considerably.

Unfortunately, the situation is slightly more complicated if an application has to move seamlessly between compile time and runtime computation, as described above. Although runtime computations in C++ can equally well be realised in a purely functional way, this is often not desirable for performance reasons. As soon as the application comprises a mix of static and dynamic polymorphism, the developer faces a multi-paradigm situation for which we are not aware of any conceptually clear and theoretically sound way of specification. Haskell as a specification language is fine for the purely functional parts realised by the TMP code; as soon as Haskell is used to describe those parts of the C++ code that are realised using dynamic polymorphism, however, the correspondence between the two languages begins to diminish. We are not aware of any methodological approach to deal with this problem, although this would — in our experience — be certainly desirable.

7.3 Summary

In summary it can be said that working with C++ TMP in greater depth than one would in a conventional development project was a challenging but highly rewarding experience. Although the learning curve of TMP in general and related libraries such as Loki, Boost.MPL, or Boost.Fusion in particular is very high, it offers deep, interesting and useful insights into the structure of computation. As described above, TMP forces the developer to have a clear and conceptually deep picture of the application logic prior to the actual implementation. In that sense, TMP clearly shows its purely functional face. The syntactic burden of TMP can be easily

[1] The idea of `concepts` has been proposed as a solution to this problem, but it has not yet found its way into the standard.

[2] In his blog, Milewski writes: "You don't have to become a Haskell programmer, but you need it as a language for sketching and documenting ideas to be implemented in C++. That's exactly how I got started with Haskell. I found its terse syntax and powerful type system a great help in understanding and implementing C++ templates, data structures, and algorithms." [27]

overcome by exploiting its functional nature: in combination with Haskell, TMP appears much less daunting than it does in the beginning. Nevertheless, as described above, TMP is rarely used in isolation; as soon as compile time and runtime computation has to be mixed (which can be assumed to be the case for most applications), Haskell is no longer sufficient as a specification language. A clear methodological guideline of how such multi-paradigm applications that move seamlessly between compile time and runtime calculation can be developed efficiently is critically important yet, to the best of our knowledge, still missing.

8 Conclusion & future work

In the age of increasing connectivity, microscopic, agent-based computer simulation will become an indispensable tool in many areas and can be expected to reach beyond its current, mostly academic areas of application. The growing complexity of the scenarios to be studied by means of simulation requires increasingly sophisticated tools for their design, implementation, and analysis. As illustrated in Chapters 2 and 3, a shallow, 'myopic' approach for the implementation of individual agents will, in many cases, no longer suffice. The BDI framework represents a well-established conceptual and technological foundation for the implementation of practically reasoning agents. Besides its capability to implement practically reasoning agents that act in complex, unknown environments, it provides a convenient means to 'break up' a static, behaviour-tree-based decision logic into a more flexible, event-based one. Despite its solid scientific grounding and its conceptual complexity, BDI is lightweight enough to be usable within a resource-bounded computational context — in contrast to other cognitive frameworks such as SOAR or ACT-R.

To date, agent-oriented programming has received comparatively little attention in the agent-based modelling and simulation community. As described above, this is mostly due to the fact that agent-based modellers often possess a non-technical background and are thus not familiar with approaches originating in computer science in general and artificial intelligence in particular. On the other hand, in the multiagent systems community, the BDI framework has had considerable impact over the last two decades, with a rich variety of AOP languages resulting from academic research. However, most of those languages focus on the implementation of real systems rather than simulations and thus prioritise usability and interoperability with existing languages and frameworks over efficiency. The goal of this dissertation project was to address this problem and provide an implementation of the BDI framework with a particular focus on its application in the context of agent-based modelling and simulation. Rather than sacrificing behavioural richness for the sake of efficiency, the challenge of the project was to reconcile both aspects: whilst still being efficient at runtime, the framework should be easily accessible and seamlessly capable of being integrated into an existing simulation environment. The resulting framework is based on C++, for two major reasons. First, we believe that, among the most popular high-level programming languages, C++ provides the best balance between efficiency and usability — the latter especially against the background of the latest standards C++11, C++14, and C++17; second, with template metaprogramming, C++ comprises a powerful feature for the development of embedded domain-specific languages. It is thus possible to conceal complex computations behind a convenient and easy-to-use interface, yet without compromising on the runtime performance of the application.

With almost two decades of research, the BDI architecture is of great complexity and it would be imprudent to claim that the framework developed as part of this dissertation project offers the same functionality as established AOP languages and frameworks. Rather than implementing a fully-fledged BDI framework with all its intricate details and branches, the project focussed on the conceptual and methodological aspects of the framework design for the pur-

pose of agent-based simulation. The most important limitations of the framework are quickly summarised below.

Event types: At the moment, the framework only supports two types of event: goal addition and removal. As a consequence, the framework does not produce events when a change to the belief base occurs. It is thus currently not possible for an agent to react to belief addition or removal.

Triggering events: At the moment, the framework only supports goal addition as a triggering event for plans. It is currently not possible to formulate a plan that reacts to goal removal.

Plan failure: Due to the lack of goal removal as the triggering event of a plan, it is currently not possible to implement plan failure handling by means of contingency plans.

Agent communication: Although easily implementable as basic actions using plain C++ in the framework, building blocks for inter-agent communication are currently not integrated into the framework. The lack of agent communication mechanisms also excludes the possibility for plan exchange.

For clarity and space, we have omitted the aforementioned aspects from the BDI framework. It would, however, be straightforward to implement them, as briefly described in the following paragraphs. We aim to work on those points as part of our future work.

Event types: In order for the agent to be able to react to events, they need, of course, be produced at some point during the execution. At the current stage, the framework only allows for explicit goals within the plan bodies. Belief addition or removal — both actions that agents in Jason can react to — do currently not produce events. However, as described in Section 5.1.2, the addition and removal of beliefs is under the explicit control of the framework (through functions `addBelief` and `removeBelief`). It would thus be straightforward to enable the belief base to create appropriate events and add them to the agent's event store every time a change to the belief base occurs. The only required change to the framework would be to establish a connection between the belief base and the agent state which could, for example, be realised by means of passing a reference to the agent state to the belief base upon construction.

Triggering events: For the implementation of additional triggering events, one more layer of indirection is required. At the moment, we merely consider goal addition and goal removal and hardcode them as templates (`AGoal` and `RGoal`). In order to decouple the desired action (addition or removal) from the item to be acted upon (goal, belief, etc.), we just require two types, e.g. `EvAdd` and `EvRemove`, each parametrised with the type to be handled. Furthermore, in order to identify the kind of type to be acted upon (goal, belief, etc.), user-defined types are required to derive from appropriate base classes `Goal`, `Belief`, etc. They would also have to be added to the framework. Finally, metafunction classes `expandPlan` and `expandEvent` need to be changed such that they take into account the additional layer of indirection incurred by `EvAdd` and `EvRemove`.

Plan failure: As described by Bordini *et al.* [6], plan failure can be handled by means of *contingency plans*. A contingency plan can be seen as analogous to a conventional plan. Its triggering event is a *goal removal event* which, as described in Section 5.3, is created every time a plan fails. The only requirement for contingency plans to work is the capability of the framework to allow for goal removal as a plan's triggering event. The realisation of plan failure is thus closely related with the problem of triggering events described further above.

Agent communication: Strictly speaking, agent communication is not part of the core BDI framework. Nevertheless, as shown by Bordini *et al.* [6], it can be usefully integrated. In order for communication to be realisable efficiently, appropriate functions (e.g. `send` or `broadcast`) are required; they can be easily implemented (e.g. as basic actions) and integrated into the plan formulation process. Furthermore, once belief addition and removal is ensured to throw appropriate events as described above, agents can be enabled to react to received messages in a straightforward way.

Bibliography

[1] D. Abrahams and A. Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series)*. Addison-Wesley Professional, 2004.

[2] A. Alexandrescu. *Modern C++ design: generic programming and design patterns applied*. Addison-Wesley, 2001.

[3] R. Bordini, L. Braubach, M. Dastani, A. El Fallah-Seghrouchni, J.J. Gómez-Sanz, J. Leite, G.M.P. O'Hare, A. Pokahr, and A. Ricci. A survey of programming languages and platforms for multi-agent systems. *Informatica (Slovenia)*, 30(1):33–44, 2006.

[4] R. Bordini and J. F. Hübner. *A Java-based interpreter for an extended version of AgentSpeak*. University of Durham, Universidade Regional de Blumenau, 2007. http://jason.sourceforge.net/Jason.pdf (accessed 11/2015).

[5] R. H. Bordini and J. F. Hübner. Agent-based simulation using BDI programming in Jason. In A.M. Uhrmacher and D. Weyns, editors, *Multi-Agent Systems: Simulation and Applications*, pages 451–471. Citeseer, 2009.

[6] R. H. Bordini, J. F. Hübner, and M. Wooldridge. *Programming Multi-agent Systems in AgentSpeak using Jason*, volume 8. John Wiley & Sons, 2007.

[7] M. Bratman. *Intention, Plans, and Practical Reason*. Center for the Study of Language and Information, Stanford University, Stanford, 1987.

[8] A. Champandard. Understanding behaviour trees. http://aigamedev.com/open/article/bt-overview/, Sep 2007. (accessed 11/2015).

[9] N. Collier. Repast: An extensible framework for agent simulation. *Natural Resources and Environmental Issues*, 8, 2001.

[10] K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

[11] M. Dastani. 2APL: a practical agent programming language. *Autonomous Agents and Multi-Agent Systems*, 16:214–248, 2008.

[12] M. d'Inverno, D. Kinny, M. Luck, and M. Wooldridge. A formal specification of dMARS. In M. Singh, A. Rao, and M. Wooldridge, editors, *Intelligent agents IV: agent theories, architectures, and languages*, volume 1365 of *Lecture Notes in Computer Science*, pages 155–176. Springer, 1998.

[13] M. d'Inverno and M. Luck. Engineering AgentSpeak(L): A formal computational model. *Journal of Logic and Computation*, 8(3):233–260, 1998.

[14] M. d'Inverno and M. Luck. *Understanding Agent Systems*. Springer, 2004.

[15] M. d'Inverno, M. Luck, M. Georgeff, D. Kinny, and M. Wooldridge. The dMARS architecture: A specification of the distributed multi-agent reasoning system. *Autonomous Agents and Multi-Agent Systems*, 9:5–53, July 2004.

[16] M. P. Georgeff and A.S. Rao. A profile of the Australian Artificial Intelligence Institute. *IEEE Expert*, 11(6):89–92, December 1996.

[17] D. Goldsman. Introduction to simulation. In *Proceedings of the 39th Winter Simulation Conference (WSC '07)*, pages 26–37, 2007.

[18] B. Herd. *Statistical runtime verification of agent-based simulations*. PhD thesis, King's College London, 2015.

[19] F. F. Ingrand, M. P. Georgeff, and A. S. Rao. An architecture for real-time reasoning and system control. *IEEE Expert: Intelligent Systems and Their Applications*, 7(6):34–44, December 1992.

[20] J. Jacky. *The Way of Z: Practical Programming with Formal Methods*. Cambridge University Press, 1997.

[21] W. O. Kermack and A. G. McKendrick. Contributions to the mathematical theory of epidemics. *Bulletin of mathematical biology*, 53(1):33–55, 1991.

[22] C. M. Macal and M. J. North. Tutorial on agent-based modeling and simulation. In *Proceedings of the 37th Winter Simulation Conference (WSC '05)*, pages 2–15, 2005.

[23] C. M. Macal and M. J. North. Tutorial on agent-based modelling and simulation. *Journal of Simulation*, 4(3):151–162, 2010.

[24] V. Mascardi, D. Demergasso, and D. Ancona. Languages for programming BDI-style agents: an overview. In *Proceedings of the 6th AI*IA/TABOO Joint Workshop "From Objects to Agents" (WOA '05)*, pages 9–15, 2005.

[25] F. McCabe and K. Clark. April - agent process interaction language. In M. Wooldridge and N. Jennings, editors, *Intelligent Agents*, volume 890 of *Lecture Notes in Computer Science*, pages 324–340. Springer, 1995.

[26] B. McNamara and Y. Smaragdakis. Functional programming in C++ using the FC++ library. *ACM SIGPLAN Notices*, 36(4):25–30, 2001.

[27] B. Milewski. Category theory for programmers. http://bartoszmilewski.com/2014/10/28/category-theory-for-programmers-the-preface/, 2014. (accessed 10/2015).

[28] J. H. Miller and S. E. Page. *Complex Adaptive Systems: An Introduction to Computational Models of Social Life*. Princeton Studies in Complexity. Princeton University Press, Princeton, NJ, USA, 2007.

[29] T. Miller and P. McBurney. Multi-agent system specification using TCOZ. In T. Eymann, F. Klügl, W. Lamersdorf, M. Klusch, and M. Huhns, editors, *Multiagent System Technologies*, volume 3550 of *Lecture Notes in Computer Science*, pages 216–221. Springer, 2005.

[30] L. Padgham, D. Scerri, G. Jayatilleke, and S. Hickmott. Integrating BDI reasoning into agent based modeling and simulation. In *Proceedings of the Winter Simulation Conference (WSC '11)*, pages 345–356, 2011.

[31] H. V. D. Parunak, R. Savit, and R. L. Riolo. Agent-based modeling vs. equation-based modeling: A case study and users' guide. In *Proceedings of the 1st International Workshop on Multiagent Systems and Agent-Based Simulation (MABS'98)*, pages 10–25, London, UK, 1998. Springer.

[32] Z. Porkoláb. Functional programming with C++ template metaprograms. In *Central European Functional Programming School*, volume 6299 of *Lecture Notes in Computer Science*, pages 306–353. Springer, 2010.

[33] K. Preston White and R. G. Ingalls. Introduction to simulation. In *Proceedings of the 41st Winter Simulation Conference (WSC'09)*, pages 12–23, 2009.

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

[35] I. Sakellariou, P. Kefalas, and I. Stamatopoulou. Enhancing NetLogo to simulate BDI communicating agents. In J. Darzentas, G. A. Vouros, S. Vosinakis, and A. Arnellos, editors, *Artificial Intelligence: Theories, Models and Applications*, volume 5138 of *Lecture Notes in Computer Science*, pages 263–275. Springer, 2008.

[36] J. R. Searle. *Speech Acts: An Essay in the Philosophy of Language*. Cambridge University Press, 1969.

[37] R. E. Shannon. Introduction to the art and science of simulation. In *Proceedings of the 30th Winter Simulation Conference (WSC'98)*, pages 7–14, 1998.

[38] Y. Shoham. Agent-oriented programming. *Artificial Intelligence*, 60:51–92, March 1993.

[39] G. Smith. *The Object-Z Specification Language*. Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[40] J. M. Spivey. *The Z notation: A Reference Manual*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, 1992.

[41] W. Taha. A gentle introduction to multi-stage programming. In C. Lengauer, D. Batory, C. Consel, and M. Odersky, editors, *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, pages 30–50. Springer, 2004.

[42] T. Veldhuizen. C++ templates are Turing-complete. Technical report, University of Indiana, 2003.

[43] U. Wilensky. NetLogo. Technical report, Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL., 1999.

[44] M. Winikoff. JACK^{TM} intelligent agents: An industrial strength platform. In *Multi-Agent Programming*, volume 15 of *Multiagent Systems, Artificial Societies, And Simulated Organizations*, pages 175–193. Springer, 2005.

[45] M. J. Wooldridge and N. R. Jennings. Intelligent agents: theory and practice. *The Knowledge Engineering Review*, 10(02):115–152, 1995.

A The Z Notation

The following glossary is adapted from the book "The Way of Z" by Jonathan Jacky [20].

Names

a, b	identifiers
d, e	declarations (e.g., $a : A; b : B...$)
f, g	functions
m, n	numbers
p, q	predicates
s, t	sequences
x, y	expressions
A, B	sets
C, D	bags
Q, R	relations
S, T	schemas
X	schema text (e.g., d, $d \mid p$, or S)

Definitions

$a == x$	Abbreviation definition
$T ::= c \mid d\langle\langle U \rangle\rangle$	Free type definition
$[a]$	Introduction of a given set (or $[a, ...]$)
$a\_$	Prefix operator
$\_a$	Postfix operator
$\_a\_$	Infix operator

Logic

$true$	Logical true constant
$false$	Logical false constant
$\neg\, p$	Logical negation, *not*
$p \wedge q$	Logical conjunction, *and*
$p \vee q$	Logical disjunction, *or*
$p \Rightarrow q$	Logical implication
$p \Leftrightarrow q$	Logical equivalence
$\forall X \bullet q$	Universal quantification
$\exists X \bullet q$	Existential quantification
$(\textbf{let}\ a == x; ... \bullet p)$	Local definition

Sets and expressions

$x = y$	Equality
$x \neq y$	Inequality
$x \in A$	Set membership
$x \notin A$	Non-membership
$\varnothing$	Empty set
$A \subseteq B$	Set inclusion
$A \subset B$	Strict set inclusion
$\{x, y, ...\}$	Set display
$\{X \bullet x\}$	Set comprehension
$(\lambda X \bullet x)$	Lambda expression
(**let** $a == x$; ... $\bullet$ y)	Local definition
if p **then** x **else** y	Conditional expression
$(x, y, ...)$	Tuple
(x, y)	Pair
$A \times B \times ...$	Cartesian product
$\mathbb{F} A$	Finite set
$\mathbb{P} A$	Power set
$A \cap B$	Set intersection
$A \cup B$	Set union
$A \setminus B$	Set difference
$x.1$	First element of an ordered pair
$x.2$	Second element of an ordered pair
$\#A$	Number of elements in a set

Relations

$A \leftrightarrow B$	Binary relation ($\mathbb{P}(A \times B)$)
$a \mapsto b$	Maplet ((a, b))
$\mathrm{dom}\, R$	Domain of a relation
$\mathrm{ran}\, R$	Range of a relation
$Q \fatsemi R$	Forward relational composition
$Q \circ R$	Backward relational composition ($R \fatsemi Q$)
$A \lhd R$	Domain restriction
$A \lhd\!\!\!\!- R$	Domain anti-restriction
$A \rhd R$	Range restriction
$A \rhd\!\!\!\!- R$	Range anti-restriction
$R(\!\vert A \vert\!)$	Relational image
$R^{\sim}$	Inverse of relation
R^{+}	Transitive closure
$Q \oplus R$	Relational overriding
$a \mathrel{\underline{\mathrm{R}}} b$	Infix relation

Functions

$A \pfun B$	Partial functions
$A \fun B$	Total functions

$A \rightarrowtail\!\!\!\!\!\!+\; B$	Partial injections
$A \rightarrowtail B$	Total injections
$A \rightarrowtail\!\!\!\!\!\!\twoheadrightarrow B$	Bijections
$f\,x$	Function application (or $f(x)$)

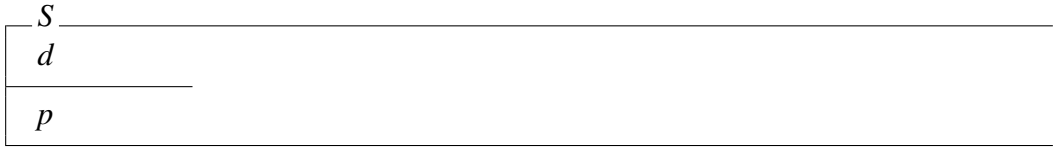
Numbers

$\mathbb{Z}$	Set of integers
$\mathbb{N}$	Set of natural numbers $\{0,1,2,...\}$
$\mathbb{N}_1$	Set of strictly positive numbers $\{1,2,...\}$
	Set of real numbers
$m+n$	Addition
$m-n$	Subtraction
$m*n$	Multiplication
$m \operatorname{div} n$	Division
$m \bmod n$	Remainder (modulus)
$m \leqslant n$	Less than or equal
$m < n$	Less than
$m \geqslant n$	Greater than or equal
$m > n$	Greater than
$m\,..\,n$	Number range
$min\,A$	Minimum of a set of numbers
$max\,A$	Maximum of a set of numbers

Sequences

$\operatorname{seq} A$	Set of finite sequences
$\operatorname{seq}_1 A$	Set of non-empty finite sequences
$\operatorname{iseq} A$	Set of finite injective sequences
$\langle\rangle$	Empty sequence
$\langle x,y,...\rangle$	Sequence display
$s \frown t$	Sequence concatenation
$head\,s$	First element of a sequence
$tail\,s$	All but the head element of a sequence
$last\,s$	Last element of a sequence
$front\,s$	All but the last element of a sequence
$s \operatorname{in} t$	Sequence segment relation

Schema



Axiomatic definition

$$\begin{array}{|l} d \\ \hline p \end{array}$$

Generic definition

$$\begin{array}{|l} [a, \ldots] \\ \hline d \\ \hline p \end{array}$$

Schema calculus

$S \mathrel{\widehat=} [X]$	Horizontal schema
$[T; \ldots \mid \ldots]$	Schema inclusion
$z.a$	Component selection (given $z : S$)
θS	Binding
$\neg\, S$	Schema negation
$S \land T$	Schema conjunction
$S \lor T$	Schema disjunction
$S \fatsemi T$	Schema composition
$S \gg T$	Schema piping

Conventions

$a?$	Input to an operation
$a!$	Output from an operation
a	State component before an operation
a'	State component after an operation
S	State schema before an operation
S'	State schema after an operation
ΔS	Change of state
ΞS	No change of state

B Precondition analyses

On the following pages, detailed preconditions for all state-changing operations are provided. For clarity, we have omitted from the expanded $Agent'$ schema those helper functions (*selectPlan*, *perceive*, etc.) that are **not** used in the respective operation schema.

Perceive

$$\text{pre } Perceive = \exists Agent' \bullet Perceive$$

Expansion of $Agent'$ and *Perceive*:

$$\begin{array}{l} = [Agent \mid \\ \quad \exists beliefs' : Beliefs; events' : Events; \\ \quad intentions' : Intentions; plans : PlanLibrary; capabilities' : Capabilities \mid \\ \qquad (\forall p : \mathrm{dom}\, plans' \bullet (\mathrm{ran}\, p \subseteq capabilities') \land \forall i : intentions' \bullet (\mathrm{ran}\, i \subseteq capabilities')) \bullet \\ \qquad\quad (env?, beliefs) \in \mathrm{dom}\, perceive \land \\ \qquad\quad beliefs' = perceive(env?, beliefs) \land \\ \qquad\quad intentions' = intentions \land \\ \qquad\quad events' = events \land \\ \qquad\quad plans' = plans \land \\ \qquad\quad capabilities' = capabilities] \end{array}$$

One-point rule for *intentions*, *events*, *plans*, and *capabilities*:

$$\begin{array}{l} = [Agent \mid \\ \quad \exists beliefs' : Beliefs \bullet \\ \qquad (\forall p : \mathrm{dom}\, plans \bullet \mathrm{ran}\, p \subseteq capabilities) \land \\ \qquad (\forall i : intentions \bullet \mathrm{ran}\, i \subseteq capabilities) \land \\ \qquad (env?, beliefs) \in \mathrm{dom}\, perceive \land \\ \qquad beliefs' = perceive(env?, beliefs)] \end{array}$$

One-point rule for *beliefs*:

$$\begin{array}{l} = [Agent \bullet \\ \quad (\forall p : \mathrm{dom}\, plans \bullet \mathrm{ran}\, p \subseteq capabilities) \land \\ \quad (\forall i : intentions \bullet \mathrm{ran}\, i \subseteq capabilities) \land \\ \quad (env?, beliefs) \in \mathrm{dom}\, perceive \land \\ \quad perceive(env?, beliefs) \in BeliefBase] \end{array}$$

Definition of *perceive*:

$$\begin{array}{l} = [Agent \bullet \\ \quad (\forall p : \mathrm{dom}\, plans \bullet \mathrm{ran}\, p \subseteq capabilities) \land \\ \quad (\forall i : intentions \bullet \mathrm{ran}\, i \subseteq capabilities) \land \\ \quad (env?, beliefs) \in \mathrm{dom}\, perceive] \end{array}$$

PickEvent

$$\text{pre } PickEvent = \exists Agent' \bullet PickEvent$$

Expansion of $Agent'$ and $PickEvent$:

$$\begin{array}{l}
= [Agent \mid \\
\quad \exists beliefs' : Beliefs; events' : Events; \\
\quad intentions' : Intentions; plans : PlanLibrary; capabilities' : Capabilities \mid \\
\qquad (\forall p : \mathrm{dom}\, plans' \bullet (\mathrm{ran}\, p \subseteq capabilities') \land \forall i : intentions' \bullet (\mathrm{ran}\, i \subseteq capabilities')) \bullet \\
\qquad\quad events \neq \varnothing \land \\
\qquad\quad beliefs' = beliefs \land \\
\qquad\quad intentions' = intentions \land \\
\qquad\quad events' \subset events \land \\
\qquad\quad plans' = plans \land \\
\qquad\quad capabilities' = capabilities]
\end{array}$$

One-point rule for *beliefs*, *intentions*, *plans*, and *capabilities*:

$$\begin{array}{l}
= [Agent \mid \\
\quad \exists events' : Events \bullet \\
\qquad (\forall p : \mathrm{dom}\, plans \bullet \mathrm{ran}\, p \subseteq capabilities) \land \\
\qquad (\forall i : intentions \bullet \mathrm{ran}\, i \subseteq capabilities) \land \\
\qquad events \neq \varnothing \land \\
\qquad events' \subset events]
\end{array}$$

By definition $events \in Events$ holds. As a consequence, if $events' \subset events$, then $events' \in Events$ also holds. We can thus apply the one-point-rule for *events* and reduce the schema above as follows.

$$\begin{array}{l}
= [Agent \bullet \\
\quad (\forall p : \mathrm{dom}\, plans \bullet \mathrm{ran}\, p \subseteq capabilities) \land \\
\quad (\forall i : intentions \bullet \mathrm{ran}\, i \subseteq capabilities) \land \\
\quad events \neq \varnothing]
\end{array}$$

AddIntention

$$\text{pre } AddIntention = \exists Agent' \bullet AddIntention$$

Expansion of $Agent'$ and $AddIntention$:

$$\begin{array}{l}
= [Agent; i? : Intention \mid \\
\quad \exists beliefs' : Beliefs; events' : Events; \\
\quad intentions' : Intentions; plans : PlanLibrary; capabilities : Capabilities \mid \\
\qquad (\forall p : \mathrm{dom}\, plans' \bullet (\mathrm{ran}\, p \subseteq capabilities') \land \forall i : intentions' \bullet (\mathrm{ran}\, i \subseteq capabilities')) \bullet \\
\qquad beliefs' = beliefs \land \\
\qquad intentions' = intentions \cup \{i?\} \land \\
\qquad events' = events \land \\
\qquad plans' = plans \land \\
\qquad capabilities' = capabilities]
\end{array}$$

One-point rule for *beliefs*, *events*, *plans*, and *capabilities*:

$$= [Agent; i? : intention \mid \exists intentions' : Intentions \bullet (\forall p : \mathrm{dom}\, plans \bullet \mathrm{ran}\, p \subseteq capabilities) \land (\forall i : intentions' \bullet \mathrm{ran}\, i \subseteq capabilities) \land intentions' = intentions \cup \{i?\}]$$

One-point rule for *intentions*:

$$= [Agent; i? : intention \mid (\forall p : \mathrm{dom}\, plans \bullet \mathrm{ran}\, p \subseteq capabilities) \land (\forall i : intentions \cup \{i?\} \bullet \mathrm{ran}\, i \subseteq capabilities) \land intentions \cup \{i?\} \in Intentions]$$

Definition of $\cup$:

$$= [Agent; i? : intention \bullet (\forall p : \mathrm{dom}\, plans \bullet \mathrm{ran}\, p \subseteq capabilities) \land (\forall i : intentions \bullet \mathrm{ran}\, i \subseteq capabilities) \land i? \in capabilities]$$

RemoveEvents

$$\mathrm{pre}\ RemoveEvents = \exists Agent' \bullet RemoveEvents$$

Expansion of $Agent'$ and *RemoveEvents*:

$$= [Agent; g? : Goal \mid \exists beliefs' : Beliefs; events' : Events; intentions' : Intentions; plans : PlanLibrary; capabilities : Capabilities \mid (\forall p : \mathrm{dom}\, plans' \bullet (\mathrm{ran}\, p \subseteq capabilities') \land \forall i : intentions' \bullet (\mathrm{ran}\, i \subseteq capabilities')) \bullet beliefs' = beliefs \land intentions' = intentions \land events' = events \setminus \{Add(g?)\} \land plans' = plans \land capabilities' = capabilities]$$

One-point rule for $beliefs'$, $intentions'$, $plans'$, and $capabilities'$:

$$= [Agent; g? : Goal \mid \exists events' : Events \bullet (\forall p : \mathrm{dom}\, plans \bullet \mathrm{ran}\, p \subseteq capabilities) \land (\forall i : intentions \bullet \mathrm{ran}\, i \subseteq capabilities) \land events' = events \setminus \{Add(g?)\}]$$

One-point rule for $events'$:

$$= [Agent; g? : Goal \bullet (\forall p : \mathrm{dom}\, plans \bullet \mathrm{ran}\, p \subseteq capabilities) \land (\forall i : intentions \bullet \mathrm{ran}\, i \subseteq capabilities) \land events \setminus \{Add(g?)\} \in Events]$$

Definition of $\setminus$:

$$= [Agent; g? : Goal \bullet (\forall p : \mathrm{dom}\, plans \bullet \mathrm{ran}\, p \subseteq capabilities) \land (\forall i : intentions \bullet \mathrm{ran}\, i \subseteq capabilities)]$$

PickIntention

$$\text{pre } PickIntention = \exists Agent' \bullet PickIntention$$

Expansion of $Agent'$ and $PickIntention$:

$$\begin{aligned}
&= [Agent \mid \exists beliefs' : Beliefs; events' : Events; \\
&\quad intentions' : Intentions; plans : PlanLibrary; capabilities : Capabilities \mid \\
&\quad (\forall p : \operatorname{dom} plans' \bullet (\operatorname{ran} p \subseteq capabilities') \land \forall i : intentions' \bullet (\operatorname{ran} i \subseteq capabilities')) \bullet \\
&\qquad intentions \neq \varnothing \land \\
&\qquad beliefs' = beliefs \land \\
&\qquad intentions' \subset intentions \land \\
&\qquad events' = events \land \\
&\qquad plans' = plans]
\end{aligned}$$

One-point rule for *beliefs*, *events*, *plans*, and *capabilities*:

$$\begin{aligned}
&= [Agent \mid \exists intentions' : Intentions \bullet \\
&\quad (\forall p : \operatorname{dom} plans \bullet \operatorname{ran} p \subseteq capabilities) \land \\
&\quad (\forall i : intentions' \bullet \operatorname{ran} i \subseteq capabilities) \land \\
&\quad intentions' \subset intentions]
\end{aligned}$$

By definition $intentions \in Intentions$ holds. As a consequence, if $intentions' \subset intentions$, then $intentions' \in Intentions$ also holds. Thus, if $\forall i : intentions \bullet \operatorname{ran} i \subseteq capabilities$ holds, then $\forall i : intentions' \bullet \operatorname{ran} i \subseteq capabilities$ also holds. We can thus apply the one-point-rule for *intentions* and reduce the schema above as follows.

$$\begin{aligned}
&= [Agent \bullet \\
&\quad (\forall p : \operatorname{dom} plans \bullet \operatorname{ran} p \subseteq capabilities) \land \\
&\quad (\forall i : intentions \bullet \operatorname{ran} i \subseteq capabilities)]
\end{aligned}$$

ExecuteAction

$$\text{pre } ExecuteAction = \exists Agent' \bullet ExecuteAction$$

Expansion of $Agent'$ and $ExecuteAction$:

$$\begin{aligned}
&= [Agent; i? : Intention; p? : PlanItem \mid \\
&\quad \exists beliefs' : Beliefs; events' : Events; \\
&\quad intentions' : Intentions; plans : PlanLibrary; capabilities : Capabilities \mid \\
&\quad (\forall p : \operatorname{dom} plans' \bullet (\operatorname{ran} p \subseteq capabilities') \land \forall i : intentions' \bullet (\operatorname{ran} i \subseteq capabilities')) \bullet \\
&\qquad (\exists a : Ac \bullet \\
&\qquad\quad p? = Action(a) \land \\
&\qquad\quad (a, beliefs) \in \operatorname{dom} executeAction \land \\
&\qquad\quad beliefs' = executeAction(a, beliefs)) \land \\
&\qquad events' = events \land \\
&\qquad intentions' = intentions \land \\
&\qquad plans' = plans \land \\
&\qquad capabilities' = capabilities]
\end{aligned}$$

One-point rule for *events*, *intentions*, *plans*, and *capabilities*:

$$
\begin{array}{l}
= [Agent; i? : Intention; p? : PlanItem \mid \\
\quad \exists beliefs' : Beliefs \bullet \\
\qquad (\forall p : \mathrm{dom}\, plans \bullet \mathrm{ran}\, p \subseteq capabilities) \land \\
\qquad (\forall i : intentions \bullet \mathrm{ran}\, i \subseteq capabilities) \land \\
\qquad (\exists a : Ac \bullet \\
\qquad\quad p? = Action(a) \land \\
\qquad\quad (a, beliefs) \in \mathrm{dom}\, executeAction \land \\
\qquad\quad beliefs' = executeAction(a, beliefs))]
\end{array}
$$

One-point rule for *beliefs*:

$$
\begin{array}{l}
= [Agent; i? : Intention; p? : PlanItem \bullet \\
\quad (\forall p : \mathrm{dom}\, plans \bullet \mathrm{ran}\, p \subseteq capabilities) \land \\
\quad (\forall i : intentions \bullet \mathrm{ran}\, i \subseteq capabilities) \land \\
\quad (\exists a : Ac \bullet \\
\qquad p? = Action(a) \land \\
\qquad (a, beliefs) \in \mathrm{dom}\, executeAction \land \\
\qquad executeAction(a, beliefs) \in BeliefBase)]
\end{array}
$$

Definition of $\longrightarrow$ and *executeAction*:

$$
\begin{array}{l}
= [Agent; i? : Intention; p? : PlanItem \bullet \\
\quad (\forall p : \mathrm{dom}\, plans \bullet \mathrm{ran}\, p \subseteq capabilities) \land \\
\quad (\forall i : intentions' \bullet \mathrm{ran}\, i \subseteq capabilities) \land \\
\quad (\exists a : Ac \bullet p? = Action(a) \land (a, beliefs) \in \mathrm{dom}\, executeAction)]
\end{array}
$$

ExpandSubgoal

$$\mathrm{pre}\ ExpandSubgoal = \exists Agent' \bullet ExpandSubgoal$$

Expansion of $Agent'$ and *ExecuteAction*:

$$
\begin{array}{l}
= [Agent; i? : Intention; p? : Plan \mid \\
\quad \exists beliefs' : Beliefs; events' : Events; \\
\quad intentions' : Intentions; plans : PlanLibrary; capabilities : Capabilities \mid \\
\qquad (\forall p : \mathrm{dom}\, plans' \bullet (\mathrm{ran}\, p \subseteq capabilities') \land \forall i : intentions' \bullet (\mathrm{ran}\, i \subseteq capabilities')) \bullet \\
\qquad\quad i? \neq \varnothing \land \\
\qquad\quad i? \in intentions \land \\
\qquad\quad p? \in plans \land \\
\qquad\quad beliefs' = beliefs \land \\
\qquad\quad events' = events \land \\
\qquad\quad intentions' = (intentions \setminus \{i?\}) \cup \{p? \frown (tail\, i?)\} \land \\
\qquad\quad plans' = plans \land \\
\qquad\quad capabilities' = capabilities]
\end{array}
$$

One-point rule for *beliefs*, *events*, *plans*, and *capabilities*:

$$
\begin{array}{l}
= [Agent; i? : Intention; p? : Plan \mid \\
\quad \exists intentions' : Intentions \bullet \\
\qquad (\forall p : \mathrm{dom}\, plans \bullet \mathrm{ran}\, p \subseteq capabilities) \land \\
\qquad (\forall i : intentions' \bullet \mathrm{ran}\, i \subseteq capabilities) \land \\
\qquad i? \neq \varnothing \land \\
\qquad i? \in intentions \land \\
\qquad p? \in plans \land \\
\qquad intentions' = (intentions \setminus \{i?\}) \cup \{p? \frown (tail\, i?)\}]
\end{array}
$$

One-point rule for *intentions*:

$$
\begin{array}{l}
= [Agent; i? : Intention; p? : Plan \bullet \\
\quad (\forall p : \mathrm{dom}\, plans \bullet \mathrm{ran}\, p \subseteq capabilities) \land \\
\quad (\forall i : (intentions \setminus \{i?\}) \cup \{p? \frown (tail\, i?)\} \bullet \mathrm{ran}\, i \subseteq capabilities) \land \\
\quad i? \neq \varnothing \land \\
\quad i? \in intentions \land \\
\quad p? \in plans \land \\
\quad (intentions \setminus \{i?\}) \cup \{p? \frown (tail\, i?)\} \in Intentions]
\end{array}
$$

$i? \in Intentions$ holds. Thus, $(intentions \setminus \{i?\}) \in Intention$ also holds. Furthermore, since $p? \in Plan$ and $tail\, i? \in Intentions$, $\{p? \frown (tail\, i?)\} \in Intentions$ holds by definition of '$\frown$'.

$$
\begin{array}{l}
= [Agent; i? : Intention; p? : Plan \bullet \\
\quad (\forall p : \mathrm{dom}\, plans \bullet \mathrm{ran}\, p \subseteq capabilities) \land \\
\quad (\forall i : (intentions \setminus \{i?\}) \cup \{p? \frown (tail\, i?)\} \bullet \mathrm{ran}\, i \subseteq capabilities) \land \\
\quad i? \neq \varnothing \land \\
\quad i? \in intentions \land \\
\quad p? \in plans]
\end{array}
$$

(i) $\forall x : \{p? \frown (tail\, i?)\} \bullet \mathrm{ran}\, x \subseteq capabilities$ holds iff $\mathrm{ran}\, p? \subseteq capabilities$ (as ensured by $p? \in plans$) and $\mathrm{ran}\, i? \subseteq capabilities$ (as ensured by $i? \in intentions$). (ii) $\forall y : (intentions \setminus \{i?\}) \bullet \mathrm{ran}\, y \in capabilities$ holds iff $\forall i : intentions \bullet \mathrm{ran}\, i \subseteq capabilities$ and $\mathrm{ran}\, i? \subseteq capabilities$ (as ensured by $i? \in intentions$). By definition of $\cup$, $\forall i : (intentions \setminus \{i?\}) \cup \{p? \frown (tail\, i?)\} \bullet \mathrm{ran}\, i \subseteq capabilities$ thus holds iff (i) and (ii) hold. As a consequence, we can simplify the schema above as follows.

$$
\begin{array}{l}
= [Agent; i? : Intention; p? : Plan \bullet \\
\quad (\forall p : \mathrm{dom}\, plans \bullet \mathrm{ran}\, p \subseteq capabilities) \land \\
\quad (\forall i : intentions \bullet \mathrm{ran}\, i \subseteq capabilities) \land \\
\quad i? \neq \varnothing \land \\
\quad i? \in intentions \land \\
\quad p? \in plans]
\end{array}
$$

ExecuteParAddition

$$\mathrm{pre}\, ExecuteParAddition = \exists Agent' \bullet ExecuteParAddition$$

Expansion of $Agent'$ and *ExecuteParAddition*:

$$
\begin{array}{l}
= [Agent; i? : Intention; p? : PlanItem \mid \\
\quad \exists beliefs' : Beliefs; events' : Events; \\
\quad intentions' : Intentions; plans : PlanLibrary; capabilities : Capabilities \mid \\
\qquad (\forall p : \mathrm{dom}\, plans' \bullet (\mathrm{ran}\, p \subseteq capabilities') \land \forall i : intentions' \bullet (\mathrm{ran}\, i \subseteq capabilities')) \bullet \\
\qquad\quad (\exists g : Goal \bullet p? = ParAddition(g) \land events' = events \cup \{Add(g)\}) \land \\
\qquad\quad beliefs' = beliefs \land \\
\qquad\quad intentions' = intentions \land \\
\qquad\quad plans' = plans \land \\
\qquad\quad capabilities' = capabilities]
\end{array}
$$

One-point rule for *beliefs*, *intentions*, *plans*, and *capabilities*:

$$
\begin{array}{l}
= [Agent; i? : Intention; p? : PlanItem \mid \\
\quad \exists events' : Events \bullet \\
\qquad (\forall p : \mathrm{dom}\, plans \bullet \mathrm{ran}\, p \subseteq capabilities) \land \\
\qquad (\forall i : intentions \bullet \mathrm{ran}\, i \subseteq capabilities) \land \\
\qquad (\exists g : Goal \bullet p? = ParAddition(g) \land events' = events \cup \{Add(g)\})]
\end{array}
$$

One-point rule for *events*:

$$
\begin{array}{l}
= [Agent; i? : Intention; p? : PlanItem \bullet \\
\quad (\forall p : \mathrm{dom}\, plans \bullet \mathrm{ran}\, p \subseteq capabilities) \land \\
\quad (\forall i : intentions \bullet \mathrm{ran}\, i \subseteq capabilities) \land \\
\quad (\exists g : Goal \bullet p? = ParAddition(g) \land events \cup \{Add(g)\} \in Events)]
\end{array}
$$

Definition of $\cup$:

$$
\begin{array}{l}
= [Agent; i? : Intention; p? : PlanItem \bullet \\
\quad (\forall p : \mathrm{dom}\, plans \bullet \mathrm{ran}\, p \subseteq capabilities) \land \\
\quad (\forall i : intentions \bullet \mathrm{ran}\, i \subseteq capabilities) \land \\
\quad (\exists g : Goal \bullet p? = ParAddition(g))]
\end{array}
$$

ExecuteRemoval

$$\mathrm{pre}\ ExecuteRemoval = \exists Agent' \bullet ExecuteRemoval$$

Expansion of $Agent'$ and *ExecuteRemoval*:

$$
\begin{array}{l}
= [Agent; i? : Intention; p? : PlanItem \mid \\
\quad \exists beliefs' : Beliefs; events' : Events; \\
\quad intentions' : Intentions; plans : PlanLibrary; capabilities : Capabilities \mid \\
\qquad (\forall p : \mathrm{dom}\, plans' \bullet (\mathrm{ran}\, p \subseteq capabilities') \land \forall i : intentions' \bullet (\mathrm{ran}\, i \subseteq capabilities')) \bullet \\
\qquad\quad (\exists g : Goal \bullet p? = Removal(g) \land events' = events \cup \{Remove(g)\} \land \\
\qquad\quad beliefs' = beliefs \land \\
\qquad\quad intentions' = intentions \land \\
\qquad\quad plans' = plans \land \\
\qquad\quad capabilities' = capabilities)]
\end{array}
$$

One-point rule for *beliefs*, *intentions*, *plans*, and *capabilities*:

$$= [Agent; i? : Intention; p? : PlanItem \mid$$
$$\exists events' : Events \bullet$$
$$(\forall p : \mathrm{dom}\, plans \bullet \mathrm{ran}\, p \subseteq capabilities) \land$$
$$(\forall i : intentions \bullet \mathrm{ran}\, i \subseteq capabilities) \land$$
$$(\exists g : Goal \bullet p? = Removal(g) \land events' = events \cup \{Remove(g)\})]$$

One-point rule for *events*:

$$= [Agent; i? : Intention; p? : PlanItem \bullet$$
$$(\forall p : \mathrm{dom}\, plans \bullet \mathrm{ran}\, p \subseteq capabilities) \land$$
$$(\forall i : intentions \bullet \mathrm{ran}\, i \subseteq capabilities) \land$$
$$(\exists g : Goal \bullet p? = Removal(g) \land events \cup \{Remove(g)\} \in Events)]$$

Definition of $\cup$:

$$= [Agent; i? : Intention; p? : PlanItem \bullet$$
$$(\forall p : \mathrm{dom}\, plans \bullet \mathrm{ran}\, p \subseteq capabilities) \land$$
$$(\forall i : intentions \bullet \mathrm{ran}\, i \subseteq capabilities) \land$$
$$(\exists g : Goal \bullet p? = Removal(g))]$$

Cleanup

Expansion of $Agent'$ and *Cleanup*:

$$= [Agent \mid$$
$$\exists beliefs' : Beliefs; events' : Events;$$
$$intentions' : Intentions; plans : PlanLibrary; capabilities : Capabilities \mid$$
$$(\forall p : \mathrm{dom}\, plans' \bullet (\mathrm{ran}\, p \subseteq capabilities') \land \forall i : intentions' \bullet (\mathrm{ran}\, i \subseteq capabilities')) \bullet$$
$$beliefs' = beliefs \land$$
$$events' = events \land$$
$$intentions' = intentions \setminus \{\langle\rangle\} \land$$
$$plans' = plans \land$$
$$capabilities' = capabilities]$$

One-point rule for *beliefs*, *events*, *plans*, and *capabilities*:

$$= [Agent \mid$$
$$\exists intentions' : Intentions \bullet$$
$$(\forall p : \mathrm{dom}\, plans \bullet \mathrm{ran}\, p \subseteq capabilities) \land$$
$$(\forall i : intentions' \bullet \mathrm{ran}\, i \subseteq capabilities) \land$$
$$intentions' = intentions \setminus \{\langle\rangle\}]$$

One-point rule for *intentions*:

$$= [Agent \bullet$$
$$(\forall p : \mathrm{dom}\, plans \bullet \mathrm{ran}\, p \subseteq capabilities) \land$$
$$(\forall i : (intentions \setminus \{\langle\rangle\}) \bullet \mathrm{ran}\, i \subseteq capabilities)]$$

Definition of $\setminus$:

$$= [Agent \bullet$$
$$(\forall p : \mathrm{dom}\, plans \bullet \mathrm{ran}\, p \subseteq capabilities) \land$$
$$(\forall i : intentions \bullet \mathrm{ran}\, i \subseteq capabilities)]$$

C C++ sources

C.1 Control flow

Listing C.1: The base class for control flow nodes (sequences & selectors)

```
template <class Body_, class Type, const char* Param = g_sEmpty>
struct CtrlFlow {
  BOOST_MPL_ASSERT(( boost::mpl::is_sequence<Body_> ));
  typedef Body_ Body;

  // necessary, e.g., to erase it from a vector (comparison is needed here)
  bool operator==(CtrlFlow<Body_,Type,Param> const&) const { return true; }

  void setParam(Type const& _param) { param = _param; }

  Type param;
};

template <class Body_, const char* Param>
struct CtrlFlow<Body_,void,Param> {
  BOOST_MPL_ASSERT(( boost::mpl::is_sequence<Body_> ));
  typedef Body_ Body;

  bool operator==(CtrlFlow<Body_,void,Param> const&) const { return true; }
};
```

C.2 Sequence

Listing C.2: The sequence class hierarchy (part 1/3)

```
template <int Name, class Event, class Plans, class ParamType>
struct SequenceBase : CtrlFlow<Plans, ParamType> {
  typedef typename boost::make_variant_over<Plans>::type VariantType;
  typedef std::queue<VariantType> PlanVector;
  typedef Event event;

  SequenceBase() {
    boost::mpl::for_each<Plans>([this](auto plan) {
      vec.push(plan);
    });
```

```
  }

  SequenceBase(Plans plans) {
    boost::fusion::for_each(plans, [this](auto plan) {
      vec.push(plan);
    });
  }

  template <class AS>
  bool operator()(AS& as) {
    if(!done())
      return apply_visitor(executePlan<AS>(as), vec.front());
  }

  bool done() const {
    return (vec.empty());
  }

  PlanVector vec;
};
```

Listing C.3: The sequence class hierarchy (part 2/3)

```
/*
 * Parametrised case of Sequence
 */
template <int Name, class Event, class Plans, class ParamType>
struct Sequence : SequenceBase<Name, Event, Plans, ParamType> {
  Sequence() : SequenceBase<Name,Event,Plans,ParamType>() {}
  Sequence(Plans plans) : SequenceBase<Name,Event,Plans,ParamType>(plans) {}

  void setParam(ParamType const& param) {
    this->param = param;
    apply_visitor(setParameter<ParamType>(param),this->vec.front());
  }

  void advance() {
    if(!this->done()) {
      apply_visitor(advancePlan(), this->vec.front());
      apply_visitor(setParameter<ParamType>(this->param),this->vec.front());
      if(apply_visitor(isDone(), this->vec.front())) {
        this->vec.pop();
      }
    }
  }

  template <class AS>
  bool _applicable(AS const& as, ParamType const& param) const {
    return applicable<Name, ParamType>()(as, param);
  }
```

```
  template <class AS>
  float _utility(AS const& as, ParamType const& param) const {
    return utility<Name,ParamType>()(as, param);
  }
};
```

Listing C.4: The sequence class hierarchy (part 3/3)

```
/*
 * Non-Parametrised case of Sequence
 */
template <int Name, class Goal, class Plans>
struct Sequence<Name,Goal,Plans,void> : SequenceBase<Name, Goal, Plans, void> {
  Sequence() : SequenceBase<Name,Goal,Plans,void>() {}
  Sequence(Plans plans) : SequenceBase<Name,Goal,Plans,void>(plans) {}

  void advance() {
    if(!this->done()) {
      apply_visitor(advancePlan(), this->vec.front());
      if(apply_visitor(isDone(), this->vec.front())) {
        this->vec.pop();
      }
    }
  }

  template <class AS>
  bool _applicable(AS& as) const {
    return applicable<Name, void>()(as);
  }

  template <class AS>
  float _utility(AS const& as) const {
    return utility<Name>()(as);
  }
};
```

C.3 Utility calculation

Listing C.5: Utility base class hierarchy (part 1/1)

```
template <class T, class AS, class ParamType>
struct getUtility_2 {};

template <
  int Name,
  class Event,
  class Body,
  class PT,
  class AS,
  class ParamType
>
struct getUtility_2<Sequence<Name, Event, Body, PT>,AS,ParamType> {
  float operator()(
    Sequence<Name, Event, Body, PT> const& plan,
    AS const& as,
    ParamType const& param
  ) const {
    return plan._utility(as,param);
  }
};

template <int Name, class Event, class Body, class ParamType, class AS>
struct getUtility_2<Sequence<Name, Event, Body, ParamType>,AS,void> {
  float operator()(
    Sequence<Name, Event, Body, ParamType> const& plan,
    AS const& as
  ) const {
    return plan._utility(as);
  }
};
```

Listing C.6: Utility base class hierarchy (part 2/2)

```
template <class T, class AS, class ParamType=void>
struct getUtility_ : getUtility_2<T,AS,ParamType> {};

template <class AS, class ParamType>
struct getUtility : boost::static_visitor<float> {
  getUtility(AS const& _as, ParamType const& _param)
  : as(_as),
    param(_param)
  {}

  template <class P>
  float operator()(P const& plan) const {
    return getUtility_<P,AS,ParamType>()(plan,as,param);
```

```
  }

  AS const& as;
  ParamType param;
};

template <class AS>
struct getUtility<AS,void> : boost::static_visitor<float> {
  getUtility(AS const& _as)
  : as(_as)
  {}

  template <class P>
  float operator()(P const& plan) const {
    return getUtility_<P,AS>()(plan,as);
  }

  AS const& as;
};
```

D Party scenario (C++ baseline)

```
/*
 * partyScenario_baseline.h
 *
 *  Created on: 2 Mar 2015
 *      Author: Benjamin C. Herd
 */

#ifndef SRC_PARTYSCENARIO_BASELINE_H_
#define SRC_PARTYSCENARIO_BASELINE_H_

#include <unordered_map>
#include <set>
#include <iostream>
#include <map>
#include <functional>

#include <boost/range/irange.hpp>

#include "bdi4mabs.h"

using namespace std;
using namespace boost;
using namespace boost::mpl;

////////////////////////////////////////
///// ENVIRONMENT
////////////////////////////////////////

struct Environment
{
  template <class AS>
  void perceive(AS const& as)
  {
  }
};

////////////////////////////////////////
///// AGENT LOGIC
////////////////////////////////////////

float ut_getMoneyFromFriend() {
```

```
  return 1.0;
}

float ut_getMoneyFromATM() {
  return 1.0;
}

void walkToShop() {
#ifdef DEBUG_OUTPUT
  cout << "Walk to shop" << endl << flush;
#endif
}

void purchasePresent() {
#ifdef DEBUG_OUTPUT
  cout << "Purchase present" << endl << flush;
#endif
}

void getMoneyFromFriend() {
#ifdef DEBUG_OUTPUT
  cout << "Get money from friend" << endl << flush;
#endif
}

void getMoneyFromATM() {
#ifdef DEBUG_OUTPUT
  cout << "Get money from ATM" << endl << flush;
#endif
}

void getMoney() {
  std::map<float,std::function<void()>> utilityMap;
  float total = 0;

  // collect utility values
  float utFriend = ut_getMoneyFromFriend();
  total += utFriend;
  utilityMap[total] = getMoneyFromFriend;
  float utATM = ut_getMoneyFromATM();
  total += utATM;
  utilityMap[total] = getMoneyFromATM;

  // perform utility evaluation
  std::uniform_real_distribution<> dist(0,1);
  float rand = dist(rng);
  for(auto it=utilityMap.begin(); it!=utilityMap.end(); ++it) {
    if(rand<(it->first/total)) {
      it->second();
    }
  }
```

```
}

void buyPresent() {
  getMoney();
  walkToShop();
  purchasePresent();
}

void tick() {
  buyPresent();
}

void run() {
  Environment env;

  for(int i=0; i<1000; i++) {
    for(int j=0; j<1000; j++) {
      tick();
    }
    cout << "Tick " << i << " done." << endl << flush;
  }
  cout << "done." << endl << flush;
}

#endif /* SRC_PARTYSCENARIO_BASELINE_H_ */
```

E Party scenario (Jason)

Listing E.1: Agent logic for the party scenario

```
/* Initial beliefs and rules */

/* Initial goals */

!getPresent.

/* Plans */

+!getPresent : true <-
  !getMoney;
  walkToShop;
  purchasePresent;
  !!getPresent.

+!getMoney : true <-
  walkToATM;
  withdrawCash.

+!getMoney : true <-
  walkToFriend;
  collectMoney.
```