Doan, Thu Trang (2014) Meta-APL: a general language for agent programming. PhD thesis, University of Nottingham.

Meta-APL

A general language

for agent programming

by Thu Trang Doan, MSc

Thesis submitted to The University of Nottingham

for the degree of Doctor of Philosophy, December 2013

The University of Nottingham

UNITED KINGDOM · CHINA · MALAYSIA

Abstract

A key advantage of BDI-based agent programming is that agents can deliberate about which course of action to adopt to achieve a goal or respond to an event. However while state-of-the-art BDI-based agent programming languages provide flexible support for expressing plans, they are typically limited to a single, hard-coded, deliberation strategy (perhaps with some parameterisation) for all task environments. In this thesis, we describe a novel agent programming language, **meta-APL**, that allows both agent programs and the agent's deliberation strategy to be encoded in the same programming language. Key steps in the execution cycle of **meta-APL** are reflected in the state of the agent and can be queried and updated by **meta-APL** rules, allowing a wide range of BDI deliberation strategies to be programmed. We give the syntax and the operational semantics of **meta-APL**, focussing on the connections between the agent's state and its implementation. Finally, to illustrate the flexibility of **meta-APL**, we show how Jason and 3APL programs and deliberation strategy can be translated into **meta-APL** to give equivalent behaviour under weak bisimulation equivalence.

Acknowledgements

Above all, I am extremely grateful to Dr. Brian Logan, my principal supervisor as well as my primary resource for my study, for his scientific advice and knowledge, and insightful discussion and suggestion. This thesis would not exist at all if it were not for his tireless help and encouragement. I would also like to express my heartfelt gratitude to Dr. Natasha Alechina for not only encouraging and constructive feedback, thoughtful and detailed comments but also her constant support in my study as well as my life. She is someone everybody will instantly love and never forget once they meet her.

I would like to send my thanks to Dr. Henrik Nilsson and Dr. John-Jules Ch. Meyer for their valuable comments and suggestions on my thesis.

I would also like to thank the University of Nottingham and the School of Computer Science for giving me the opportunity to carry out my research, for their financial support and for providing a warm and inviting place to study.

Special thanks to my colleagues in Agent Lab, my wonderful friends in Nottingham and Swansea, for your invaluable help and support during my last five years.

Last but not least, I would like to thank my family, especially Hoàng Ngà Nguyen, my best friend, my soul-mate, my husband, for your unconditional and endless love, genuine care and deep sympathy at any time and anywhere I need. Without you, I am just a dust in the wind!

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

Agent technology offers a promising approach to the development of large systems consisting of distributed, intelligent agents who interact via message passing and/or by performing actions in a shared environment. The Belief-Desire-Intention (BDI) approach to designing, programming and verifying such systems has been very successful, and is perhaps now the dominant paradigm in the field of multi-agent systems. In this architecture, states of agents are comprised of mental attitudes such as beliefs, desires and intentions where beliefs are the agents' description about the world, desires specify states of affairs about which the agents want to bring and intentions are some of the desires to which the agents are committed. Agents use these mental attitudes to deliberate and decide which actions to perform.

In agent programming languages based on the BDI approach, agents select plans in response to changes in their environment or to achieve goals. In most of these languages, plan selection follows four steps. First the set of relevant plans is determined. A plan is relevant if its triggering condition matches a goal to be achieved or a change in the agent's beliefs the agent should respond to. Second, the set of applicable plans is determined. A

plan is applicable if its belief context evaluates to true, given the agent's current beliefs. Third, the agent commits to (intends) one or more of its relevant, applicable plans. Finally, from this updated set of intentions, the agent then selects one or more intentions, and executes one (or more) steps of the plan for that intention. This deliberation process then repeats at the next cycle of agent execution.

Current BDI-based agent programming languages such as AgentSpeak(L) [Rao, 1996], 3APL [Hindriks et al., 1999; Dastani et al., 2003b], GOAL [Hindriks et al., 2000; de Boer et al., 2007], Dribble [van Riemsdijk et al., 2003], Jason [Bordini et al., 2007], and 2APL [Dastani, 2008] provide considerable support for steps one and two (determining relevant, applicable plans). A programmer can write rule-based expressions in these languages which include triggering conditions and belief contexts. For example, a *plan* in AgentSpeak(L) contains a triggering event used to determine if the plan is relevant and a context query used to determine if the plan is applicable.

However, with the exception of some flags for plan in Jason, the third and fourth steps (adopting intentions and selecting one or more intentions for execution) cannot be programmed in the agent programming language itself. For example, in Jason, a plan can be accompanied with flags such as the flag **atomic** meaning that if an intention generated by the plan is selected for execution, the intention will be selected for execution in the subsequent deliberation cycles until completed. In other words, a Jason agent will not select another intention for execution if the intention which was selected to be executed in the last cycle is accompanied with the flag atomic has not been executed completely. No single deliberation strategy is clearly "best" for all agent task environments. For example, in a deliberation strategy, intentions can be executed in an interleaving or non-interleaving fashion. Interleaving can improve the performance of agents as they can achieve more than one goals in parallel. However, it might also give rise to the contention of resource so that no intentions were executed unsuccessfully [Thangarajah et al., 2003]. For example, an agent has two intentions for two goals about being at two different locations which are

in opposite directions. Then, interleaving the execution of these two intentions will make the agent to go around a position between the two locations. It is therefore important that the agent developer has the freedom to adopt the strategy which is most appropriate to a particular problem.

Some languages allow the programmer to override the default deliberation cycle behaviour by redefining "selection functions". For example, in Jason, there are three selection functions for selecting an event to react to, an applicable plan to generate intentions and an intention for execution. However, the redefinition of these selection functions cannot be done in the agent programming languages since they do not provide any support to do so. Therefore, the redefinitions of these selection functions must be done in the host language (i.e., the language in which the interpreters of these languages are themselves implemented). A different approach to modify the default deliberation cycle is to specify the deliberation strategy in a different language such as [Dastani et al., 2003a]. This language provides primitive meta-statements such as for selecting a goal planning rule and for generating a plan from a selected rule. Then, a deliberation cycle is defined by combining these meta-statements using sequential, conditional and iterative constructs.

Clearly, both redefining selection functions in host languages and specifying deliberation strategies in a different language are less than ideal. If often requires considerable knowledge of how the deliberation cycle is implemented in the host language, for example. Moreover, without reading additional code (usually written in a different language), an agent developer cannot tell how a program will be executed. Therefore, it would be better to be able to program steps three and four in the agent programming language itself. In other words, this agent programming language must provide facilities to query and manipulate its first class objects (where intentions are obviously first class objects) so that strategies of adopting new intentions and selection existing intentions for execution can be encoded in the language.

One way to achieve such a meta agent programming language is to use procedural re-

flection. A reflective programming language [des Rivières and Smith, 1984] incorporates a model of (aspects of) the language's implementation and state of program execution in the language itself, and provides facilities to manipulate this representation. Critically, changes in the underlying implementation are reflected in the model, and manipulation of the representation by a program results in changes in the underlying implementation and execution of the program. Perhaps the best known reflective programming language is 3-Lisp [Smith, 1984]. However, many agent programming languages also provide some degree of support for procedural reflection. For example, the Procedural Reasoning System (PRS) [Georgeff and Lansky, 1987] incorporated a meta-level, including reflection of some aspects of the state of the execution of the agent such as the set of applicable plans, allowing a developer to program deliberation about the choice of plans in the language itself. Similarly, many agent programming languages (such as Jason) provide facilities to manipulate the set of intentions. However, the support for procedural reflection in current state-of-the-art agent programming languages is always partial, in the sense that it is difficult to express the deliberation strategy of the agent directly in the agent programming language.

In this thesis, we show how procedural reflection can be applied in a BDI-based agent programming language to allow a straightforward implementation of steps three and four in the deliberation cycle of a BDI agent. By exploiting procedural reflection an agent programmer can customise the deliberation cycle to control which relevant applicable plan(s) to intend, and which intention(s) to execute. We argue this brings the advantages of the BDI approach to the problem of selecting an appropriate deliberation strategy given the agent's state, and moreover, facilitates a modular, incremental approach to the development of deliberation strategies.

We describe a novel agent programming language, **meta-APL**, that allows both agent programs and the agent's deliberation strategy to be encoded in the same programming language. This is achieved by adding the ability to query the agent's plan state and actions

which manipulate the plan state to the language. Currently, no interpreter has been implemented for **meta-APL**. There are three key goals underlying the design of **meta-APL**:

- it should be possible to specify a wide range of deliberation cycles, including the deliberation cycles of current, state-of-the art agent programming languages;
- programs of other agent programming languages should be translatable into **meta-APL** in a modular fashion, namely the program of an agent programming language should be translated to an "object program" of **meta-APL** and the same specification of the agent programming language's deliberation cycle should be used with all such translations (resulting in a bisimilar execution); and
- programming at the object level in this language should be at least as simple and easy as in state-of-the-art BDI agent programming languages; however, agent programmers can take advantage of specifying alternative deliberation strategies in this language.

In order to show that programs of other agent programming languages can be translated into **meta-APL**, we choose to simulate agent programs in Jason and 3APL. Although both Jason and 3APL are based on the BDI approach, they support different features. For example, agents in Jason generate intentions to react to events and do not repair intentions, whereas agents in 3APL generate intentions to achieve declarative goals. 3APL agents are also capable of repairing intentions. These differences exhibit different challenges when translating their agent programs into **meta-APL**. Finally, both Jason and 3APL agents are provided with formal semantics as transition systems. Therefore, the equivalence between these transition systems can be identified using the concept of weak bisimulation.

We believe the ability to express deliberation strategies (and other language features) in a clear, transparent and modular way provides a flexible tool for agent design. By expressing a deliberation strategy in **meta-APL**, we provide a precise, declarative operational semantics for the strategy which does not rely on user-specified functions. Key steps

in the execution cycle of **meta-APL** are reflected in the state of the agent and can be queried and updated by **meta-APL** rules, allowing a wide range of BDI deliberation strategies to be programmed. Furthermore, even low level implementation details of a strategy, such as the order in which rules are fired, or intentions executed can be expressed if necessary.

1.2 Research objectives and contributions

The main research objectives of this thesis are:

- To design an agent programming language (**meta-APL**) with facilities that enable the encoding of both agent programs and agent deliberation strategies;
- To define the operational semantics of **meta-APL**;
- To show how **meta-APL** encodes typical deliberation strategies;
- To illustrate the flexibility of **meta-APL** by simulating state-of-the-art BDI-based agent programming languages;
- A theory to support the correctness proof of simulating in **meta-APL** under weak bisimulation equivalence;
- The correctness proofs of these simulations under weak bisimulation equivalence.

The main contributions of this thesis are:

- A brief survey of existing agent programming languages; the limitations of these languages with respect to implementing deliberation strategies are also discussed;
- An overview of procedural reflection as an approach to enable programming deliberation strategies in agent programming languages;
- A summary of existing results on simulating agent programming languages;

- An agent programming language, namely **meta-APL**, with formal operational semantics which supports procedural reflection so that different deliberations strategies can be implemented; the encoding of typical deliberation strategies in **meta-APL** is also discussed;
- Identifying a general theory for establishing a weak bisimulation between two agent programs through the strong bisimulation of their deliberation cycles;
- Developing the simulation of two state-of-the-art agent programming languages Jason and 3APL in **meta-APL**; the equivalence result for each simulation is proved.

1.3 Overview and structure of Thesis

The rest of this thesis is organised as follows:

Chapter 2 is a review of the background literature. Here, we recall the notion of intelligent agents, the BDI approach to designing, specifying and programming agents, BDI-based agent programming languages and the concept of procedural reflection in programming languages.

Chapter 3 defines the syntax of **meta-APL**. **Meta-APL** includes constructs for mental attitudes such as beliefs, goals, plans, intentions, facilities for enabling procedural reflection such as querying and manipulating mental attitudes.

Chapter 4 defines the operational semantics of **meta-APL**. Here, we give the definition of agent configurations in **meta-APL**, how agents run by transiting from one configuration to another via transition rules. We also define semantics of the facilities which enable procedural reflection in **meta-APL**.

Chapter 5 develops a theory for proving the correctness of simulating other agent programming languages in **meta-APL**. Here, the correctness is based on weak bisimu-

lations which can be identified by showing the equivalence of operations between deliberation cycles only.

Chapter 6 presents the simulation of Jason in **meta-APL**. We define a translation function to convert a Jason agent program into a **meta-APL** agent program. Then, we show that these two agents operate equivalently under weak bisimulation.

Chapter 7 presents the simulation of 3APL in **meta-APL**. 3APL supports several features not found in Jason such as declarative goals and plan revision. Here we also define a translation function to convert each 3APL agent program into a **meta-APL** agent program and show the equivalence between these two programs.

Chapter 8 gives a brief conclusion about the work and results of this thesis. It also provides directions for future work.

Chapter 2

Background: Agents and programming agents

The notion of an intelligent agent is the central to this thesis. In this chapter, we attempt to answer a number of questions regarding agents including what intelligent agents are, how to specify them, and how to build them by drawing on the multi-agent systems literature. In particular, we review the definition of intelligent agents, the BDI architecture and give a short overview of agent programming languages. At the end of this chapter, we present a brief review of work related to procedural reflection.

2.1 Intelligent agents

In order to develop an agent programming language, it is important to understand the concept of intelligent agents. Here, we follow Woodridge and Jenning's definition of intelligent agents [Wooldridge and Jennings, 1995; Wooldridge, 2002]:

> *An agent is a computer system that is situated in some environment, and that is capable of autonomous action in this environment in order to meet its design objectives.*

This definition provides an abstract view of an intelligent agent, as illustrated in Figure 2.1, where the agent is situated in an environment and autonomously runs in order to achieve its goals. Being situated in an environment means that the agent can receive sensory information provided by equipment. For example, a mobile agent is equipped with a camera in order to detect if there is an obstacle in front of the agent. The agent is also provided with a repertoire of certain external actions which are used to change the state of the environment. For example, a mobile agent may have effectors such as motorised wheels which allows the agent to move forward; as the agent moves to a different place, the state of the environment changes. In many environments, external actions may fail.

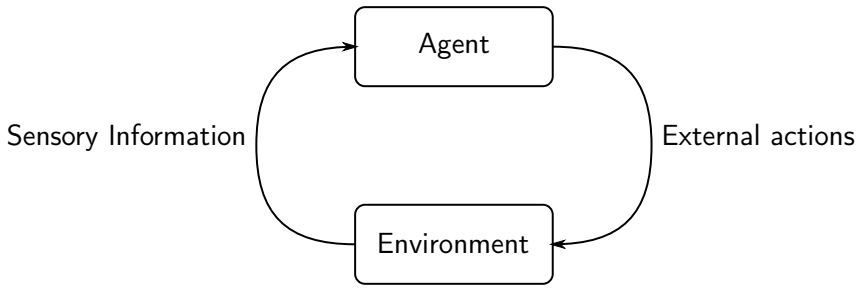

Figure 2.1: An abstract view of intelligent agents.

Therefore, we assume that the execution of an external action may fail to have its desired effect.

Additionally, agents are also argued to have further properties [Wooldridge and Jennings, 1995]:

Autonomy: Intelligent agents have the ability to operate autonomously. Here, autonomy means that an intelligent agent determines which action to perform and the freedom to decide how to satisfy its design objectives.

Pro-activeness: Intelligent agents are capable of exhibiting goal-directed behaviours. This means that if an agent has an intention to achieve a goal, the agent will try to exploit any opportunity to complete that intention.

Reactivity: Intelligent agents are capable of reacting to changes in the state of their environment which are recognised by perception. Reactivity requires the agents to react to these changes in a timely fashion.

Social ability: Intelligent agents are provided with communication languages so that they can communicate with each other (and possibly their owners). Social ability allows the agents to exchange information, to negotiate and to cooperate.

A key challenge confronting researchers in the field of multi-agent systems is to engineer intelligent agents that can achieve their design objectives. In the following sections, we briefly review some of the approaches in building and implementing intelligent agents.

2.2 BDI architecture

One of most common approaches to designing intelligent agents is to use the Belief-Desire-Intention (BDI) architecture. In this architecture, states of agents consist of mental attitudes including beliefs, desires and intentions. An agent uses these mental attitudes when deliberating and deciding which actions to perform. Intuitively, beliefs are the information which agents have about the world. These beliefs may be incomplete or incorrect. In contrast to beliefs, the desires of an agent are descriptions of states of affairs that the agent wants to bring about. In general, it may not be possible to achieve all agent's desires since desires may be in conflict and the agent might not have enough resources to achieve all desires. Intentions are the particular desires which an agent is committed to achieving.

Agents can apply different strategies to commit to intentions. There are three commonly used commitment strategies in the literature [Rao and Georgeff, 1991]:

- *Blind commitment:* agents with blind commitment are those who, once committed to an intention, will maintain this intention as long as the agents believe that the intention is not achieved;

- *Single minded commitment:* agents with single minded commitment are those who, once committed to an intention, will maintain this intention as long as the agents believe that the intention is not achieved and it is possible to achieve the intention; and

- *Open minded commitment:* agents with open minded commitment are those who, once committed to an intention for achieving a goal, will maintain this intention as long as the agents believe that the intention is not achieved and the intention is still one of agent's goals.

Among the above commitment strategies, blind commitment is the most conservative: agents with this strategy will keep an intention as long as the intention is not achieved, i.e., if the intention is never achieved, it will be kept forever. Single mined commitment relaxes this condition by allowing intention can be dropped as soon as the agent knows that it is not possible to achieve the intention. Open mined commitment relaxes the condition to keep intentions further, by allowing an agent to drop an intention as soon as the intention is not a goal.

The BDI architecture has been used as a basis for defining logical formalisms which allow the design of intelligent agents and systems of multiple agents to be specified. These logics are usually combinations of epistemic logics and dynamic logics or temporal logics where different modalities are used to describe mental attitudes of agents such as beliefs, goals (for desires) and intentions and possible worlds are used to define semantics of formulas. For example, Cohen and Levesque's logic, [1990], is a combination of epistemic and dynamic logics. In this logic, beliefs and goals are specified by modal operators BEL and GOAL, respectively. However, intentions are described by dynamic formulas containing action expressions and formulas about beliefs and goals. Another example is Rao and Georgeff's logic, [1991], which adopts an alternative approach to possible world semantics where epistemic logics are combined with computation tree logic (CTL*). Furthermore, intentions are treated as first-class citizens apart from beliefs and goals; i.e., there is an

operator, namely INTEND, which is used to specify intentions of agents.

The BDI architecture has also been used as a basis for defining agent programming languages that allow intelligent agents and systems of multiple agents to be implemented. These languages provide data structures to encode beliefs, goals and intentions. Examples of agent programming languages based on the BDI architecture include: PRS [Georgeff and Lansky, 1987], AgentSpeak(L) [Rao, 1996], 3APL [Hindriks et al., 1999; Dastani et al., 2003b], GOAL [Hindriks et al., 2000; de Boer et al., 2007], Dribble [van Riemsdijk et al., 2003], Jason [Bordini et al., 2007], and 2APL [Dastani, 2008]. Interpreters of these languages are generally implemented as loops which include activities such as receiving percepts from the environment, generating new intentions, selecting an intention and executing it, and dropping intentions. Generating new intentions may include two steps: first to select some desires to be achieved, then to generate intentions in order to achieve these selected desires. The selection of intentions for execution can be implemented using two common strategies: interleaving (which allows existing intentions to be executed in parallel in an interleaving fashion) and non-interleaving (which keeps selecting an intention until it is completed or dropped). Then, the selection of intentions to be dropped are implemented according to some commitment strategy such as blinded commitment, single minded commitment or open minded commitment). Such a loop is called a deliberation cycle which is an idealisation of components, namely generating intentions, deliberating, executing and handling intentions, in practical reasoning [Bratman, 1999].

2.3 Agent programming languages

In this section, we present a brief survey of BDI-based languages. Our aim is to illustrate the broad spectrum of such languages and the design ideas which underlie them. Here, agent programming languages are listed according to categories, namely logic-based languages, imperative languages and hybrid languages, as presented in [Bordini et al., 2006].

2.3.1 Logic-based languages

Logic-based agent programming languages allow the direct specifications of intelligent agents and systems of multiple agents to be described in a logic language. Typical examples of these languages include METATEM [Gabbay, 1987; Barringer et al., 1989, 1995; Fisher, 2005], Golog [Levesque et al., 1997], MINERVA [Leite et al., 2001], EVOLP [Alferes et al., 2002], and FLUX [Thielscher, 2005]. The specification of an intelligent agent in such a language is executed by means of its interpreter, which attempts to construct a formal model of the specification. In the remainder of this section, we review two well-known logic-based languages: METATEM and Golog.

METATEM

Agent programs in METATEM [Gabbay, 1987; Barringer et al., 1989, 1995; Fisher, 2005] are specifications which consist of declarative statements. These statements specify the behaviour that agents are expected to exhibit. Each statement is a Linear Temporal Logic (LTL) [Pnueli, 1977] formula where Linear temporal logic LTL is an extension of the propositional logic with temporal operators including:

- the *next* operator: $\bigcirc\varphi$ means that φ is satisfied at the next moment in time;
- the *some* operator: $\diamond\varphi$ means that φ is satisfied at some future moment in time;
- the *all* operator: $\Box\varphi$ means that φ is satisfied at all future moments in time; and
- the *until* operator: $\varphi U \psi$ means that φ is satisfied at all future moments in time until ψ is satisfied.

LTL allows properties of an individual agent to be expressed concisely. For example, the formula $request \rightarrow reply\ U\ ack$ describes an agent which will reply continuously to another one whenever it receives a request from that agent until it obtains an acknowledgement message.

A METATEM agent is executed by means of iteratively constructing a temporal model for the agent's specification. This is equivalent to proving that the specification is satisfiable. The process of constructing temporal models of the specification is facilitated by translating the statements comprising the specification into equivalent statements in Separate Normal Form. In this form, temporal statements are conjunctions of formulas of three forms:

- initial rules which specify initial states of the agent, for example

  $$\textbf{start} \Rightarrow (sad \vee optimistic)$$

  states that at the beginning, either sad or $optimistic$ is true. Intuitively, this means the agent is either sad or optimistic at the beginning.

- step rules which specify the relation between consecutive states of the agent, for example

  $$sad \wedge \neg optimistic \Rightarrow \bigcirc(sad)$$

  states that from a state where the agent is sad and not optimistic, in the next state the agent must remain sad.

- sometime rules which specify constraints to be satisfied by executions, for example

  $$optimistic \Rightarrow \Diamond(\neg sad)$$

  states that from any state where the agent is optimistic, any execution of the agent will eventually lead to a state where the agent is not sad.

Golog

Golog [Levesque et al., 1997] is a logic programming language based on Situation Calculus. The program of a Golog agent consists of a number of axioms specifying the preconditions and the effects (post-conditions) of actions, and a strategy for executing the agent.

In general, a run of a Golog agent starts in an initial situation and moves to another situation which is produced by applying an applicable (primitive) action at the previous situation. The state of the world is described by fluents whose truth values may vary from situation to situation (hence, a fluent has an argument for situations) together with other predicates whose truth values do not depend on situations. The applicability of an action is provided by a precondition axiom. For example, an action to move north is possible at a situation s if and only if there is no obstacle in the north direction in this situation. This can be expressed in the following First Order Logic (FOL) formula:

$$Poss(move(north), s) \equiv$$
$$\forall cl, nl(location(cl, s) \wedge next(cl, north, nl) \rightarrow \neg obstacle(nl, s))$$

Recall that each fluent in the above FOL formula has an situation argument s. When all situation arguments of fluents in a FOL formula ϕ are suppressed, ϕ is called a pseudo-fluent logical formula and $\varphi(s)$ denotes the formula obtained from ϕ by adding a situation argument s to every fluent in ϕ.

The axioms for the effects of actions show how actions change the truth values of fluents. For each fluent, programmers provide an axiom to indicate in which situations it is true or false and by which actions it becomes true or false. For example, the truth value of the fluent $location$ can be defined as follows:

$$\begin{aligned} Poss(a, s) \supset \quad & [location(l, do(a, s)) \equiv \\ & (\exists cl, d.location(cl, s) \wedge a = move(d) \wedge next(cl, d, l)) \vee \\ & (\neg \exists d.a = move(d) \wedge location(l, s))] \end{aligned}$$

The set of programmer-defined axioms together with the foundational ones including a set of axioms for unique names and a set of axioms from situation calculus is called $Axioms$.

The strategy in a Golog agent program is defined inductively from primitive actions as follows:

- if a is a primitive action, then a is a complex action;

- if ϕ is a pseudo-fluent logical formula , then the test action $\phi?$ is a complex action;
- if δ, δ_1 and δ_2 are complex actions, so is the sequence of actions $\delta_1; \delta_2$;
- if δ, δ_1 and δ_2 are complex actions, so is the non-deterministic choice of actions $\delta_1|\delta_2$;
- if $\delta(x)$ is a complex action, so is the non-deterministic choice of action arguments $(\pi x)\delta(x)$; and
- if δ, δ_1 and δ_2 are complex actions, so is the non-deterministic iteration of actions δ^*.

The meaning of a strategy is defined inductively through a macro Do as follows:

- $Do(a, s, t) \equiv Poss(a, s) \wedge t = do(a, s)$
- $Do(\phi?, s, t) \equiv \phi(s) \wedge s = t$
- $Do(\delta_1; \delta_2, s, t) \equiv \exists s'.Do(\delta_1, s, s') \wedge Do(\delta_2, s', t)$
- $Do(\delta_1|\delta_2, s, t) \equiv Do(\delta_1, s, t) \vee Do(\delta_2, s', t)$
- $Do((\pi x)\delta(x), s, t) \equiv \exists x.Do(\delta(x), s, t)$
- $Do(\delta^*, s, t) \equiv \forall P.\{(\forall s_1.P(s_1, s_1)) \wedge (\forall s_1, s_2, s_3.P(s_1, s_2) \wedge Do(\delta, s_2, s_3) \supset P(s_1, s_3))\} \supset P(s, t)$

The last equivalence defines the semantics of a non-deterministic iteration in which t is reached from s by performing the complex action δ zero or more times.

Running an agent which is defined by $Axioms$ and a strategy δ from an initial situation S_0 is to use a theorem prover to perform the following task:

$$Axioms \models \exists s.Do(\delta, S_0, s)$$

2.3.2 Imperative languages

Agents can also be implemented using conventional (imperative) programming languages, such as Pascal, Java and C++. In such a language, a programmer must perform tasks including defining data structures to represent agent's beliefs, goals and intentions as well as functions to decide which action to execute. Therefore, several extensions of imperative programming languages have been proposed such as JAL [Winikoff, 2005], JADE [Bellifemine et al., 2005] and Jadex [Pokahr et al., 2005] which facilitate the development of agents and multi-agent systems. As Jadex is an extension of JADE, we only review JAL and JADE in the following.

JAL

JAL [Winikoff, 2005] extends Java with a number of language constructs to allow agent programmers to define a range of notions including:

- Beliefset: beliefs in JAL are stored in a small relational database which is a set of beliefsets. Each beliefset, or relation, stores beliefs of the form $name(data_1, data_2, ...)$ which have the same name and different values for data fields.
- View: virtual beliefsets that are generated from other beliefsets.
- Event: events which indicate some changes, such as the receipt of messages, the adoption of a new goal, the receipt of perception from from the environment.
- Plan: a plan includes indications of which event it handles, which events it generates, a context to determine in which situations it can be used, and a body that contains Java code.

A JAL agent consists of beliefsets that the agent has, events that it handles and posts, and plans that it has to deal with events. A plan has a method, namely *relevant*, which takes an event as argument and returns true if the plan is relevant to deal with the event.

The relevant method can only access data contained in the event. A plan also has another method, namely *context*, which returns true if the plan is applicable. Finally, we define a body of the plan which includes only Java code.

A JAL agent runs by repeatedly handling events. The process of handling events is as follows:

1. Event posted;
2. Find plans that handle it;
3. Determine the relevant plans;
4. Determine the applicable plans;
5. Select a plan and run its body;
6. If plan fails, go to step 4.

As JAL is based heavily on Java, it has no formal semantics.

JADE

JADE [Bellifemine et al., 2005] is also based on the programming language Java. However, rather than an agent programming language, JADE is a framework for building multi-agent systems. JADE defines a set of Java classes for defining an agent such as Agent and Behaviour. Realising an agent involves constructing a class extending the Agent class. The internal data of agents built in JADE such as beliefs and goals are defined as variables of the class and behaviours of the agents are inner classes which inherit from the Behaviour class defined in JADE. Implementing a behaviour for an agent involves defining an *action* method and optionally another *done* method to indicate whether the behaviour has completed its execution. Like JAL, JADE also has no formal semantics.

2.3.3 Hybrid languages

Hybrid languages are combinations of logical languages and imperative ones. For most of the languages in this category, mental notions such as beliefs and goals are represented and queried by means of logical languages, while intentions are expressed in imperative languages by using conventional programming constructs including sequential statements, conditional statements and iteration statements. In this section, we review some popular hybrid agent programming languages including PRS, SPARK, AgentSpeak(L), Jason, 3APL, GOAL and Gwendolen.

PRS

A PRS [Georgeff and Lansky, 1987] agent contains a database for storing its beliefs, a set of goals, an ACT library and a set of intentions.

Given a set of predicate symbols, a set of function symbols and a set of variables, the database is a set of logical formulas P which have the following syntax:

$$\begin{aligned} P &::= (p\ t^*) \mid (NOT\ P) \mid (AND\ P^+) \\ t &::= x \mid (f\ t^*) \end{aligned}$$

where p is an element of the set of predicate symbols, f is an element of the set of function symbols, and x is a variable from the set of variable. PRS has procedures to maintain the consistency of sets of ground literals.

A goal describes a desired behaviour that an agent intends to perform. It can be one of the following forms:

- *ACHIEVE* C, to obtain C;
- *ACHIEVE-BY* C $(A1, \ldots, An)$, to obtain C by some procedure $A1, \ldots, An$ from the ACT library (see below);
- *TEST* C, to test the condition C

- *USE-RESOURCE* R to allocate a resource R which is either a name or a variable;
- *WAIT-UNTIL* C, to wait until C becomes true;
- *REQUIRE-UNTIL* G C, to check that a goal G is always true until C becomes true;
- *CONCLUDE* P, to add P into the database;
- *RETRACT* P, to delete P from the database;

where C is a well formed formula which has the following syntax:

$$C \quad ::= \quad P \mid (OR\ C^{+})$$

The ACT library is a set of procedure specifications, called *Knowledge Area* (KA), which show how to accomplish a goal or to respond to a particular situation. Each KA consists of a body (describing the steps to be performed of the KA), and an *invocation condition* (specifying when it is suitable to use the KA). The body (also called the plot) of a KA is described by a graph containing an initial node and one or more end nodes. Each node on the graph is labelled with a subgoal to be archived while carrying out the KA. The formalism for KAs supports complicated control constructs such as conditional choice, loops and recursion.

SPARK

SPARK[Morley and Myers, 2004] is a successor of PRS. It is a framework for building systems of agents supporting scalability and clean semantics. Compared to PRS, the language has made a number of simplifications and improvements. A SPARK agent's knowledge base is a set of ground literals rather than a set of restricted FOL formulas as in the database in PRS and the plot of a KA is extended with syntactical constructs for compound task expressions. A task can be a composition of other tasks in sequence or parallel, or can be constructed using familiar constructs from conventional imperative programming

languages such as *if – then – else*, *try – catch*, and *while – do*. SPARK also provides operational semantics for each construct so that programmers can understand the meaning of compound tasks.

AgentSpeak(L)

AgentSpeak(L) [Rao, 1996] is a simplified version of PRS. An AgentSpeak(L) agent consists of a belief base, an event base, a plan library (or the plan base) and an intention base. The belief base consists of ground literals. The event base consists of events that occur during the execution of the agent. Events specify changes in mental states of agents such as the addition and deletion of beliefs and goals. There are two types of goals: achievement goals and test goals. An achievement goal specifies a state of affairs that the agent would like to bring about whereas a test goal simply is for checking whether a predicate is true with respect to the current belief base of the agent. The plan library consists of plans used to generate intentions. Each plan is comprised of a triggering event, a context query and a body as a sequence of external actions, belief update actions, and subgoal actions (achievement goal or test goal). Finally, the intention base consists of intentions that the agent is committed to.

AgentSpeak(L) is provided with a formal operational semantics which is defined by a set of transition rules. For example, there is a transition rule describing the application of a plan when its triggering event matches an event in the event base and its query evaluates to true with respect to the belief base. The result of applying the plan is that its body is inserted into the intention base. In AgentSpeak(L), when an intention is created, the agent will not drop the intention until it completes. There are also transition rules for executing actions of intentions in the intention base. Here, executing an external action means to perform the action on the environment. Executing a belief update action is to add or delete beliefs to/from the belief base. Executing an achievement goal means to generate an internal event into the event base. Executing a test goal means to evaluate it

against the belief base.

Jason

Jason [Bordini et al., 2007] is an extension of AgentSpeak(L). Details about AgentSpeak(L) and Jason are described in detail in Chapter 6 where we show how to simulate Jason agents in **meta-APL**.

3APL

3APL was first proposed by [Hindriks et al., 1999]. A 3APL agent consists of beliefs and procedural goals (i.e., intentions) which describe what to do. In later work, 3APL was extended in [Dastani et al., 2003b] to incorporate the separation between procedural goals and declarative goals (where a declarative goal describes what to achieve) and the ability to revise intentions. In this extended language, a 3APL agent program consists of a belief base, similar to a Prolog program, consisting of facts and Horn clauses, a goal base consisting of goals as conjunctions of literals, and a rule base consisting of goal planning rules for generating new intentions and plan revision rules for revising existing intentions.

The execution of 3APL agents is defined by the 3APL operational semantics comprised of transition rules. These transition rules define how to apply goal planning rules and plan revision rules, and how to execute intentions. 3APL is also provided with an interpreter, namely the 3APL Platform [Dastani et al., 2005]. 3APL is described in detail in Chapter 7 where we also show how to simulate 3APL agents in **meta-APL**.

GOAL

GOAL was proposed in [Hindriks et al., 2000] to support declarative goals which were missing in AgentSpeak(L) [Rao, 1996] and the first version of 3APL [Hindriks et al., 1999].

A GOAL agent consists of a set of actions, an initial belief base and an initial goal base. It reasons about beliefs and goals based on Propositional Logic (PL). In the following, we

use PL to denote the set of all formulas of PL and $\models$ to denote the logical consequence relation in PL.

Given a set of propositional variables $Prop$, a belief ϕ or goal ψ in GOAL is a formula of PL and has the following syntax:

$$\phi, \psi ::= p \mid \neg\phi \mid \phi_1 \wedge \phi_2$$

where $p \in Prop$.

In GOAL, queries are defined as follows:

$$\varphi ::= B\phi \mid G\psi \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2$$

A GOAL agent is equipped with a set of basic actions, namely $Bcap$, which are used to update the agent's beliefs. Basic actions in $Bcap$ are specified by a function $T : Bcap \times \wp(PL) \to \wp(PL)$ where $\wp(PL)$ is the set of all subsets of PL, i.e., $\wp(PL) = \{\Sigma \mid \Sigma \subseteq PL\}$. Then, given a basic action $a \in Bcap$ and a set of formulas $\Sigma \subseteq PL$ which represents the agent's beliefs, $T(a, \Sigma)$ is the resulting set of the agent's beliefs after executing the basic action a.

Furthermore, a GOAL agent can also perform two special basic actions $adopt(\psi)$ and $drop(\psi)$ which are used to add and to delete goals, respectively.

An action b in GOAL is defined by a query (specifying when the action is enabled) and a basic action (specifying the effect of the action). The syntax of a is as follows:

$$b ::= \varphi \to do(a)$$

where $a \in Bcap \cup \{adopt(\psi), drop(\psi) \mid \psi \in PL\}$. Here, $adopt(\psi)$ and $drop(\psi)$ are two predefined basic actions in GOAL where the semantics is to add to the agent's goals a new goal ψ and to delete from the agent's goals a goal ψ, respectively.

Then, a GOAL agent is a triple $(\Pi, \Sigma_0, \Gamma_0)$ where Π is a finite set of actions, Σ_0 is a finite set of beliefs and Γ_0 is a finite set of $goals$ such that for all $\psi \in \Gamma_0$, $\Sigma_0 \not\models \psi$. The condition $\Sigma_0 \not\models \psi$ on the goal set Γ_0 means that goals in GOAL agents are declarative; i.e.,

the agents do not maintain goals that have been achieved (those which are consequences of the agent's beliefs). During execution, this condition will be checked whenever there are changes in the agent's beliefs and goals and maintained by removing any goal that violates the condition.

A configuration of a GOAL agent is a pair (Σ, Γ) of the belief base Σ and the goal base Γ. A query φ is evaluated against the configuration (Σ, Γ) is defined as follows:

- $(\Sigma, \Gamma) \vdash B\phi$ iff $\Sigma \models \phi$;
- $(\Sigma, \Gamma) \vdash G\psi$ iff $\exists \psi' \in \Gamma : \psi' \models \psi$;
- $(\Sigma, \Gamma) \vdash \neg\varphi$ iff $(\Sigma, \Gamma) \nvdash \varphi$;
- $(\Sigma, \Gamma) \vdash \varphi_1 \wedge \varphi_2$ iff $(\Sigma, \Gamma) \vdash \varphi_1$ and $(\Sigma, \Gamma) \vdash \varphi_2$;

The GOAL operational semantics implements the blind commitment strategy, i.e., a GOAL agent drops a goal iff it achieves the goal. From a configuration (Σ, Γ), an action $b = \varphi \rightarrow do(a) \in \pi$ can be performed if the condition φ evaluates to true, i.e., $(\Sigma, \Gamma) \vdash \varphi$. Then, the transition of an action b is defined as follows:

- If $a \in Bcap$, the transition by b is defined as $(\Sigma, \Gamma) \xrightarrow{b} (\Sigma', \Gamma')$ where $\Sigma' = T(a, \Sigma)$ and $\Gamma' = \Gamma \setminus \{\phi \mid \Sigma' \models \phi\}$.
- If $a = adopt(\psi)$, the transition by b is defined as $(\Sigma, \Gamma) \xrightarrow{b} (\Sigma, \Gamma \cup \{\psi\})$ if $\Sigma \not\models \psi$ or $(\Sigma, \Gamma) \xrightarrow{b} (\Sigma, \Gamma)$ otherwise. This means that $adopt(\psi)$ only adds the goal ψ into Γ if ψ is not already achieved; i.e., is not a consequence of Σ.
- If $a = drop(\psi)$, the transition by b is defined as $(\Sigma, \Gamma) \xrightarrow{b} (\Sigma, \Gamma \setminus \{\psi' \mid \psi \models \psi'\})$. Note that $drop(false)$ does not drop any goal while $drop(true)$ removes all goals.

Gwendolen

Gwendolen [Dennis and Farwer, 2008] is a BDI-based agent programming language which shares many features with both AgentSpeak(L) and GOAL. An agent in Gwendolen con-

sists of:

- a belief base which is a finite set of literals;
- a goal base which is also a finite set of literals; and
- a plan base which is a finite set of plans for generating new intentions and repairing existing intentions.

Given an agent in Gwendolen, its deliberation cycles are defined by the operational semantics of Gwendolen which consists of two selection functions:

- S_{int} to select an intention to be executed at each deliberation cycle,
- S_{plan} to select a plan to apply at each deliberation cycle,

and transition rules which determine transitions in a deliberation cycle. These two functions S_{int} and S_{plan} are not implemented in Gwendolen but in the host language in which the Gwendolen interpreter is implemented. Each Gwendolen deliberation cycle is organised in six consecutive stages:

Stage A: which is responsible for selecting an intention to execute at the current cycle;

Stage B: which is responsible for generating applicable plans from the plan base with respect to existing beliefs, goals, and intentions;

Stage C: which is responsible for selecting an applicable plan to generate a new intention or repairing an existing intention;

Stage D: which is responsible for executing the top action of the selected intention in this cycle;

Stage E: which is responsible for updating the belief base and the goal base according to perception received from the environment;

Stage F: which is responsible for handling messages received from other agents.

2.3.4 Discussion

In this section, we briefly summarise the agent programming languages that have been presented so far. We will concentrate on the similarities as well as the differences between these agent programming languages in order to highlight common characteristics of BDI agent programming languages.

The main purpose of an agent programming language is to create a practical tool for expressing concepts in agent theories (such as the BDI-architecture). This characteristic is shared by most of agent programming languages that we have reviewed. For example, beliefs and desires are represented in METATEM in Propositional Logic. Similarly, Golog expresses these notions by using fluents in the situation calculus. The obvious advantage of using logical languages is that it facilitates representation of and reasoning with beliefs and desires by inheriting the decision procedure from the corresponding logical language. In contrast, agent programming languages such as JAL and JADE, that follow the imperative approach, do not use logic, and beliefs and desires must be expressed in an ad-hoc fashion. For example, JAL uses a small relational database to store beliefs of the form $name(data_1, data_2, \ldots)$ which are only equivalent to atoms in logics. However, imperative agent programming languages have their own advantages. In particular, it allows one to reuse legacy code implemented in imperative programming languages.

Furthermore, logic-based agent programming languages suffer from several signification drawbacks [Mascardi et al., 2003] such as low in efficiency, poor in scalability and lack of modularity. Furthermore, while these languages provide a highly expressive tool, they are not convenient and straightforward as implementation languages [Hindriks, 2001]; therefore, they are not likely to be used in mass production of multi-agent systems. Nevertheless, logic-based agent programming languages are suitable to build small and simple prototypes which are verifiable by rigorous methods such as model checking and theorem proving.

From this point of view, hybrid agent programming languages belong to a different

stream of agent programming language design that strives to combine the advantages of both logic-based and imperative agent programming languages. In particular, hybrid agent programming languages share the common characteristic that mental attitudes such as beliefs, events and goals are expressed in logical languages, while intentions are specified in an imperative manner. The connection between mental attitudes and intentions are usually object rules which specify when to adopt certain intentions. Here, the "when" conditions of these rules are determined through reasoning in the logics which are used to express the mental attitudes. Even when intentions are generated, they are linked to the mental information (such as events or goals) to justify the existence and purpose of the behaviours. For example, in Jason, the event that triggers a plan to generate an intention is kept as part of the intention (see Section 6 for details). Similarly, in 3APL, the declarative goal that is used to generate an intention is added as a part of the intention. This part is then used to justify the existence of the intention (see Section 7 for details). In particular, if the goal is achieved, the intention is deleted (even if it has not been executed completely).

Even though hybrid agent programming languages share the characteristics listed above, they differ greatly in terms of the mental attitudes that they support, the logical languages that are used, and the rules that generate and repair intentions. Figure 2.2 summarises these differences, where FOL* means a fragment of First Order Logic where only negation and conjunction operators are allowed; PL means propositional logic; and FOL** means a fragment of First Order Logic where only literals are allowed.

Furthermore, hybrid agent programming languages also differ in the support they provide for defining deliberation strategies to select intentions to adopt and execute. In the next section, we shall discuss these differences in more detail.

	Mental attitudes				Rules	
Language	Logic	Belief	Event	Goal	To generate intentions	To repair intentions
PRS/SPARK	FOL*	x		x	x	
AgentSpeak(L)/Jason	PROLOG	x	x		x	
3APL	PROLOG	x		x	x	x
GOAL	PL	x		x	x	
Gwendolen	FOL**	x		x	x	

Figure 2.2: Features of hybrid agent programming languages.

2.4 Programming deliberation strategies

In this section, we discuss the level of support that existing agent programming languages have for specifying a deliberation strategy for selecting intentions for adoption and execution.

2.4.1 Programming selection of intentions for adoption

In most hybrid agent programming languages, the programmer can specify which plan to select for adoption by means of rule-based constructs which include a condition to determine when they are applicable. For example, in PRS, the programmer uses KAs where the body of a KA specifies the plan and the invocation condition of the KA determines when it is suitable to use the plan in the KA. "Plans" in AgentSpeak(L) also allow the programmer to specify sequences of actions to adopt where the triggering event of a plan determines when it is relevant to consider the plan, and the context query of the plan determines when it is applicable to use the plan with respect to the beliefs of the agent. Goal planning rules in 3APL and actions in GOAL have a similar purpose.

However, with the exception of PRS and SPARK, most of these languages do not pro-

vide direct support for programming a strategy to select among multiple applicable plans in the languages. Instead, such a strategy is specified by means of selection functions which do not belong to the syntax of these agent programming languages and are defined the host languages which are used to implements their interpreters. For example, the selection function to select an applicable plan in Jason is implemented in Java, the host language which implement the Jason interpreter. The only way to alter this selection function is to reprogram it in Java and recompile the Jason interpreter.

PRS is an exception as it allows the programmer to define strategies to select multiple applicable KAs. Here, the programmer can use so-called *metalevel* beliefs for storing meta information about object-level KAs such as their estimated costs in time and money, their priorities, and their success rate to realise a goal. Furthermore, she can define *metalevel* KAs for reasoning about the level of importance and utility of object-level KAs so that the "best" KA can be determined and selected.

2.4.2 Programming selection of intentions for execution

Agent programming languages provide even less support for programming strategies to select intentions for execution. The interpreters of these languages usually implement a default strategy for selecting a plan to execute. For example, Jason by default implements a selection strategy [Bordini et al., 2007] based on a "round-robin" scheduler. A different strategy can be implemented by over-riding this default selection function and recompiling the Jason interpreter. Additionally, the programmer can only change the default selection for execution slightly by setting the *atomic* flag for plans. For an intention generated by a plan declared with the *atomic* flag, once it is selected for execution, it will be selected for execution in subsequent cycles until completed. However, this feature of Jason only leaves the agent programmer with a little space to change the default selection strategy.

A different approach, presented in [Hindriks et al., 1998; Dastani et al., 2003a], is to

define a meta language for programming deliberation strategies. In such a language, the programmer is provided with terms to define information about plans such as costs and meta-statements for selecting plans, applying plans and executing a plan. This meta-statements then can be combined using sequential, conditional and iterative constructs to define a deliberation program. The language defined in [Hindriks et al., 1998] was used to program deliberation cycles of AgentSpeak(L) and 3APL [Hindriks et al., 1999]. The language defined in [Dastani et al., 2003a] was used to program deliberation cycles of 3APL [Hindriks et al., 1999]. Obviously, this approach requires a language (in addition to agent programming languages) to define deliberation strategies.

No single deliberation strategy is clearly "best" for all agent task environments. For example, in a deliberation strategy, intentions can be executed in an interleaving or non-interleaving fashion. Interleaving can improve the performance of agents as they can achieve more than one goals in parallel. However, it can also give rise to contention for resources so that no intentions are executed successfully [Thangarajah et al., 2003]. For example, an agent has two intentions that achieve two goals to be at two different locations which are in opposite directions. Then, interleaving the execution of these two intentions makes the agent to go around a position between the two locations. It is therefore important that the agent developer has the freedom to adopt the strategy which is most appropriate to a particular problem. To this end, it is worth looking back at the discussion in [Hayes, 1977] about designing languages to program knowledge reasoners Hayes argues it is a good engineering practice that the language to represent knowledge and the language to specify strategies to reason about knowledge are the same. Similarly, agents can use their own knowledge (beliefs, goals, plans, intentions, etc.) to reason about which new plan to adopt, which existing intention to execute during deliberation. Therefore, it is also a good engineering practice if the language to program an agent and the language to specify the deliberation strategy for this agent are the same. One way to achieve this is to use procedural reflection.

2.5 Procedural reflection

In this section, we briefly review the notion of procedural reflection [des Rivières and Smith, 1984] in programming languages. A reflective programming language incorporates a model of (aspects of) the language's implementation and state of program execution in the language itself, and provides facilities to manipulate this representation. Critically, changes in the underlying implementation are reflected in the model, and manipulation of the representation by a program results in changes in the underlying implementation and execution of the program. Perhaps the best known reflective programming language is 3-Lisp [Smith, 1984].

2.5.1 A computational view

One way to understand procedural reflection is to view computation as relations on syntactical objects and runtime objects of programming languages. Here, syntactical objects of a programming language appear in the source code of programs which are written in the programming language. For example, the source code of a Jason program contains the textual representations of beliefs, events and plans which are the syntactical objects of Jason. When the source code of a program is parsed by an interpreter, syntactical objects in the source code are converted into internal representations which are called runtime objects. Runtime objects are created by using suitable data structures (defined in another programming language which implements the interpreter). For example, in the Java implementation of the Jason interpreter, there is a class $Plan$ for storing Jason plans; when the interpreter parses a Jason agent program, each plan (a syntactical object) gives rise to a Java object (a runtime object) of the class $Plan$.

In this computational view, there are two types of relations on syntactical objects and runtime objects. First, relations from syntactical objects to runtime objects are called *internalisations*. This is a mapping which converts textual representations from programs of a language into runtime objects as internal representations within an interpreter as pro-

grams are parsed. Second, relations from runtime objects to runtime objects are called *normalisations*. This is a mapping which evaluates internal representations into their *simplest* form. For example, the evaluation of query, the application of plans, and the execution of an action in Jason. In particular, the context query of a plan in Jason is evaluated into a simplest value of truth – either true or false – in order to determine if the plan can be applied to generate an intention or not. The application of a plan results in a new intention for the agent which is a sequence of actions. Then, the execution of an action of an intention results either changes to the environment (in case of an external action) or to states of an agent (in case of belief update actions and subgoal actions).

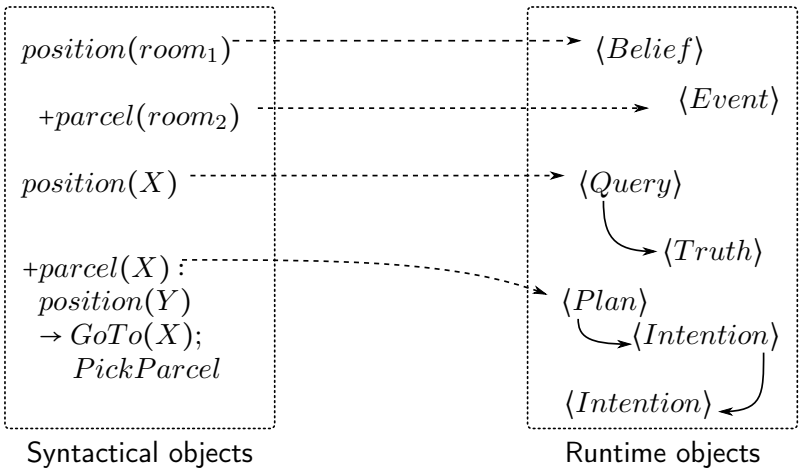

Figure 2.3: A computational view for procedural reflection.

Figure 2.3 illustrates the above computational view of agent programs in Jason. In the left hand side box are examples of syntactical objects while examples of runtime objects are grouped in the right hand side box. The notation $\langle T \rangle$ denotes a runtime object of the Java class T defined in the implementation of the interpreter Jason. Furthermore, internalisations from syntactical objects are drawn as dashed arrows while normalisations are drawn in solid arrows.

2.5.2 Reflection in programming languages

Implementing an interpreter for a programming language can be considered as implementing internalisations and normalisations in this language. Then, adding procedural reflection to the programming language involves proving facilities to influence these internalisations and normalisations. To this end, des Rivières and Smith defined two levels of reflection in a programming language, namely *structural reflection* and *procedural reflection*

Structural reflection: a language has structural reflection if it is provided with means to query and modify runtime objects. For example, one can define in a Lisp program a function which can modify the definition of other functions. When this function is applied to another function, it changes the internal representation of the other function. Thus, this allows one to influence internalisations.

Procedural reflection: a language has procedural reflection if it has structural reflection and additionally is provided with means to access the execution context of a program. Thus, this allows one to influence normalisations.

Many agent programming languages also provide support for procedural reflection. For example, PRS [Georgeff and Lansky, 1987] incorporated a meta-level, including reflection of some aspects of the state of the execution of the agent such as the set of applicable plans, allowing a developer to program deliberation about the choice of plans in the language itself. Similarly, languages such as Jason provide facilities to manipulate sets of beliefs and intentions such as belief update actions for adding and removing beliefs. However, these languages do not provide facilities to program steps in a deliberation cycle such as different strategies to select applicable plans for generating new intentions or to select intentions for execution. Therefore, the support for procedural reflection in current state-of-the-art agent programming languages is often up to the level of structural reflection only.

2.6 Simulating agent programs

In this last section, we present a brief review of the notion of bisimulation [Milner, 1989] used to compare expressive power of languages. Influenced by the work of Hindriks in [2001] where weak bisimulation is used to show the correctness of the translation, we will use bisimulation in order to underpin the translation of other agent programming languages into **meta-APL**.

When simulating an agent programming language (called the source language) in another agent programming language (called the target language), agent programs (called the source programs) of the source language are translated into agent programs (called the target programs) of the target language such that the source programs and the target programs behave equivalently with respect to a weak bisimulation [Milner, 1989]. The most recent and significant work related to simulating agent programs up to now is done by Hindriks in [2001] and by Dennis et al. in [2008]

Simulation in 3APL

In [2001], Hindriks developed an approach to simulate a source language in a target language by identifying a *translation bisimulation* which is a function translating programs and configurations of the source language into programs and configurations of the target language. Hindriks [2001] showed that such a translation bisimulation produces a weak bisimulation between the configurations of the source language and the configurations of the target language. Hence, for each run of a source program, there is an equivalent run of the target program and vice versa. Therefore, the target agent behaves equivalently to its source agent.

This approach is applied to the simulation of AgentSpeak(L) and Golog in the first version of 3APL [Hindriks et al., 1999] where the operational semantics of 3APL does not include a deliberation cycle. Therefore, it is not clear whether it is still straightforward to apply the same approach for simulating source languages whose operational seman-

tics take into account the deliberation cycles as implemented in their interpreters such as Jason [Bordini et al., 2007] and the 3APL platform [Dastani et al., 2005].

Simulation in AIL

At the level of implementing an interpreter for source languages, the Agent Infrastructure Layer (AIL) [Dennis et al., 2008] attempted to simulate behaviour of source programs in languages such as 3APL, GOAL and Jason. The purpose of AIL is to provide an intermediate layer between different agent programming languages and a model-checking framework Agent JPF (AJPF) – an extension of Java Path Finder [Visser et al., 2003].

AIL is a collection of data structures implemented as Java classes abstracting capabilities of BDI agent programming languages into which those languages can be easily translated. As Java has no formal semantics, implementations of interpreters of agent programming languages are not provided with formal semantics. Hence, there is no direct basis for establishing the equivalence of behaviours of agents running within interpreters implemented by AIL and their formal semantics.

Currently, AIL and AJPF have been applied to implement interpreters of two languages SAAPL [Winikoff, 2007] and GOAL [Hindriks et al., 2000; de Boer et al., 2007], and verify properties of agents implemented in these two languages. However, as stated in [Dennis et al., 2012], there has been no proof yet to underpin the correctness of implementing the operational semantics of the two languages in AIL.

2.7 Summary

In this chapter, we presented a brief review of intelligent agents and approaches to programming them. In particular, we reviewed the BDI architecture for designing agents. The BDI architecture has had a major impact on the development of research in the field of multi-agent systems, such as technologies for specifying and reasoning about agents via BDI logics, and building agent via BDI-based agent programming languages. To this

end, we also gave a brief survey of several existing agent programming languages. We then gave a comprehensive discussion on the support provided by current agent programming languages to program different deliberation strategies. We also reviewed the notion of procedural reflection in programming languages and its application to existing agent programming languages. Finally, we reviewed the result of simulating agents using translation bisimulation which enables the comparison on the expressiveness of different agent programming languages.

Chapter 3

The agent programming language meta-APL

In this chapter, we introduce the agent programming language **meta-APL** which allows one to encode the plans of an agent and to specify its deliberation strategy in the same agent program. The basic building blocks of the syntax of **meta-APL** are atoms, plans, and a small number of primitive operations for querying and updating the mental state and the plan state of an agent. They are then used in the syntax of other elements of **meta-APL** including clauses, macros, object-level rules, and meta rules.

3.1 Introduction

We begin with a brief introduction to the agent programming language **meta-APL**. **Meta-APL** is a BDI-based agent programming language that follows the same approach as other hybrid agent programming languages (like Jason and 3APL) where logical programming and imperative programming are combined. This allows **meta-APL** to inherit the advantages of both programming paradigms; in particular, mental attitudes can be specified and reasoned in an expressive logical language and intentions can be specified and executed in an imperative fashion. Furthermore, **meta-APL** also features procedural reflection in

order to allow the implementation of different deliberation strategies within the same programming language. In order to enable querying and modifying runtime objects (such as mental attitudes and intentions), **meta-APL** comes with meta queries to inspect the internal presentations of mental attitudes (e.g., beliefs, goals and events) and intentions as well as meta actions to manipulate them. Queries and meta actions to inspect and to modify the execution state of intentions are also provided.

Like other hybrid agent programming languages, the state of a **meta-APL** agent consists of two main components: a *mental state* and a *plan state*.

The mental state consists of mental attitudes, called *atom instances*, which represent for example a belief about the current state of affairs, a goal that the agent wants to achieve, or an event to which the agent should react, etc. Each atom instance encapsulates an atom usually of the form $type(p(t_1, \ldots, t_n))$ where $type$ specifies the type of the atom (such as $belief$, $goal$ or $event$) and p is the predicate of the atom with possible arguments $t_1, \ldots, t_n$. Recall that in other hybrid agent programming languages such as Jason and 3APL, each type of mental attitudes is stored in a separate base (e.g., belief base, goal base, and event base). However, in **meta-APL**, atom instances are gathered in the same set. There are at least two advantages in having such a design of **meta-APL**'s mental state. Firstly, it allows us to minimise the set of queries and meta actions for atom instances where queries to inspect atom instances and meta actions to modify them are the same for different types of atom instances. Secondly, it allows programmers to invent their own types of mental attitudes by using other symbols to denote the type of atom instances.

The plan state consists of a set of *plan instances* which encapsulate the intentions of an agent. In other words, they play a role similar to intentions in other hybrid agent programming languages. Similar to plans in Jason and goal planning rules in 3APL which are used to generate intentions, plan instances in **meta-APL** are generated by means of *object-level rules*.

The existence of a plan instance in the plan state is justified by one or more atom in-

stances in the mental state. For example, an atom instance is created in order to achieve a goal or react to an event represented in an atom instance. Here, atom instances which are used to justify a plan instance are called the justifications of this plan instance.

An atom instance can also be a subgoal of a plan instance. When executing a subgoal action of a plan instance, an atom instance is created. This atom instance is called the subgoal of the plan instance. Informally, the existence of the subgoal is because of the existence of the plan instance. Hence, when the plan instance is deleted from the plan state, the subgoal is also removed from the mental state.

In order to succinctly specify the above possible relationships between atom instances and plan instances, we assign each atom instance and plan instance a distinct identifier. In order to specify that an atom instance is a subgoal of a plan instance, the atom instance also stores the identifier of the plan instance as a parental pointer. If the atom instance is not a subgoal, this parental pointer has a special value nil. Similarly, in order specify that a plan instance is justified by some atom instances, the plan instance stores identifiers of these atom instances in a justification set. Using identifiers for atom instances and plan instances also allows us to differentiate different atom instances and plan instances even if their atoms or intentions are syntactically the same. Identifiers are then can also be used in queries and meta actions in order to correctly and succinctly refer to the instances that one needs to inspect or modify.

Atom instances and plan instances are manipulated by means of meta actions which are organised into *meta rules*. Each meta rule consists of a condition describing when it can be applied and a sequence of meta actions describing updates to the mental state and the plan state and enabling procedural reflection in **meta-APL**.

In order to specify the interaction between object level and meta level in **meta-APL**, object-level rules and meta rules are separated into different rule sets. Although this design decision seems to separate the object aspect and the meta aspect of **meta-APL**, having both aspects in the same agent programming language **meta-APL** can provide a good en-

gineering practice.

In particular, object-level rules and meta rules are grouped into a list of rule sets which determines the order in which they are applied. Therefore, we can easily define an arbitrary order of applying object level rules (to generate intentions) and meta rules. This allows necessary modifications to mental attitudes and intentions to happen at several times during the deliberation. For example, one can have a set R_1 of object-level rules to generate new plan instances for achieving goals, a set R_2 of meta rules to process newly received beliefs, and another set of meta rules R_3 to select certain intentions to execute. A reasonable order for these sets are R_2, R_1, then, R_3 which can be organised by the list (R_2, R_1, R_3). This means the agent first updates its mental state according to new beliefs, such as to remove achieved goals, etc., then, generates new plan instances to achieve the remaining goals, and finally, selects one of its intentions to execute in a deliberation cycle.

Furthermore, as both object and meta levels in **meta-APL** share the same syntax for mental attitudes (atom instances) and intentions (plan instances), this leaves no space for ambiguity when specifying mental attitudes and intentions in both levels. Finally, having a separate meta language (for programming deliberation strategies) and a number of object languages where the syntax of beliefs, goals, events and intentions are different require necessarily conversion to the syntax of equivalent objects in the meta language. As both object and meta levels are in **meta-APL**, such a conversion are no longer needed. This certainly simplifies the programming task as well as avoids mistakes when implementing the conversion.

An agent programmer encodes an agent in **meta-APL** by declaring initial atoms, additional queries, additional meta actions and a list of rule sets. The execution of the agent is organised in cycles each of which involves updating mental attitudes according to percepts received from the environment, applying object level rules and meta rules according to the order defined by the list of rule sets, and executing scheduled intentions. Each cycle is numbered incrementally where the first cycle is always numbered 0. The execution

of **meta-APL** agents is described in the next chapter in more detail. In the rest of this chapter, we first detail the syntax of terms as the first essential building blocks of the language. Then, we present primitive operations which are a small number of predefined queries and meta actions. These primitive operations can be categorised into two groups: one consisting of operations related to atom instances, and the other consisting of operations related to plan instances. In addition to these primitive operations, one is allowed to define additional queries and additional meta actions by means of clauses and macros, respectively. Finally, we introduce the syntax of object level rules and meta rules.

3.2 Syntax

3.2.1 Terms and atoms

Terms are the essential building blocks of **meta-APL** and can appear in every element of the language. In particular, they are the representation of different types of mental information such as perceptions, beliefs, goals and events, as well as the representation of parameters in queries and meta actions. Terms are constructed by symbols from the following disjoint sets:

- *ID* is a non-empty totally ordered set of identifiers (ids);
- *PRED* is a non-empty set of predicate symbols;
- *FUNC* is a non-empty set of function symbols; and
- *VAR* is a non-empty set of variables;

ID is the set of identifiers (ids) which are assigned to atom instances and plan instances. They are used to differentiate between different atom and plan instances (even if they are syntactically identical); e.g., two identical atom instances of a subgoal posted by different plan instances. Each atom and plan instance has a unique id.

In what follows, we denote elements of *ID* as i, j and k, elements of *FUNC* as f, g, and h, elements of *PRED* as p, q, and r, and elements of *VAR* as X, Y, and Z (like Prolog), with subscripts if necessary.

Each element of *PRED* and *FUNC* is accompanied with a number $m \in \mathbb{N}$ which specifies its arity. As usual, if a predicate $p \in$ *PRED* has arity 0, it is called a *proposition* symbol. If a function $f \in$ *FUNC* has arity 0, it is called a *constant* symbol.

A term, then, is defined inductively as follows:

- A variable $X \in$ *VAR* is a term;
- If $t_1, \ldots, t_m$ where $m \geq 0$ are terms, then so is $t_1; \ldots; t_m$
- If $f \in$ *FUNC* is a m-ary function and $t_1, \ldots t_m$ are terms, so is $f(t_1, \ldots, t_m)$; and
- If $p \in$ *PRED* is a m-ary predicate and $t_1, \ldots t_m$ are terms, so is $p(t_1, \ldots, t_m)$.

In the second case, if $m = 0$, we denote $t_1; \ldots; t_m$ by the empty sequence symbol ϵ. In the last case, the term $p(t_1, \ldots, t_m)$ is also called an atom. A term (or an atom) is ground if it does not have any variable.

In other words, the syntax of terms t and atoms a is given by:

$$\begin{aligned} t &::= i \mid X \mid \epsilon \mid t(;t)^* \mid f[(t(,t)^*)] \mid p[(t(,t)^*)] \\ a &::= p[(t(,t)^*)] \end{aligned}$$

where $i \in$ *ID*, $X \in$ *VAR*, $f \in$ *FUNC*, and $p \in$ *PRED*.

As in Prolog, we write a list of terms (or atoms) as $[t_1, \ldots, t_n]$. Note that a list is also a term which is defined by two predefined functions: the empty list constant $[]$ and the concatenation function $[t|l]$ which constructs a new list from two parameters: a term t as the head of the new list and a list l as the rest of the new list. Therefore, $[t_1, \ldots, t_n]$ is the abbreviation of the term $[t_1|[t_2|\ldots[t_n|[]]\ldots]]$.

Variables are bound to terms by means of substitutions. A substitution θ is a finite set $\{X_1/t_1, \ldots, X_n/t_n\}$ where $n \geq 0$, X_i's are distinct variables in *VAR* and t_i's are terms such

that X_i does not occur in t_i for all $i \in \{1, \ldots, n\}$. If $n = 0$, θ is called the empty substitution. The domain of θ is defined as $dom(\theta) = \{X \mid X/t \in \theta\}$. The range of θ is defined as $ran(\theta) = \{t \mid X/t \in \theta\}$. If $dom(\theta) = ran(\theta)$, then θ is call a renaming. For any $X/t \in \theta$, we write $\theta(X)$ to denote t.

The application of a substitution θ to a term t, denoted by $t\theta$, is the result of replacing simultaneously each occurrence in t of each variable $X \in dom(\theta)$ its corresponding term $\theta(X)$. Formally, $t\theta$ is defined inductively on the structure of t as follows:

- If $t = X$ where $X \in dom(\theta)$, then $t\theta = \theta(X)$;
- If $t = X$ where $X \notin dom(\theta)$, then $t\theta = X$;
- If $t = t_1; \ldots; t_m$, then $t\theta = t_1\theta; \ldots; t_m\theta$;
- If $t = f(t_1, \ldots, t_m)$, then $t\theta = f(t_1\theta, \ldots, t_m\theta)$; and
- If $t = p(t_1, \ldots, t_m)$, then $t\theta = p(t_1\theta, \ldots, t_m\theta)$.

The composition of two substitutions θ and η where $n, m \geq 0$ is denoted as $\theta\eta = \{X/(t\eta) \mid X/t \in \theta \text{ and } t\eta \neq X\} \cup \{Y/u \mid Y/u \in \eta \text{ and } Y \notin dom(\theta)\}$.

3.2.2 Primitive operations on atom instances

Atom instances constitute the mental state of an agent. Each atom instance consists of a unique id from *ID* and an atom.

Mental state queries:

For atom instances, we have a query for retrieving the id and the atom of an atom instance and another query for retrieving the cycle number at which the atom instance was created. They are listed below:

- atom(i, a): true if there is an atom instance whose id is i and atom is a;

- cycle(i, n): true if there is an atom instance whose id is i and it was created at cycle n (runs of an agent are organised in deliberation cycles which are numbered starting from 0 (initial) and incremented at each new cycle).

We also have a query for retrieving the number of the current deliberation cycle:

- cycle(c): true if the current deliberation cycle matches c.

Mental state actions:

Besides the above queries for retrieving information about atom instances, we also have meta actions for adding a new atom instance into the mental state, for deleting an atom instance from the mental state, and for deleting atom instances whose atoms match an expression. They are listed below:

- `add-atom`(a): create a new instance of the atom a (where a does not have to be ground);
- `delete-atom`(i, a): deletes an atom instance whose id is i and atom is a. This removal leads to the deletion of plan instances where the atom instance is their justification. (See Section 3.2.4 for the notions of plan instances and justifications);

3.2.3 Plans

Plans are the basic static constituents of plan states of agents. A plan is a textual representation of a sequence of actions which an agent can execute in order to modify its environment or its mental state. Let *ActionNames* be a set of external actions. Plans are built of external actions, user-defined mental state actions and subgoal actions which are defined as follows:

- ea ::= e$(t_1, \ldots, t_n)$: an external action where $e \in$ *ActionNames*, $n \geq 0$, and $t_1, \ldots, t_n$ are terms;

- mt ::=?q: a mental state test where q is a (user-defined) mental state query (see Section 3.2.5 for the definition of user-defined mental state queries);
- ma ::= m: a (user-defined) mental state action;
- sg ::=!$g(u_1, \ldots, u_m)$: a subgoal action where $g(u_1, \ldots, u_n)$ is a (possibly non-ground) term, $n \geq 0$.

Then, we define the syntax of a plan π as follows:

$$\pi ::= \epsilon \mid (ea \mid mt \mid ma \mid sg); \pi$$

where ϵ stands for an empty plan.

Note that plans are first class objects in **meta-APL**. This means one is allowed to assert beliefs about them, such as $cost(\pi_1, high)$ or $duration(\pi_2, 10)$, etc. Those beliefs can be used to reason about plans allowing the implementation of more complex deliberation cycles such as for the agent programming language AgentSpeak(XL) [Bordini et al., 2002].

3.2.4 Primitive operations on plan instances

Plan instances are elements of the plan state of agents. Each plan instance consists of an unique id, an initial plan (the one assigned for the instance when it is generated), a current suffix (the part of the instance still to be executed), one or more justifications, a substitution, and at most one subgoal. As with atom instances, ids are used to distinguish between different plan instances, even if they have syntactically identical plans.

A justification specifies an atom instance. Informally, a justification is a "reason" for executing (this instance of) the plan; e.g., an atom representing a belief, a goal or an event. In general, a plan instance may have multiple justifications, and an atom instance may be the justification of multiple plan instances.

The substitution $\theta = \{X_1/t_1, \ldots, X_n/t_n\}$ specifies the current binding of variables of the plan to terms.

A subgoal is created by the execution of a subgoal action $!g(u_1, \ldots, u_m)$, and is an instance of the atom $g(u_1, \ldots, u_m)$ which shares variables with the term in the subgoal action of the plan instance.

The substitution of a plan instance is initiated when the plan instance is created as the result of evaluating the head of object level rule against the atom instances and applying the rule (see the next chapter for more detail). However, it is not necessary that all variables in the plan instance are instantiated by its substitution. If (i) this plan instance creates a subgoal which contains one of the non-instantiated variables and (ii) the subgoal gives rise to another plan instance, the non-instantiated variable can be instantiated (such as by executing a test action in the second plan instance). This instantiation is now kept in the substitution of the second plan instance and can be propagated back to the first plan instance by the meta action `set-substitution` which will be introduced later.

A plan instance also has a set of execution state flags σ. σ is the subset of a set of flags F which includes at least `intended`, `scheduled`, `stepped` and `failed`, and may contain additional user-defined flags. For example, some deliberation strategies may require a `suspended` execution flag to specify that the execution of a plan instance is suspended. The `intended` flag indicates the agent is committed to executing this plan instance. The `scheduled` flag indicates that the plan instance is selected to be executed at the current cycle. The `stepped` flag indicates that the plan instance was executed at the last cycle. Finally, the `failed` flag indicates that the plan instance has failed to execute an action.

Plan instances which have the `intended` flag are called intentions. An intention p may have a subgoal as the result of executing its subgoal action. Then, if the subgoal is the justification of another intention p', p' is called the sub-intention of p. Furthermore, the subgoal must be achieved before continuing executing p. Therefore, we call an intention executable iff it has no subgoal.

Plan state queries

For plan instances, **meta-APL** includes queries for retrieving their plans, their initial plans, their justifications, their subgoals, their substitutions, and their state flags. There is also a query for retrieving the cycle number at which a plan instance was created. These primitive queries are listed below:

- plan(i, π): true if there is a plan instance in the plan state of the agent whose id is i and plan is π;
- init-plan(i, π): true if there is a plan instance in the plan state of the agent whose id is i and initial plan is π;
- justification(i, j): true if there is a plan instance whose id is i and there is a justification of the plan instance whose id is j;
- substitution(i, θ): true if there is a plan instance whose id is i and substitution is θ;
- subgoal(i, j): true if there is a plan instance whose id is i and there is a subgoal of the plan instance whose id is j;
- state(i, s): true if there is a plan instance whose id is i and one of its flags is s; and
- cycle(i, n): true if there is a plan instance created at cycle n whose id is i. Here, we overload the query $cycle(i, n)$ for atom instances. However, no ambiguity can arise as atom instances and plan instances will not share the same identifier.

Plan state actions:

For plan instances, **meta-APL** provides meta actions for replacing their plans, extending their substitutions, adding and removing their state flags, and deleting plan instances. These meta actions are listed below:

- `set-plan`(i, π): replaces the current suffix of a plan instance whose id is i with π,

- `set-substitution`(i, θ): extends the substitution of a plan instance whose id is i with the provided substitution θ,
- `set-state`(i, s): adds the state flag s to the set of flags of a plan instance whose id is i,
- `unset-state`(i, s): removes the state flag s from the set of flags of a plan instance whose id is i,
- `delete-plan`(i): deletes a plan instance whose id is i. This also leads to the deletion of any subgoal of the plan instance.

3.2.5 User-defined queries and meta actions

In **meta-APL**, users may define additional queries and additional meta actions from primitive operations by means of clauses and macros, respectively.

Clauses

User-defined queries are defined by means of Prolog-style Horn clauses. A clause has the following form:

$$q \leftarrow q_1, \ldots, q_n$$

where $n \geq 0$ and q_i is either a mental state query q, a user-defined query q or its negation not q. Negation is interpreted as negation as failure, and we assume that the set of clauses is always *stratified*, i.e., there are no cycles in predicate definitions involving negations. Clauses are evaluated as a sequence of queries, with backtracking on failure. Note that clauses must be "side-effect-free", i.e., they can only contain queries, not meta actions.

We say that a user-defined query is a user-defined mental state query if it is defined by using only (user-defined) mental state queries.

Example 3.2.1. Let us consider some clause examples. One can check if there is an atom instance of a given atom by a user-defined query as follows:

$$\text{atom}(X) \leftarrow \text{atom}(I, X)$$

In order to check whether a plan instance is an intention, one can define a query as follows:

$$\text{intention}(I) \leftarrow \text{state}(I, \texttt{intended})$$

To check whether an intention is executable, i.e., has no subgoal, one can define the following query:

$$\text{executable-intention}(I) \leftarrow \text{intention}(I), \text{not subgoal}(I, J).$$

Macros

Users may define additional meta actions by means of macros. A macro has the following form:

$$a = a_1; \ldots; a_n$$

where $n \geq 1$ and each of $a_1; \ldots; a_n$ is either a (user-defined) meta action or of the form $?q$ where q is either a (user-defined) query or its negation. Such a sequence is called an action sequence. The execution of a macro results in the sequential execution of meta actions or queries a_i's. Execution terminates if an action or query fails (see the next chapter for more detail).

A user-defined meta action is called a user-defined mental state action if it is defined by using only (user-defined) mental state queries and (user-defined) mental state actions.

Example 3.2.2. Let us consider some macro examples. In order to add a new atom instance of a belief b and an event of the new belief, as in AgentSpeak(L), we can define the following macro:

$$\texttt{add-belief}(B) = \texttt{add-atom}(belief(B)); \texttt{add-atom}(+belief(B)).$$

In this macro, a new instance of atom $belief(B)$ is added into the mental state together with another new atom instance of the event $+belief(B)$. Obviously, this is a user-defined mental state action.

In order to set a plan instance to become an intention, i.e., to add the `intended` flag into the set of flags of the plan instance, one may define a meta action by the following macro:

$$\texttt{add-intention}(I) = \texttt{set-state}(I, \texttt{intended})$$

3.2.6 Object-level rules

The object-level rules of **meta-APL** are used to generate plan instances. The syntax of an object-level rule is defined as follows:

$$\textit{reasons} \left[: \textit{context}\right] \rightarrow \pi$$

where:

- π is a plan,
- *reasons* ::= $q_1, \ldots, q_n$

  where $n \geq 1$ and $q_1, \ldots, q_n$ are non-negated mental state queries (user-defined ones are not allowed here).

- *context* ::= $q_1, \ldots, q_n$

  where $n \geq 0$ and $q_1, \ldots, q_n$ are (user-defined) mental state queries or their negation.

The *reasons* and the *context* parts of an object-level rule are evaluated against the mental state of an agent. The rule, then, is applicable when both of *reasons* and *context* are true. The result of applying the rule is a plan instance added into the plan state of the agent. Atom instances in the mental state which are used to evaluate the *reason* part are called the *justifications* of the plan instance. The *context* may be null (in which case the ":" may be omitted), but each plan instance must be justified by at least one reason. Furthermore, it is

also important to note that object-level rules do not make use of any meta actions for plan instances. However, we allow plans appearing in object level rules to include user-defined mental state actions as means of making changes to mental attitudes such as beliefs. The *reasons* part of an object-level rule is also called the *justification query*. In the rest of this thesis, we sometimes call object-level rules *object rules*.

3.2.7 Meta rules

Meta rules in **meta-APL** make use of (user-defined) meta actions in order to manipulate elements of the mental state and the plan state of an agent.

The syntax of meta rules is defined as follows:

$$\textit{meta-context} \rightarrow m_1; \ldots; m_n$$

where *meta-context* is a conjunction of (user-defined) queries, and $m_1, \ldots, m_n$ are (user-defined) meta actions or of the form $?q$ where q is either a (user-defined) query or its negation.

In contrast to object-level rules, there is no restriction on the usage of queries and meta actions in meta rules. The left hand side of a meta rule specifies a condition when the rule can be applied. Applying a meta rule means to execute sequentially all actions on the right hand side of the meta rule immediately. This execution terminates as soon as a query in the body of the meta rule fails. This differs from the execution of plan instances generated by object-level rules, where only the first steps of their plans are executed at a time. A bigger difference is that the plan instance generated by an object rule may never be executed.

3.2.8 Meta-APL program

A **meta-APL** agent program consists of a set of atoms defining the agent's initial mental state, e.g., beliefs, goals, events, etc., a set of clauses for defining user-defined queries, a

set of macros for defining user-defined meta actions, and a list of rule sets each of which contains either only object-level rules or only meta rules.

The syntax of an agent program in **meta-APL** is as follows:

$$(A, Q, M, R_1, \ldots, R_n)$$

where:

- $A = \{a_1, \ldots, a_m\}$ where $m \geq 0$, and $a_1, \ldots, a_m$ are atoms.
- Q is a set of clauses.
- M is a set of macros.
- $n \geq 1$ and R_i is a set of either only object-level rules or only meta rules.

Similar to Prolog, we allow the occurrence of anonymous variables, denoted by the wildcard _, in **meta-APL** agent programs. These anonymous variables in a **meta-APL** program are treated as distinct variables that do not occur in the program.

3.3 Core deliberation cycle

In this section, we describe informally the operation of a **meta-APL** agent program The core deliberation cycle of **meta-APL** consists of three main phases. In the first phase, the agent updates its mental state with atom instances resulting from perception of the agent's environment, messages from other agents etc. In the second phase, the rule sets comprising the agent's program are processed in sequence. The rules in each rule set are run to quiescence to update the agent's mental and plan state. Mental and plan state actions performed by rules directly update the internal (implementation-level) representations maintained by the deliberation cycle, and can be queried using mental state and plan state queries. Processing a set of object rules creates new plan instances. Processing a set of meta-rules may involve updating the agent's beliefs and goals, deleting intentions

for achieved goals, deleting unintended plan instances from the previous deliberation cycle, updating the agent's intentions or selecting which intention(s) to execute at this cycle, etc. Finally, in the third phase, the next step of all scheduled object-level plans is executed. The deliberation cycle then repeats.

3.4 Example Deliberation Cycles

To illustrate how **meta-APL** can be used to program deliberation strategies, we give sample code for three deliberation strategies commonly found in the BDI-based agent programming language literature. We stress that these examples do not exhaust the types of strategy that can be encoded. First, we assume to have the following queries which can be defined by **meta-APL** primitive queries:

- intended-means(I, i): I is the id list of intentions which have a common justification with id i;
- executable-intention(I): I is the id of an intention which has no subgoal;
- executable-intentions(I): I is the id list of all executable intentions.

Then, the first strategy we consider is the parallel execution of a non deterministically chosen plan for each top-level goal. It can be programmed in **meta-APL** as follows:

intended-means($[\,], R$), justification(I, R) → `set-state`(I, `intended`)

executable-intention(I) → `set-state`(I, `scheduled`)

state(I, `intended`), plan(I, ϵ), justification(I, J), not subgoal(_, J)
→ `delete-atom`(J)

state(I, `intended`), plan(I, ϵ), justification(I, J), subgoal(K, J),
substitution(I, S) → `set-substitution`(K, S); `delete-atom`(J)

The first rule selects a plan instance for each reason (goal or subgoal) and makes it an intention. The second rule schedules each executable intention for execution at this cycle.

The third and fourth rules handle the completion of subplans. The third rule caters for the case when an intention is executed completely (i.e., its plan's remainder is empty) but its justification is not a subgoal of any other plan instance: we simply delete the justification which also leads to the deletion of the intention. The fourth rule caters for the case when the justification is a subgoal of another plan instance. In this case, before deleting the justification (as in the third rule), we propagate the substitution of any variables in the subgoal by the meta action `set-substitution` which extends the substitution of the parent plan K with the substitution of the plan instance I.

The second strategy we consider is the non-interleaved execution of a single intention until completion (the rules to handle completion of subplans are as above)

not state(_, `intended`), plan(I, _) → `set-state`(I, `intended`)

state(I, `intended`), subgoal(I, J), intended-means([ ], J), justification(K, J)

→ `set-state`(K, `intended`)

executable-intention(I) → `set-state`(I, `scheduled`)

Finally, we give a simple "round-robin" strategy, which executes a single step of the next intention in the set of intentions at this cycle (again the rules to handle completion of subplans are as for parallel execution)

intended-means([ ], R), justification(I, R)

→ `set-state`(I, `intended`)

executable-intentions(I), $append$(_, [I_1, I_2|_], I), state(I_1, `stepped`)

→ `set-state`(I_2, `scheduled`)

In the second meta rule, the $append$ query to append to lists can be defined like in Prolog.

Many other deliberation strategies can be defined analogously.

3.5 Example of a meta-APL agent program

In this section, we give an example of a simple agent program in **meta-APL**. The example is the program of a service robot, as depicted in Figure 3.1, which is responsible for cleaning

rooms and delivering one box from some room to another in a building. The robot can move clock-wise from one room to another. Once arriving in a room, it can clean the room, pick up a box and drop a carried box. We assume that each room in the building is equipped with sensors for checking if the room is dirty or not, if there is a box to be delivered to another room, and the position of the robot. Whenever the agent senses the environment, it collects information obtained by the sensors in three rooms of the building. Initially, the robot is in room 2, room 1 is dirty, and there is a box in room 3 which needs to be delivered to room 2.

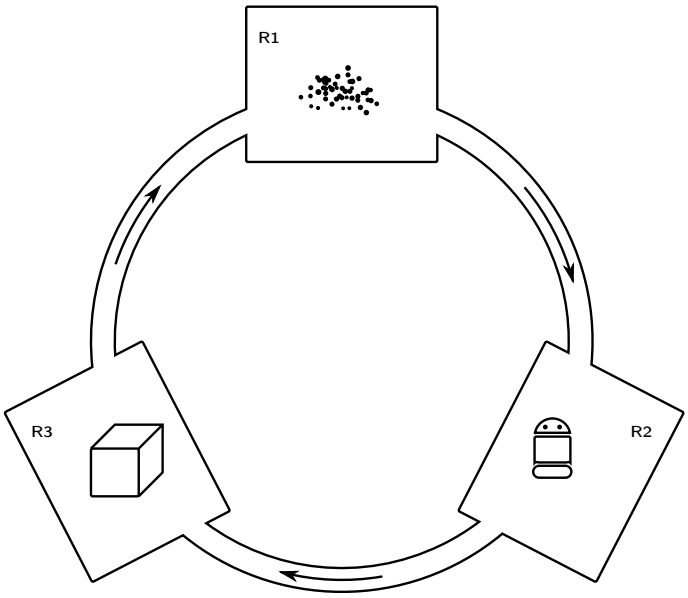

Figure 3.1: The service robot example.

In order to encode the mental state of the agent, we have the following predicates:

- $dirty(R)$: states that room R is dirty;
- $pos(R)$: states that the robot is currently in room R;
- $box(R)$: states that the box is currently in room R;
- $dest(R)$: states that the destination to deliver the box is R.

The robot is equipped with the following four external actions:

- $Go()$: moves itself to the next room;
- $Vacuum()$: cleans the current room;
- $Pick()$: picks the box up;
- $Drop()$: puts the box the agent is carrying down.

The agent program is a tuple (A, Q, M, R_1, R_2, R_3) where A is the initial mental state, Q is a set of clauses, M is empty, R_1 is a set of meta rules for processing new beliefs, R_2 is a set of object-level rules for generating plans, and R_3 is a set of meta rules for selecting plans to execute. In the following, we present these components in detail.

The initial mental state A contains a belief atom stating that the robot is initially in room 2 as follows:

$$A = \{belief(pos(room_2))\}$$

The set Q of clauses contains:

$$atom(A) \leftarrow \text{atom}(\_, A). \tag{3.1}$$

$$belief(X) \leftarrow \text{atom}(belief(X)). \tag{3.2}$$

$$goal(X) \leftarrow \text{atom}(goal(X)). \tag{3.3}$$

$$plan(I) \leftarrow \text{plan}(I, \_). \tag{3.4}$$

$$subplan(I, J) \leftarrow \text{justification}(J, K), \text{subgoal}(I, K). \tag{3.5}$$

$$intention(I) \leftarrow \text{state}(I, intended). \tag{3.6}$$

$$\textit{executable-intention}(I) \leftarrow intention(I), \text{not } subgoal(I, \_). \tag{3.7}$$

$$\textit{scheduled-intention}(I) \leftarrow intention(I), \text{state}(I, \texttt{scheduled}). \tag{3.8}$$

Most of these are straightforward. The clause (3.5) defines a query for checking the sub-plan relationship between two plan instances where a plan instance is the sub-plan of

another if one of its justifications is the subgoal of the other plan instance. The clause (3.8) defines a query for checking if an intention has been selected for execution, i.e., an intention with the state flag `scheduled`. Note that these user-defined queries are not allowed in the reason part of an object-level rule.

R_1 contains meta rules to update the mental state according to new beliefs received from the environment. It consists of the following meta rules:

$$belief(dest(X)), \text{not } goal(box(X)) \rightarrow \texttt{add-atom}(goal(box(X))). \tag{3.9}$$

$$\text{atom}(I, belief(dest(X))), goal(box(X)) \rightarrow \texttt{delete-atom}(I, \_). \tag{3.10}$$

$$\text{atom}(I, goal(X)), belief(X) \rightarrow \texttt{delete-atom}(I, \_). \tag{3.11}$$

$$\text{plan}(I, \epsilon), \text{not subgoal}(I, \_) \rightarrow \texttt{delete-plan}(I). \tag{3.12}$$

The meta rule (3.9) is for the robot to generate a new goal when it has the belief $belief(dest(X))$. This belief states that the box should be delivered to room X. The new goal $goal(box(X))$ means that has a goal to take the box to room X. The condition not $goal(box(X))$ prevents the rule from generating an instance of the goal $goal(box(X))$ if the same goal is already in the mental state. The meta rule (3.10) removes the belief $belief(dest(X))$ when the robot has adopted $goal(box(X))$ as a goal. The meta rule (3.11) is for deleting any achieved goal. Finally, the meta rule (3.12) is for deleting any plan instance which has an empty plan and no subgoal.

R_2 contains object-level rules to generate plans. It consists of the following object-level rules:

$$atom(belief(dirty(X))) \rightarrow !goal(pos(X)); Vacuum() \tag{3.13}$$

$$atom(goal(box(X))) : belief(box(Y)) \rightarrow !goal(pos(Y)); Pick(); !goal(pos(X)); Drop() \tag{3.14}$$

$$atom(goal(pos(X))) \rightarrow Go(); !goal(pos(X)) \tag{3.15}$$

The first object-level rule (3.13) generates a plan to clean a dirty room whenever the robot

has a belief that there is a dirty room. The plan consists of a subgoal to go to room X, and then perform the external action $Vacuum$. The second object-level rule (3.14) generates a plan to move a box from room Y to room X. The plan consists of a subgoal to go to room Y to pick up the box, and another subgoal to go to room X to drop the box. The last object-level rule (3.15) generates a plan to go to room X. The plan causes the robot to go around, by means of the external action Go, until it arrives at the desired room.

Finally, R_3 contains meta rules for adopting intentions from plan instances and for selecting intentions to execute. It consists of the following meta rules:

$$\text{not } intention(\_), plan(I) \rightarrow \texttt{set-state}(I, \texttt{intended}) \tag{3.16}$$

$$intention(I), subplan(I, J), \text{not } intention(J) \rightarrow \texttt{set-state}(J, \texttt{intended}) \tag{3.17}$$

$$\text{not } \textit{scheduled-intention}(\_), \textit{executable-intention}(I) \rightarrow \texttt{set-state}(I, \texttt{scheduled}) \tag{3.18}$$

These meta rules are for implementing a non-interleaved execution strategy. The meta rule (3.16) adopts an intention from an arbitrary plan instance when there is no intention in the plan state. All subplans of an intention are also intentions. This is implemented by the second meta rule (3.17). Finally, the last meta rule (3.18) selects an executable intention, i.e., the one without any subgoal, to execute.

An excerpt from a run of this agent is presented in Appendix B in order to give an idea of how the agent works. Note that the agent has a formal semantics which can be seen as a model for Computation Tree Logic (CTL). If one could formalise correctness properties of the agent program as CTL formulas, they could be verified using a suitable theorem prover or model checker. However, as formal verification is not in the scope of this thesis, we do not discuss the correctness of **meta-APL** agents further.

3.6 Summary

In this chapter, we presented the syntax of **meta-APL**. Agents programmed in **meta-APL** are based on the BDI architecture. Each agent is comprised of a mental state containing

information such beliefs, goals and events, and a plan state containing the plans of the agent. The language includes a set of queries for retrieving information from the mental state and the plan state, and a set of meta actions for manipulating them. These queries and meta actions are used in object-level rules which are used to generate plans of agents. They are also used to form meta rules for controlling the management of mental states and plan states.

For demonstrating the language, we also present in the chapter a full program of a robot servicing a three-room building. We informally discuss the construction and meaning of elements in the agent program.

Chapter 4

Operational semantics of meta-APL

The operation of a **meta-APL** agent is determined by its operational semantics which is a collection of transition rules specifying how agents modify their mental and plan states. Before introducing these transition rules, it is necessary to define notions including configurations and formal meanings of queries and meta actions.

4.1 Agent configuration

Informally, a configuration is a state of an agent. It contains static elements, such as clauses, macros and rules from the program of the agent, as well as dynamic elements, such as a mental state consisting of atom instances and a plan state consisting of plan instances. Let us first formalise the notions of atom instances and plan instances as follows.

4.1.1 Atom and plan instances

An atom instance is defined as follows:

Definition 4.1.1 (atom instance)**.** *An atom instance is a tuple* (i, a, j, n) *where:*

- $i \in$ *ID is the unique id;*
- a *is the atom (which need not be ground);*

- $j \in ID$ *is the id of the parental plan instance; and*
- $n \in \mathbb{N}$ *is the cycle when the atom instance was created.*

Given an atom instance (i, a, j, n), i is restricted to be unique among ids of other atom instances and plan instances. j is the id of the plan instance which creates this atom instance by means of a subgoal action. Thus, the atom instance is a subgoal of the plan instance with id j. If an atom instance is not a subgoal of any plan instance, j = *nil*.

Example 4.1.1. Let us return to the example presented in Section 3.5. A belief which states that room 1 is dirty can be stored by the following atom instance:

$$(6, belief(dirty(room_1)), nil, 3)$$

In this atom instance, 6 is the id. The predicate $belief$ is used in the second component to denote that this is an instance of a belief. The third component is *nil* which means that this atom instance is not a subgoal of any plan instance. Finally, number 3 says that this atom instance appearing since cycle number 3 in the run of the agent.

A plan instance is defined as follows.

Definition 4.1.2. *(plan instance) A plan instance is a tuple* $(i, \pi_{init}, \pi, \theta, fs, js, n)$ *where:*

- $i \in ID$ *is the unique id.*
- π_{init} *is the initial plan.*
- π *is the plan.*
- θ *is a substitution for variables in the plan.*
- *fs is the set of state flags.*
- *js is the set of ids of justifications.*
- $n \in \mathbb{N}$ *is the number of the cycle when the plan instance was created.*

Similar to atom instances, the id i of a plan instance $p = (i, \pi_{init}, \pi, \theta, fs, js, n)$ is unique among ids of other plan instances and atom instances. Let r be the object-level rule which creates the plan instance. The initial plan π_{init} is a copy of the plan of r. Justifications specified by ids in *js* are atom instances which were used to evaluate the reason part of r. The substitution θ is initially obtained by the evaluation of the reason and the context of r against the set of atom instances forming the agent's mental state. The execution of this plan instance can give rise to a subgoal (which may not be ground). When this subgoal becomes a justification of another plan instance p', variables in the subgoal is propagated to the plan instance p' and can be instantiated (such as by the test goal action). One can extend these instantiations to the substitution of the plan instance p (when p' is completely executed) by means of the meta action `set-substitution`. The initial plan π_{init} and the set *js* are unchanged during the lifespan of the plan instance. Together, they are used to prevent the same object-level rule being applied twice. If the plan instance $(i, \pi_{init}, \pi, \theta, fs, js, n)$ is in the plan state of the agent, the object-level rule r will not be fired again to generate another plan instance which shares the same initial plan π_{init} and the set *js* of justification ids even if both reason and context parts of r are true. Only when the plan instance is removed from the plan state, can the rule r be applied again.

Executing a plan instance means to execute the plan π. When the plan instance is created, π is the same as π_{init}. When the plan instance is executed, the first action of π will be performed and removed from π. Therefore, the plan π is sometimes called the remainder plan of the plan instance.

Example 4.1.2. The agent in Section 3.5 may have the following plan instance, which is generated by the object level rule (3.13), in order to clean room 1:

$$(8, !goal(pos(room_1)); Vacuum(), Vacuum(), \{X/room_1\}, \{6\}, \{\texttt{intended}, \texttt{stepped}\}, 5)$$

The initial plan of this plan instance is the same as the plan in the object level rule (3.13) with the variable X is bound to $room_1$. This binding is stored in the substitution compo-

nent of the plan instance. The set of justification ids of this plan instance is the singleton $\{6\}$, as this plan instance has only one justification with id 6. In the set of state flags there is the flag `intended`, i.e., the plan instance is an intention, and the flag `stepped`, i.e., the plan instance has just been executed in the last cycle.

Recall that atom instances and plan instances can be related. Their relationships are stored in the parental id of an atom instance, specifying a subgoal relationship, and the set of justification ids of a plan instance, specifying a justification relationship.

Each atom instance may be a subgoal of at most one plan instance since the atom instance has at most one parental id. While a plan instance can have multiple subgoals over time, it has at most one subgoal at a time. The reason is that if the plan instance has a subgoal, the plan instance is not executable. The current subgoal must be therefore achieved before it is possible to execute the plan instance again. Note that if a plan instance is deleted from the plan state, its subgoal is also deleted from the mental state.

In contrast to subgoals, a plan instance may have several justifications. The justifications are determined when the plan instance is created by applying some object-level rule. Conversely, an atom instance can be the justification of several plan instances. If the justification of a plan instance is deleted from the mental state, the plan instance is also deleted from plan state. However, if a plan instance is deleted from the plan state, its justifications are not deleted.

In order to have succinct references to components of atom instances and plan instances, we define the following functions:

- Given $\alpha = (i, a, j, n)$, we define $\mathrm{id}(\alpha) = i$, $\mathrm{atom}(\alpha) = a$, $\mathrm{par}(\alpha) = j$, and $\mathrm{cycle}(\alpha) = n$.
- Given $p = (i, \pi_{\mathit{init}}, \pi, \theta, \mathit{fs}, \mathit{js}, n)$, we define $\mathrm{id}(p) = i$, $\mathrm{init}(p) = \pi_{\mathit{init}}$, $\mathrm{plan}(p) = \pi$, $\mathrm{subs}(p) = \theta$, $\mathrm{flags}(p) = \mathit{fs}$, $\mathrm{justs}(p) = \mathit{js}$, and $\mathrm{cycle}(p) = n$.

Example 4.1.3. For the atom instance α in Example 4.1.1, we have that: $\mathrm{id}(\alpha) = 6$, $\mathrm{atom}(\alpha) = \mathit{belief}(\mathit{dirty}(\mathit{room}_1))$, $\mathrm{par}(\alpha) = \mathit{nil}$, and $\mathrm{cycle}(\alpha) = 3$.

Given a set of atom instances A, we denote ID_A to be the set of all ids of atom instances in A, i.e., $ID_A = \{\mathrm{id}(\alpha) \mid \alpha \in A\}$. Similarly, given a set of plan instances Π, we denote ID_Π to be the set of all ids of plan instances in Π, i.e., $ID_\Pi = \{\mathrm{id}(p) \mid p \in \Pi\}$.

In order to have succinct presentation of the update of components in a plan instance $p = (i, \pi_{init}, \pi, \theta, fs, js, n)$, we define the following auxiliary functions:

$$\textit{add-flag}((i, \pi_{init}, \pi, \theta, fs, js, n), f) = (i, \pi_{init}, \pi, \theta, fs \cup \{f\}, js, n) \tag{4.1}$$

$$\textit{del-flag}((i, \pi_{init}, \pi, \theta, fs, js, n), f) = (i, \pi_{init}, \pi, \theta, fs \setminus \{f\}, js, n) \tag{4.2}$$

$$\textit{set-subs}((i, \pi_{init}, \pi, \theta, fs, js, n), \theta') = (i, \pi_{init}, \pi, \theta\theta', fs, js, n) \tag{4.3}$$

$$\textit{update-rem}((i, \pi_{init}, \pi, \theta, fs, js, n), \pi') = (i, \pi_{init}, \pi', \theta, fs, js, n) \tag{4.4}$$

$$\textit{update-executing}((i, \pi_{init}, \pi, \theta, fs, js, n), \pi') = \textit{update-rem}(\textit{add-flag}(\textit{del-flag}((i, \pi_{init}, \pi, \theta, fs, js, n), \texttt{scheduled}), \texttt{stepped}), \pi') \tag{4.5}$$

$$\textit{update-failing}((i, \pi_{init}, \pi, \theta, fs, js, n), \pi') = \textit{update-rem}(\textit{add-flag}(\textit{del-flag}((i, \pi_{init}, \pi, \theta, fs, js, n), \texttt{scheduled}), \texttt{failed}), \pi') \tag{4.6}$$

Note that the above auxiliary functions are not meta actions available in **meta-APL**. They will be used as abbreviations within definitions of transition rules later in this chapter. *add-flag* is for adding a flag f in to the flag set of the plan instance p. Obviously, if f is already in the flag set, the resulting plan instance of *add-flag* is the same as p. Similarly, *del-flag* is for removing a flag f from the flag set of the plan instance p. *set-subs* extends the substitution of p by the substitution θ'. *update-rem* is for replacing the current plan with the plan π'. The last two functions *update-executing* and *update-failing* is for replacing the flag `scheduled` in p with the flag `stepped` and the flag `failed`, respectively. They also replace the plan of p with a new plan π'.

4.1.2 Configurations

We have the following definition of configurations:

Definition 4.1.3. *A configuration is a tuple* $\langle Q, M, R_1 \dots R_k, A, \Pi, \rho, n \rangle$ *where:*

- Q *is a set of clauses.*
- M *is a set of macros.*
- $k \geq 1$ *and for all* $1 \leq i \leq k$, R_i *is a set of either only object-level rules or only meta rules.*
- A *is the mental state which is a set of (possibly non-ground) atom instances.*
- Π *is the plan state which is a set of plan instances.*
- $0 \leq \rho \leq k + 2$ *is a phase counter.*
- $n \in \mathbb{N}$ *is a cycle counter.*

In a configuration, the ids of atom instances and plans instances are unique, i.e., there are no instances of either atoms or plans with the same id. Furthermore, the cycle numbers of these atom instances and plan instances are not greater than the cycle number of the configuration. In addition to atom instances and plan instances, a configuration also contains a phase counter and a cycle counter. The cycle counter is the number of the current deliberation cycle. It also keeps track of how many deliberation cycles have elapsed. Each cycle is divided into $k+3$ stages where stage 0 corresponds to the sense phase, stages $\rho \in \{1, \dots, k\}$ correspond to the phase of applying the rule sets, and $k+1$, $k+2$ correspond to the phase of executing intentions. The agent operates by moving from stage 0 to $k + 2$ incrementally; then, returns to stage 0 of the next cycle.

Given a configuration $\langle Q, M, R_1 \dots R_k, A, \Pi, \rho, n \rangle$, the set Q of clauses, the set M of macros, and the sets $R_1, \dots, R_k$ of object-level and meta rules are static. They do not change during execution of the agent. However, the mental state A, the plan state Π, the phase counter ρ and the cycle counter n are dynamic. In the interests of brevity, in the

rest of this thesis, we use $\langle A, \Pi, \rho, n \rangle$ to denote the configuration when no ambiguity can arise.

From a given agent configuration $C = \langle A, \Pi, \rho, n \rangle$, we derive the justification binary relation $Just_c$ and the subgoal binary relation Sub_c as follows:

- $Just_C = \{(i, \mathrm{id}(p)) \mid p \in \Pi, i \in \mathrm{justs}(p)\}$.
- $Sub_C = \{(\mathrm{par}(\alpha), \mathrm{id}(\alpha)) \mid \alpha \in A, \mathrm{par}(\alpha) \neq nil\}$.

These relations are used later in this chapter as we define transition rules in the operational semantics of **meta-APL**.

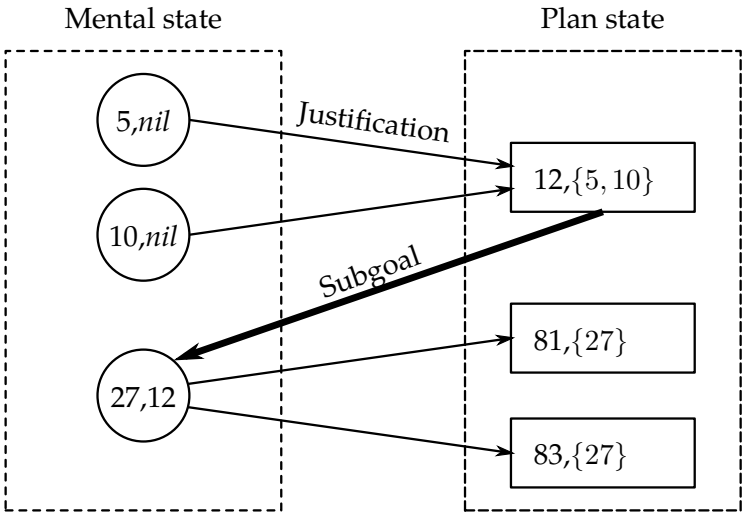

Figure 4.1: Example of the subgoal and justification relations.

Example 4.1.4. Figure 4.1 shows an example of a configuration C where we assume that there are three atom instances and three plan instances. In this example, each atom instance is illustrated by a circle annotated with its id and its parental id. Similarly, each plan instance is illustrated by a rectangle annotated with its id and the justification component. Other components of the atom and plan instances are not of interest in this example, and, are omitted. The justification relation $Just_C$ is denoted by thin arrows with direction from atom instances to plan instances. Similarly, The subgoal relation Sub_C is denoted by bold arrows with direction from plan instances to atom instances.

In the following definition, given a finite set of atoms A, we denote id_A to be some injective mapping from A into *ID*, i.e., for all $a, a' \in A$, if $a \neq a'$ then $id_A(a) \neq id_A(a')$. We shall use the notation id_A to initialise the ids for atoms of a **meta-APL** program.

Definition 4.1.4. *Let $Ag = (A, Q, M, R_1, \ldots, R_k)$ be a **meta-APL** agent program. The initial configuration of Ag is $\langle A_0, \varnothing, 0, 0\rangle$ where $A_0 = \{(id_A(a), a, nil, 0) \mid a \in A\}$.*

The above definition means that, initially, each agent has a mental state initialised from the set A of initial atoms, an empty plan state, a phase counter 0 and a cycle counter 0. In other words, the agent is at the sense phase of the first cycle of its execution.

Example 4.1.5. In the example of Section 3.5, the initial configuration of the robot is as follows:

$$(\{(1, belief(pos(room_2)), nil, 0)\}, \varnothing, 0, 0)$$

4.2 Semantics of queries and meta actions

In this section, we formally define how information is retrieved from queries and the effects of meta actions on mental states and plan states.

4.2.1 Answering queries

The evaluation of a query q with respect to a configuration C is answered by a substitution θ which is the most general unifier (mgu) of the query and some element of the configuration. This evaluation is written as $C \vdash q \mid \theta$. Given t_1 and t_2, we write $t_1 = t_2 \mid \theta$ iff t_1 and t_2 unify with mgu θ.

In the following, we define how each primitive query is evaluated against a given configuration $C = \langle A, \Pi, \rho, n\rangle$. Where there are multiple answers for a query, we assume Prolog backtracking semantics where plans are returned in the program order and atoms in ID order.

Mental state queries

- $C \vdash \text{atom}(i, a) \mid \theta$ iff $\exists\ \alpha \in A : (\text{id}(\alpha), \text{atom}(\alpha)) = (i, a) \mid \theta$.

Plan state queries

- $C \vdash \text{init-plan}(i, \pi) \mid \theta$ iff $\exists p \in \Pi : (\text{id}(p), \text{init}(p)) = (i, \pi) \mid \theta$.
- $C \vdash \text{plan}(i, \pi) \mid \theta$ iff $\exists p \in \Pi : (\text{id}(p), \text{plan}(p)) = (i, \pi) \mid \theta$.
- $C \vdash \text{justification}(i, j) \mid \theta$ iff $\exists p \in \Pi, \alpha \in A : \text{id}(\alpha) \in \text{justs}(p) \wedge (\text{id}(p), \text{id}(\alpha)) = (i, j) \mid \theta$.
- $C \vdash \text{substitution}(i, \vartheta) \mid \theta$ iff $\exists p \in \Pi, \alpha \in A : (\text{id}(p), \text{subs}(p)) = (i, \vartheta) \mid \theta$.
- $C \vdash \text{subgoal}(i, j) \mid \theta$ iff $\exists p \in \Pi, \alpha \in A : \text{id}(p) = \text{par}(\alpha) \wedge (\text{id}(p), \text{id}(\alpha)) = (i, j) \mid \theta$.
- $C \vdash \text{state}(i, s) \mid \theta$ iff $\exists p \in \Pi, f \in \text{flags}(p) : (\text{id}(p), f) = (i, s) \mid \theta$.

Queries for cycles

- $C \vdash \text{cycle}(i, c) \mid \theta$ iff $\exists \alpha \in A : (\text{id}(\alpha), \text{cycle}(\alpha)) = (i, c) \mid \theta$ or $\exists p \in \Pi : (\text{id}(p), \text{cycle}(p)) = (i, c) \mid \theta$.
- $C \vdash \text{cycle}(c)$ iff $c = n$.

Negation queries

- $C \vdash \text{not } q \mid \varnothing$ iff it does hold that $C \vdash q \mid \theta$ for any substitution θ.

User-defined queries

Let q be a user-defined query defined by $q \leftarrow q_1, \ldots, q_m \in Q$. Then we define:

- $C \vdash q \mid \theta$ iff $C \vdash q_1, \ldots, q_m \mid \theta$

where $C \vdash q_1, \ldots, q_m \mid \theta$ is defined below.

Justification and context queries

Let $q_1, \ldots, q_m$ be a justification or a context of an object-level rule or a meta rule. Then we define:

- $C \vdash q_1, \ldots, q_m \mid \theta$ iff $\exists \theta_1, \ldots, \theta_m$ such that $\theta = \theta_1 \ldots \theta_m$ and $C \vdash q_1 \mid \theta_1, \ldots, C \vdash q_m \mid \theta_1 \ldots \theta_m$.

4.2.2 Determining justifications

When evaluating the reason part of an object-level rule with respect to a configuration, it is important not only to find out the answering substitution but also to know which atom instances in the configuration are used to give the answer. These atom instances will become the justifications of the resulting plan instance when the object-level rule is applied.

Let $q_1, \ldots, q_m$ be a list of mental state queries, i.e., q_i is of the form either atom(j, a) or cycle(j, c). We define a function $ids((q_1, \ldots, q_m)) = \bigcup_{i \in \{1,\ldots,m\}} ids(q_i)$ where:

$$ids(q_i) = \begin{cases} \{j\} & \text{if } q = \text{atom}(j, a) \\ \{j\} & \text{if } q = \text{cycle}(j, c) \end{cases}$$

This function collects all the ids of the atom instances which are used to give answer to the mental state queries $q_1, \ldots, q_m$.

4.2.3 Semantics of meta actions

We specify the effects of meta actions on configurations in terms of transition relations. In particular, for each meta action ma, we define a binary relation $\xrightarrow{ma}$ on configurations which describes the resulting configuration when ma is performed from a configuration.

Meta actions for atom instances

- $\langle A, \Pi, \rho, n \rangle \xrightarrow{\texttt{add-atom}(a)} \langle A \cup \{(i, a, nil, n)\}, \Pi, \rho, n \rangle$ where i is a new id from ID, i.e., $i \in ID \setminus (ID_A \cup ID_\Pi)$.

- $\langle A, \Pi, \rho, n\rangle \xrightarrow{\texttt{delete-atom}(i,a)} \langle A', \Pi', \rho, n\rangle$ where
  - $A' = A \setminus \{\alpha \in A \mid \exists \alpha_0 \in A : \mathrm{id}(\alpha_0) = i \wedge \mathrm{atom}(\alpha_0) = a \wedge (\mathrm{id}(\alpha_0), \mathrm{id}(\alpha)) \in (\mathit{Sub}_{\langle A,\Pi,\rho,n\rangle} \cup \mathit{Just}_{\langle A,\Pi,\rho,n\rangle})^*\}$
  - $\Pi' = \Pi \setminus \{p \in \Pi \mid \exists \alpha_0 \in A : \mathrm{id}(\alpha_0) = i \wedge \mathrm{atom}(\alpha_0) = a \wedge (\mathrm{id}(\alpha_0), \mathrm{id}(p)) \in (\mathit{Sub}_{\langle A,\Pi,\rho,n\rangle} \cup \mathit{Just}_{\langle A,\Pi,\rho,n\rangle})^*\}$

As usual, the notation R^* denotes the reflexive transitive closure of a binary relation R. For the effect of $\texttt{delete-atom}(i, a)$, the reflexive transitive closure $(\mathit{Sub}_{\langle A,\Pi,\rho,n\rangle} \cup \mathit{Just}_{\langle A,\Pi,\rho,n\rangle})^*$ is used to determine atom and plan instances which are related to the atom instance with id i and atom a. Then, $\texttt{delete-atom}(i, a)$ deletes all these related atom and plan instances.

Example 4.2.1. Let us return to Example 4.1.4. The effect of the meta action $\texttt{delete-atom}(5)$ (or $\texttt{delete-atom}(10)$) is that not only the atom instance with id 5 (or 10) is deleted but also all related atom instances and plan instances with ids 12, 27, 81, and 83 are deleted as well. Conversely, the effect of the meta action $\texttt{delete-atom}(27)$ only deletes the atom instance with id 27 and the plan instances with ids 81 and 83. In this case, the plan instance with id 12 is executable again as it has no subgoal.

Meta actions for plan instances

- $\langle A, \Pi, \rho, n\rangle \xrightarrow{\texttt{set-plan}(i,\pi)} \langle A, \Pi', \rho, n\rangle$ where
  $\Pi' = \Pi \setminus \{p \in \Pi \mid \mathrm{id}(p) = i\} \cup \{\mathit{update\text{-}rem}(p, \pi) \mid \exists p \in \Pi : \mathrm{id}(p) = i\}$
- $\langle A, \Pi, \rho, n\rangle \xrightarrow{\texttt{set-substitution}(i,\theta)} \langle A, \Pi', \rho, n\rangle$ where
  $\Pi' = \Pi \setminus \{p \in \Pi \mid \mathrm{id}(p) = i\} \cup \{\mathit{set\text{-}subs}(p, \theta) \mid \exists p \in \Pi : \mathrm{id}(p) = i\}$
- $\langle A, \Pi, \rho, n\rangle \xrightarrow{\texttt{set-state}(i,f)} \langle A, \Pi', \rho, n\rangle$ where
  $\Pi' = \Pi \setminus \{p \in \Pi \mid \mathrm{id}(p) = i\} \cup \{\mathit{add\text{-}flag}(p, f) \mid \exists p \in \Pi : \mathrm{id}(p) = i\}$
- $\langle A, \Pi, \rho, n\rangle \xrightarrow{\texttt{unset-state}(i,f)} \langle A, \Pi', \rho, n\rangle$ where
  $\Pi' = \Pi \setminus \{p \in \Pi \mid \mathrm{id}(p) = i\} \cup \{\mathit{del\text{-}flag}(p, f) \mid \exists p \in \Pi : \mathrm{id}(p) = i\}$

- $\langle A, \Pi, \rho, n\rangle \xrightarrow{\texttt{delete-plan}(i)} \langle A', \Pi', \rho, n\rangle$ where
  - $\Pi' = \Pi \setminus \{p \in \Pi \mid (i, \mathrm{id}(p)) \in (\mathit{Sub}_{\langle A,\Pi,\rho,n\rangle} \cup \mathit{Just}_{\langle A,\Pi,\rho,n\rangle})^*\}$
  - $A' = A \setminus \{a \in A \mid (i, \mathrm{id}(a)) \in (\mathit{Sub}_{\langle A,\Pi,\rho,n\rangle} \cup \mathit{Just}_{\langle A,\Pi,\rho,n\rangle})^*\}$

The above definition makes use of auxiliary functions (4.4), (4.3), (4.1) and (4.2) in order to present the effect of these meta actions succinctly. As we can see, the first four transition relations specify how components of a plan instance (with id matching i) are modified. For example, the first translation, which describes the effect of the meta action `set-plan`(i, π), is obtained by firstly removing the plan instance p with id matching i, and then adding the modified plan instance *update-rem*(p, π) where the plan of p is replaced with π, as defined by *update-rem* in (4.4). Finally, the semantics of `delete-plan`(i) is similar to that of `delete-atom`, where it also leads to the deletion of related atom and plan instances.

Example 4.2.2. Return to Example 4.1.4. The effect of `delete-plan`(12) is to delete related atom instances and plan instances with ids 12, 27, 81, and 83.

Action sequences and user-defined meta actions

In order to give semantics for meta actions defined by users, we first define the semantics of action sequences which are used in macros and meta rules. Recall that such a sequence contains primitive meta actions, user-defined meta actions, primitive queries, and user-defined queries.

Definition 4.2.1. *Given an action sequence $ma_1; \ldots; ma_k$ where $k \geq 2$, we define the transition $\langle A, \Pi, \rho, n\rangle \xrightarrow{ma_1;\ldots;ma_k} \langle A', \Pi', \rho, n\rangle$ inductively as follows:*

- *If ma_1 is a primitive meta action, then:*

$$\langle A, \Pi, \rho, n\rangle \xrightarrow{ma_1;\ldots;ma_k} \langle A', \Pi', \rho, n\rangle \textit{ iff } \langle A, \Pi, \rho, n\rangle \xrightarrow{ma_1} \langle A'', \Pi'', \rho, n\rangle \textit{ and } \langle A'', \Pi'', \rho, n\rangle \xrightarrow{ma_2;\ldots;ma_k} \langle A', \Pi', \rho, n\rangle$$

- *If* ma_1 *is a user-defined meta action defined by a macro* $ma_1 = mb_1; \ldots; mb_l$*, then:*

  $\langle A, \Pi, \rho, n\rangle \xrightarrow{ma_1;\ldots;ma_k} \langle A', \Pi', \rho, n\rangle$ *iff* $\langle A, \Pi, \rho, n\rangle \xrightarrow{mb_1;\ldots;mb_l;ma_2;\ldots;ma_k} \langle A', \Pi', \rho, n\rangle$

- *If* $ma_1 = ?q$ *where* q *is a primitive or user-defined query and* $\langle A, \Pi, \rho, n\rangle \vdash q \mid \theta$*, then:*

  $\langle A, \Pi, \rho, n\rangle \xrightarrow{ma_1;\ldots;ma_k} \langle A', \Pi', \rho, n\rangle$ *iff* $\langle A, \Pi, \rho, n\rangle \xrightarrow{(ma_2;\ldots;ma_k)\theta} \langle A', \Pi', \rho, n\rangle$

- *If* $ma_1 = ?q$ *where* q *is a primitive or user-defined query and* $\langle A, \Pi, \rho, n\rangle \nvdash q$*, then:*

  $\langle A, \Pi, \rho, n\rangle \xrightarrow{ma_1;\ldots;ma_k} \langle A, \Pi, \rho, n\rangle$

From a given configuration $\langle A, \Pi, \rho, n\rangle$ and an action sequence, the above definition specifies how to recursively compute the outcome configuration $\langle A', \Pi', \rho, n\rangle$. When the first element of the action sequence is a primitive meta action, we simply use its semantics as defined previously in this section in order to determine an intermediate configuration. Then, from this intermediate configuration and the rest of the action sequence, we recursively determine the outcome configuration. When the first element is a user-defined meta action, we simply expand the sequence with the body of the macro defining the meta action. Then, from the given configuration and the expanded action sequence, we recursively determine the outcome configuration. Finally, when the first element is a (user-defined) query, we first evaluate the query. If the evaluation is true, we apply the result of the query, which is a substitution, to the rest of the action sequence. Then, from the given configuration and the rest of the action sequence, we recursively determine the outcome configuration. Otherwise, the outcome configuration is the same as the given configuration, i.e., execution finishes early and the execution of the remainder $ma_2; \ldots; ma_k$ is aborted.

4.3 Operational semantics

The operational semantics of **meta-APL** defines possible runs of an agent programmed in the language. Given a **meta-APL** agent $(A, Q, M, R_1, \ldots, R_k)$, a run of the agent consists of

consecutive deliberation cycles. Each deliberation cycle is divided into $k + 3$ stages which are categorised into three phases: the "sense" phase, the "apply" phase and the "exec" phase. Figure 4.2 illustrates stages in a deliberation cycle of agents in **meta-APL**. In stage

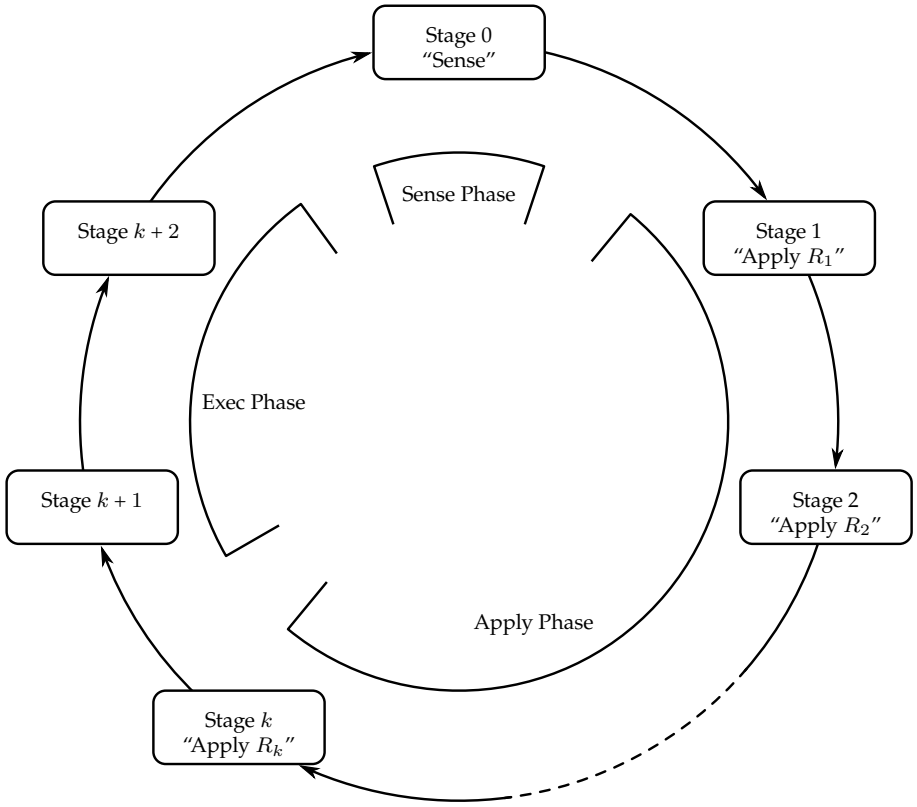

Figure 4.2: Phases in **meta-APL** deliberation cycle.

0, the agent performs a *sense* action in order to collect perceptions from the environment, then updates its mental state accordingly. After that, in the apply phase, rules from the rule sets $R_1, \ldots, R_k$ are applied to quiescence in stages from 1 to k, respectively. Then, at stage $k+1$ of the exec phase, the agent removes the flag `stepped` from intentions executed from the previous cycle. Finally, it executes scheduled intentions at the last stage $k + 2$.

Details of the execution of the agent at each stage are defined by means of a set of transition rules [Plotkin, 1981]. A transition rule has the following form:

$$\frac{\text{Conditions}}{C \longrightarrow C'} \qquad \text{(Rule name)}$$

On the top, the condition part specifies when the transition rule is applicable. The bottom of the rule describes changes of the original configuration C to which the transition rule is applied. C' is the resulting configuration of the transition rule.

In the rest of this section, we list transition rules of the operational semantics of **meta-APL**. They are gathered into three phases of a deliberation cycle.

4.3.1 The Sense phase

This is the starting phase of a deliberation cycle where the agent updates its mental state according to the perceptions received from the environment. Let *sense* be the function which specifies how the mental state A of an agent is updated with respect to the state env of the environment. The definition of *sense* depends on the nature of the agent's interaction with its environment and its deliberation cycle, but typically results in the addition and/or removal of atom instances. In particular, for any perception which indicates a belief P is true and there is no instance of atom $belief(P)$ is in A, a new atom instance of $belief(P)$ is added to A together with a new atom of $+belief(P)$. Conversely, for any perception which indicates a belief P is no longer true, any atom instance of $belief(P)$ in A is deleted from A together with a new instance of $-belief(P)$. The main idea behind having atom instances of atoms $+belief(P)$ and $-belief(P)$ is to enable one to write object level rules to react to belief changes. Once applied, these object level rules generate plan instances which have these atom instances of $+belief(P)$ and $-belief(P)$ as their justifications. When the plan instances are executed completely, these atom instances can be deleted explicitly and only by the meta action `delete-atom`.

The only transition rule in this phase is defined as follows:

$$\frac{\begin{array}{l} \mathit{sense}(env, A) = A' \\ A'' = A' \setminus \{\alpha' \in A \mid \exists \alpha \in A \setminus A' : (\mathrm{id}(\alpha), \mathrm{id}(\alpha')) \in (\mathit{Sub}_{\langle A,\Pi,0,n\rangle} \cup \mathit{Just}_{\langle A,\Pi,0,n\rangle})^*\} \\ \Pi' = \Pi \setminus \{p \in \Pi \mid \exists \alpha \in A \setminus A' : (\mathrm{id}(\alpha), \mathrm{id}(p)) \in (\mathit{Sub}_{\langle A,\Pi,0,n\rangle} \cup \mathit{Just}_{\langle A,\Pi,0,n\rangle})^*\} \end{array}}{\langle A, \Pi, 0, n\rangle \longrightarrow \langle A'', \Pi', 1, n\rangle} \quad \text{(SENSE)}$$

This transition rule is applied when the phase counter is 0. Then, the function *sense* is used to determine the updated mental state A' from the current mental state A and the current state env of the environment. This update is described in the first condition of the rule. Since some atom instances may be removed from A, i.e., elements of $A \setminus A'$, this transition rule also deletes atom and plan instances related to atom instances in $A \setminus A'$ by the reflexive transitive closure $(\mathit{Sub}_{\langle A,\Pi,0,n\rangle} \cup \mathit{Just}_{\langle A,\Pi,0,n\rangle})^*$. These are described in the second and third conditions of the rule. In the resulting configuration, the phase counter is increased to 1 and the agent moves to the next phase.

4.3.2 The Apply phase

In this phase, the agent goes through a sequence of stages ρ from 1 to k. In each stage ρ, rules in the rule set R_ρ are applied to quiescence before moving to the next one. Recall that R_ρ may contain only object-level rules or only meta rules.

Application of object-level rules

If R_ρ contains object-level rules, a new plan instance is generated when a rule in R_ρ is applied. The application of an object-level rule is defined by the following transition rule:

$$\frac{\begin{array}{l} \exists (r : c \to \pi) \in R_\rho : \langle A, \Pi, \rho, n\rangle \vdash r, c \mid \theta \\ \nexists p \in \Pi : \mathsf{init}(p) = \pi \wedge \mathsf{justs}(p) = \mathit{ids}(r\theta) \\ i_{\mathrm{new}} \in \mathit{ID} \setminus (\mathit{ID}_A \cup \mathit{ID}_\Pi) \\ \Pi' = \Pi \cup \{(i_{\mathrm{new}}, \pi, \pi, \theta, \mathit{ids}(r\theta), n)\} \end{array}}{\langle A, \Pi, \rho, n\rangle \longrightarrow \langle A, \Pi', \rho, n\rangle} \quad \text{(OBJ-APPLY-1)}$$

In OBJ-APPLY-1, an object-level rule $r : c \to \pi$ in R_ρ is applicable if its condition $r : c$ is evaluated to be true with respect to the current configuration, and if there is no plan instance in the plan state with exactly the same initial plan and justification. These are the first and the second conditions of OBJ-APPLY-1. We assume that variables of $r : c \to \pi$ have been replaced with fresh ones in order to avoid any conflict to the currently used variables. When $r : c \to \pi$ is applied, a new plan instance is added into the plan state. Its initial plan is $\pi\theta$ and its justifications are specified by $ids(r\theta)$.

When no more object-level rules can be applied, i.e., no object-level rule in R_ρ satisfies the first and the second conditions of OBJ-APPLY-1, the phase counter is advanced to $\rho+1$ and the agent moves the next stage:

$$\frac{\begin{array}{l}\forall(r : c \to \pi) \in R_\rho : \langle A, \Pi, \rho, n\rangle \vdash r, c \mid \theta \\ \quad \Longrightarrow \exists p \in \Pi : \mathsf{init}(p) = \pi \wedge \mathsf{justs}(p) = ids(r\theta)\end{array}}{\langle A, \Pi, \rho, n\rangle \longrightarrow \langle A, \Pi, \rho + 1, n\rangle} \qquad \text{(OBJ-APPLY-2)}$$

Application of meta rules

If R_ρ contains meta rules, actions in the body of a meta rule in R_ρ are executed immediately when the meta rule is applied. The application of a meta rule is defined by the following transition rule:

$$\frac{\begin{array}{l}\exists(c \to ma_1; \ldots; ma_k) \in R_\rho : \langle A, \Pi, \rho, n\rangle \vdash c \mid \theta \\ \langle A, \Pi, \rho, n\rangle \xrightarrow{(ma_1;\ldots;ma_k)\theta} \langle A', \Pi', \rho, n\rangle\end{array}}{\langle A, \Pi, \rho, n\rangle \longrightarrow \langle A', \Pi', \rho, n\rangle} \qquad \text{(META-APPLY-1)}$$

In META-APPLY-1, a meta rule $c \to ma_1; \ldots; ma_k \in R_\rho$ is applicable if its condition c is evaluated to true with respect to the current configuration. This is described in the first condition of META-APPLY-1. Then, the transition from the current configuration to the next one is equivalent to the effect of executing actions in $(ma_1; \ldots; ma_k)\theta$ as defined by Definition 4.2.1.

Similar to the case of object-level rules, when no more meta rules can be applied, i.e., no meta rule in R_ρ satisfies the first condition of (META-APPLY-1), the phase counter is

advanced to $\rho + 1$ and the agent moves to the next stage:

$$\frac{\forall (c \to ma_1; \ldots; ma_k) \in R_\rho : \langle A, \Pi, \rho, n\rangle \not\models c}{\langle A, \Pi, \rho, n\rangle \longrightarrow \langle A, \Pi, \rho + 1, n\rangle} \quad \text{(META-APPLY-2)}$$

4.3.3 The Exec phase

After the Apply phase, the phase counter is $k + 1$ and the agent is in the Exec phase. This phase consists of two stages. In the first stage, $\rho = k + 1$, the state flag `stepped` is removed from plan instances which were executed at the previous cycle. This guarantees that only plan instances which are executed at the current cycle will have the state flag `stepped` in the next cycle.

Then, the transition rule for removing the state flag `stepped` is defined by using the auxiliary function (4.2) as follows:

$$\frac{\Pi' = \{\mathit{del\text{-}flag}(p, \texttt{stepped}) \mid p \in \Pi\}}{\langle A, \Pi, k + 1, n\rangle \longrightarrow \langle A, \Pi', k + 2, n\rangle} \quad \text{(DEL-STEPPED)}$$

In this transition rule, the phase counter is also advanced to $k + 2$.

In the second stage of the Exec phase, $\rho = k + 2$, the agent executes intentions which are scheduled in this deliberation cycle. Recall that an intention is scheduled to execute by setting the state flag `scheduled`. Furthermore, only executable intentions, which have no subgoal, are allowed to execute. This means no intention with a subgoal is executed even if it has the state flag `scheduled`. If an intention with the flag `scheduled` is not executed due to having a subgoal, the flag `scheduled` remains with the intention to the next cycles. When its subgoal is achieved, the intention will be executed.

In the following, we list the transition rules in this stage according to types of actions to be executed.

Executing an external action

External actions are performed in the agent's environment. Changes to the state of the environment by external actions can only be received by the agent through perceptions

at the Sense phase at the beginning of the next deliberation cycle. Here, we assume that each external action can signal whether the action succeeded or failed. In case of success, the execution of an external action is defined by the following transition rule:

$$\frac{\begin{array}{l}\exists p \in \Pi : \texttt{scheduled} \in \mathsf{flags}(p) \wedge \mathsf{plan}(p) = ea; \pi \\ ea(\mathsf{subs}(p)) \text{ is performed successfully} \\ \Pi' = \Pi \setminus \{p\} \cup \{\textit{update-executing}(p, \pi)\}\end{array}}{\langle A, \Pi, k+2, n\rangle \longrightarrow \langle A, \Pi', k+2, n\rangle} \quad \text{(EXEC-EA)}$$

In EXEC-EA, we use the the auxiliary function *update-executing* to describe the change of the plan instance p after executed. In particular, the executed external action is removed by replacing that plan of p with the remainder π. Then, *update-executing* deletes the flag `scheduled` from p (hence, p is not executed again in this phase), and sets the flag `stepped` for p to indicate that the intention is executed in the current deliberation cycle. In case of failure, the execution of an external action is defined by the following transition rule:

$$\frac{\begin{array}{l}\exists p \in \Pi : \texttt{scheduled} \in \mathsf{flags}(p) \wedge \mathsf{plan}(p) = ea; \pi \\ ea(\mathsf{subs}(p)) \text{ fails} \\ \Pi' = \Pi \setminus \{p\} \cup \{\textit{update-failing}(p, \pi)\}\end{array}}{\langle A, \Pi, k+2, n\rangle \longrightarrow \langle A, \Pi', k+2, n\rangle} \quad \text{(FAIL-EA)}$$

Here, we use the auxiliary function *update-failing* to replace the flag `scheduled` by the flag `failed`.

Executing a test action

Mental state tests are evaluated against the current configuration. If the result is true, the resulting substitution is applied to the rest of the plan of the intention. The transition rule for this case is defined as follows:

$$\frac{\begin{array}{l}\exists p \in \Pi : \texttt{scheduled} \in \mathsf{flags}(p) \wedge \mathsf{plan}(p) = ?b; \pi \\ \langle A, \Pi, k+2, n\rangle \vdash b(\mathsf{subs}(p)) \mid \theta \\ \Pi' = \Pi \setminus \{p\} \cup \{\textit{set-subs}(\textit{update-executing}(p, \pi), \theta)\}\end{array}}{\langle A, \Pi, k+2, n\rangle \longrightarrow \langle A, \Pi', k+2, n\rangle} \quad \text{(EXEC-TEST-1)}$$

If the result of a mental state test is false with respect to the current configuration, the execution of the test action fails. This failure is recorded by replacing the flag `scheduled` with the flag `failed`. The transition rule for this case is as follows:

$$\frac{\begin{array}{l}\exists p \in \Pi : \texttt{scheduled} \in \text{flags}(p) \wedge \text{plan}(p) = ?b; \pi \\ \langle A, \Pi, k+2, n\rangle \vdash \text{not } b(\text{subs}(p)) \mid \varnothing \\ \Pi' = \Pi \setminus \{p\} \cup \{\textit{update-failing}(p, ?b; \pi)\}\end{array}}{\langle A, \Pi, k+2, n\rangle \longrightarrow \langle A, \Pi', k+2, n\rangle} \quad \text{(EXEC-TEST-2)}$$

In EXEC-TEST-2, we use the auxiliary function *update-failing* to specify the replacement of the flag `scheduled` with the flag `failed`. When a plan instance has the state flag `failed`, agent programmers are responsible to write meta rules to handle it since **meta-APL** does not implement any mechanism to deal with failed plan instances.

Executing mental state actions

Mental state actions are performed to update the mental state of the current configuration. They can be either primitive mental state actions (`add-atom` and `delete-atom`) or user-defined ones. The effect of executing mental state actions is the same as they are executed when applying a meta rule. The transition rule for executing them is defined as follows:

$$\frac{\begin{array}{l}\exists p \in \Pi : \texttt{scheduled} \in \text{flags}(p) \wedge \text{plan}(p) = ma; \pi \\ \Pi' = \Pi \setminus \{p\} \cup \{\textit{update-executing}(p, \pi)\} \\ \langle A, \Pi', k+2, n\rangle \xrightarrow{ma(\text{subs}(p))} \langle A', \Pi'', k+2, n\rangle\end{array}}{\langle A, \Pi, k+2, n\rangle \longrightarrow \langle A', \Pi'', k+2, n\rangle} \quad \text{(EXEC-META)}$$

Similar to the execution of external actions, the auxiliary function *update-executing* is also used in EXEC-META to update the plan instance p where the plan of p is replaced with the remainder π and the state flag `scheduled` is replaced with `stepped`.

Executing a sub-goal action

The execution of a subgoal action in an intention results in the creation of a new atom instance of the goal atom. The transition rule for executing a subgoal action is defined as

follows:

$$\frac{\begin{array}{l}\exists p \in \Pi : \texttt{scheduled} \in \text{flags}(p) \wedge \text{plan}(p) = !g; \pi \\ i_{\text{new}} \in ID \setminus (ID_A \cup ID_\Pi) \\ A' = A \cup \{(i_{\text{new}}, g(\text{subs}(p)), \text{id}(p), n)\} \\ \Pi' = \Pi \setminus \{p\} \cup \{\textit{update-executing}(p, \pi)\}\end{array}}{\langle A, \Pi, k+2, n\rangle \longrightarrow \langle A', \Pi', k+2, n\rangle} \quad \text{(EXEC-GOAL)}$$

In EXEC-GOAL, the new atom instance $(i_{\text{new}}, g, \text{id}(p), n)$ of the goal atom g is related to the intention by storing the id of p in the parental component of the atom instance. This creates a subgoal relation between the plan instance p with the new atom instance.

Finishing the Exec **phase**

When no more intentions from the plan state can be executed, i.e., no executable intention with the state flag `scheduled`, the phase counter is reverted to 0 and the cycle counter is advanced to $n + 1$:

$$\frac{\forall p \in \Pi : \texttt{scheduled} \in \text{flags}(p) \implies \exists \alpha \in A : \text{par}(\alpha) = \text{id}(p)}{\langle A, \Pi, k+2, n\rangle \longrightarrow \langle A, \Pi, 0, n+1\rangle} \quad \text{(NEW-CYCLE)}$$

In NEW-CYCLE, the condition specifies that the rule is applicable if all intentions which have been scheduled to be executed (i.e., with the flag `scheduled`) have been executed (i.e., their flag `scheduled` is replaced by the flag `stepped`) except those which have subgoals. In other words, we do not execute plan instances which have subgoals even if they are scheduled to be executed. When NEW-CYCLE is applied, the resulting configuration is the beginning one of the next deliberation cycle as its phase counter is 0 and its cycle counter is $n + 1$.

4.3.4 Semantics of agents

The semantics of an agent consists of runs on a (possibly infinite) computation tree derived from the transition rules of **meta-APL**, starting from an initial configuration:

Definition 4.3.1. *Given a **meta-APL** agent* $Ag = (A, Q, M, R_1, \ldots, R_k)$ *and its initial configuration* $C_0 = \langle A_0, \varnothing, 0, 0 \rangle$*, the semantics of* Ag *is a transition system (a computation tree) which is generated by applying transition rules starting in the initial configuration* C_0*.*

Example 4.3.1. We illustrate the operational semantics of **meta-APL** by revisiting our example in Section 3.5 of the previous chapter. The semantics of the agent program in this example is given in Appendix B where we present a computation run consisting of the first two cycles of the agent program.

4.4 Summary

In this chapter, we presented the operational semantics of **meta-APL**. In particular, we discuss the formal definitions of atom instances, plan instances, mental states, plan states and agent configurations. The semantics of an agent in **meta-APL** determines possible computation runs derivable from the set of transition rules. Furthermore, each run consists of deliberation cycles each of which contains $k + 3$ stages where k is the number of rule sets that the agent possesses. Stages are grouped into three phases: the Sense phase, the Apply phase, and the Exec phase. In the Sense phase, the agent updates its mental state according to perceptions received from the environment. In the Apply phase, rules from rule sets are applied to quiescence to generate plan instances. Finally, the agent executes schedules plan instances in the Exec phase.

Chapter 5

Cycle-based Bisimulation

In order to exhibit the flexibility of **meta-APL**, we will show how Jason and 3APL agent programs and their deliberation strategies can be translated into **meta-APL** to give equivalent behaviour in the next two chapters. Our approach to proving the equivalent behaviours of two agents is based on weak bisimulation. This technique was introduced in [Milner, 1989], and applied in [Hindriks, 2001] to comparing the expressiveness of agent programming languages which are accompanied with formal, operational semantics.

As with meta-APL in the previous chapter, many BDI-based agent programming languages (such as Jason, 3APL and **meta-APL**) are associated with a formal semantics which is defined as a transition system whose states correspond to agent configurations and transitions between states are derived from transition rules of the operational semantics of the language. Without loss of generality, we assume that such a transition system is a tree. Two agents whose semantics are defined as transition systems are equivalent under weak bisimulation if there is a weak bisimulation (i.e., a binary relation) between states of these two transition systems.

5.1 Bisimulation

Let us recall the notions of labelled transition systems, strong bisimulation and weak bisimulation from [Milner, 1989].

5.1.1 Labelled transition system

A labelled transition system consists of states and labelled transitions between these states. The semantics of an agent can be seen as a labelled transition system where each state of the labelled transition system corresponds to a configuration of the agent and each transition corresponds to executing an action. An action can be either external or internal. External actions are performed on the environment to modify it, hence they are observable; internal actions happen internally and aim to change mental components of agent states (such as updating beliefs or intentions), hence they cannot be observed. Transitions of external actions are called *visible transitions* and labelled with the name of the external actions; transitions of internal actions are called *silent transitions* and labelled with the symbol τ. In the context of semantics for agent programs, we only consider tree-like labelled transition systems, i.e., between any two states, there is maximally one sequence of transitions.

In this chapter, we use A to denote the set of external actions and $A^\tau = A \cup \{\tau\}$. Formally, a transition system is a pair $(S, \{\xrightarrow{a} | \ a \in A^\tau\})$ where S is a set of states and $\xrightarrow{a}$ is a binary relation on S, i.e., $\xrightarrow{a} \subseteq S \times S$ for all $a \in A^\tau$.

5.1.2 Strong bisimulation

The basic idea of two states being bisimilar is that observations in these two states are equivalent. Observations from a state consists of part of its internal elements (such as beliefs, goals and intentions) and transitions available from the state (such as external actions that can be performed). These observations are called stated-based observations and action-based observations, respectively. State-based observations are defined by an

observation function $observe(s)$ which returns observable internal elements of the state s; while action-based observations return transitions from s to other states. The notion of (strong) bisimulation is formally given below:

Definition 5.1.1 (Strong bisimulation)**.** *Let* $(S, \{\xrightarrow{a} | \ a \in A^\tau\})$ *and* $(T, \{\xrightarrow{a} | \ a \in A^\tau\})$ *be two transition systems. A relation* $\sim \subseteq S \times T$ *is a (strong) bisimulation if for any* $s \sim t$*, it is the case that:*

1. $observe(s) = observe(t)$,
2. *if* $s \xrightarrow{a} s'$*, then there exists* $t' \in T$ *such that* $t \xrightarrow{a} t'$ *and* $s' \sim t'$*, and*
3. *if* $t \xrightarrow{a} t'$*, then there exists* $s' \in S$ *such that* $s \xrightarrow{a} s'$ *and* $s' \sim t'$*.*

5.1.3 Weak bisimulation

In a strong bisimulation of two labelled transition systems, each transition of a system must be simulated by exactly a transition of the other. Hence, the notion of strong bisimulation is too strong for comparing agent programs in different agent programming languages. An alternative approach to comparing agent programs is to consider the notion of weak bisimulation.

Weak bisimulation abstracts from silent transitions τ, which do not exhibit interaction with the environment. For example, executing external actions of an agent which are used to modify the environment of the agent corresponds to visible transitions while updating the agent's beliefs or intentions corresponds to silent transitions. Then, observations over a sequence of transitions only concerns visible transitions. To this end, we define that a label of a sequence of transitions is the sequence of labels from these transitions where the label of an silent transition is the empty sequence. In particular, let $label$ be the function which yields a label for each sequence of transitions. Then, given a sequence of transitions $seq = s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} \ldots \xrightarrow{a_{n-1}} s_n$, $label(seq)$ is defined inductively as follows:

$label(\tau) = \epsilon$ where ϵ is the empty sequence

$label(a) = a$ where $a \in A$

$label(s_1) = \epsilon$ if $n = 1$

$label(s_1 \xrightarrow{a} s_2) = label(a)$ if $n = 2$ where $a \in A^\tau$

$label(s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} \ldots \xrightarrow{a_{n-1}} s_n) = label(s_1 \xrightarrow{a_1} s_2) \cdot label(s_2 \xrightarrow{a_2} \ldots \xrightarrow{a_{n-1}} s_n)$ otherwise

Note that the first case in the definition of the function $label$ (when $n = 1$) caters for the empty sequence of transitions which have only a single state. Then, we have the following definition of abstract transitions:

Definition 5.1.2 (Abstract transitions)**.** *Let $(S, \{\xrightarrow{a}| \ a \in A^\tau\})$ be a transition system. For any s and $s' \in S$, $s \overset{l}{\Longrightarrow} s'$ iff there is a sequence $seq = s_0 \xrightarrow{e_1} s_1 \ldots \xrightarrow{e_k} s_k$ where $k \geq 0$, $s = s_0$, $s' = s_k$, and $label(seq) = l$.*

Then, a weak bisimulation only requires that the abstract transitions of two programs are matched.

Similarly, not all elements of a state is observable. Again, weak bisimulation abstracts from unobservable elements.

Then, the notion of weak bisimulation is given below:

Definition 5.1.3 (Weak bisimulation)**.** *Let $(S, \{\xrightarrow{a}| \ a \in A^\tau\}$ and $(T, \{\xrightarrow{a}| \ a \in A^\tau\}$ be two transition systems. A relation $\cong \subseteq S \times T$ is a weak bisimulation if for any $s \cong t$, it is the case that:*

1. *$observe(s) = observe(t)$,*
2. *if $s \xrightarrow{\tau} s'$, then there exists $t' \in T$ such that $t \overset{\epsilon}{\Longrightarrow} t'$ and $s' \cong t'$; if $s \xrightarrow{a} s'$ where $a \in A$, then there exists $t' \in T$ such that $t \overset{a}{\Longrightarrow} t'$ and $s' \cong t'$, and*
3. *if $t \xrightarrow{\tau} t'$, then there exists $s' \in S$ such that $s \overset{\epsilon}{\Longrightarrow} s'$ and $s' \cong t'$; if $t \xrightarrow{a} t'$ where $a \in A$, then there exists $s' \in S$ such that $s \overset{a}{\Longrightarrow} s'$ and $s' \cong t'$.*

5.2 Cycle-based bisimulation

In **meta-APL** and many BDI-based agent programming languages, an agent is associated with a formal semantics which is a tree-like transition system. Each branch of this transition system corresponds to a possible run of the agent. Each run is composed of consecutive deliberation cycles. For example, deliberation cycles of agents in **meta-APL** start and end with configurations whose phase counter is 0. In order to prove that two agents behave equivalently, we show that there is a weak bisimulation between the semantics of these two agents. Our approach to determining the weak bisimulation is first to define a strong bisimulation between deliberation cycles of in the semantics these agents, hence called *cycle-based* bisimulation, and then to derive a weak bisimulation from this cycle-based bisimulation.

Let ag be an agent and s_0 be its initial configuration which also corresponds to the beginning of a deliberation cycle. Each transition in the semantics of ag is labelled with either an external action a (if it corresponds to the execution of a), or a silent action τ (otherwise). We denote the set of configurations in the semantics of ag, e.g., those reachable from s_0, as $RC(s_0)$.

Let $SC(s_0)$ be a subset of $RC(s_0)$ consisting of configurations which correspond the beginning of a deliberation cycle. We assume that $s_0 \in SC(s_0)$ and each configuration of a BDI-based programming language has a special content marking whether the configuration is the beginning of a deliberation cycle. Then, configurations in $SC(s_0)$ can be determined by a function which looks for special contents in configurations. Then, we define the notion of a deliberation cycle below:

Definition 5.2.1 (Deliberation cycle)**.** *A deliberation cycle of $RC(s_0)$ is a finite sequence of transitions $s_1 \xrightarrow{a_1} \ldots \xrightarrow{a_{n-1}} s_n$ $(n > 1)$ where:*

- *$s_i \in RC(s_0)$ for all $1 \leq i \leq n$,*
- *$a_i \in A^\tau$ for all $1 \leq i < n$,*

- $s_1, s_n \in SC(s_0)$, *and*
- $s_i \notin SC(s_0)$ *for all* $1 < i < n$.

Then, we define the first configuration, the last configuration of a deliberation cycle as follows:

Definition 5.2.2 (First and last configuration)**.** *Let* $c = s_1 \xrightarrow{a_1} \ldots \xrightarrow{a_{n-1}} s_n$ *be a deliberation cycle. We define that* $first(c) = s_1$ *and* $last(c) = s_n$.

A configuration is said to be in a deliberation cycle iff it appears within the deliberation cycle. Formally, we have the following definition:

Definition 5.2.3. *Let* $c = s_1 \xrightarrow{a_1} \ldots \xrightarrow{a_{n-1}} s_n$ *be a deliberation cycle. An arbitrary configuration* s *is in* c*, written as* $s \in c$*, iff* $s = s_i$ *for some* $i \in \{1, \ldots, n\}$.

When $s \in c = s_1 \xrightarrow{a_1} \ldots \xrightarrow{a_{n-1}} s_n$, i.e., $s = s_i$ for some $i \in \{1, \ldots, n\}$, we then write $label(c|s)$ to denote the label of the prefix of c from $first(c)$ until s, i.e., $label(c|s) = label(s_1 \xrightarrow{a_1} \ldots \xrightarrow{a_{i-1}} s_i)$.

We also lift a binary relation R on $X \times Y$ to a relation between subsets of X and subsets of Y, which we will also denote by R. In particular, we define that $X'RY'$ where $X' \subseteq X$ and $Y' \subseteq Y$ iff $\forall\, x \in X'$, $\exists\, y \in Y'$ such that xRy, and $\forall\, y \in Y'$, $\exists\, x \in X'$ such that xRy. Therefore, $X'RY'$ means that any element of X' is related by R to some element of Y' and vice verse.

Let $DC(s_0)$ denote the set of all deliberation cycles of $RC(s_0)$. We define *transitions* between consecutive deliberation cycles in $RC(s_0)$ as follows:

Definition 5.2.4 (Transitions between cycles)**.** *Given* $c, c' \in DC(s_0)$, $c \xrightarrow{l} c'$ *iff* $last(c) = first(c')$ *and* $l = label(c)$.

In the rest of this thesis, we assume that observations are only meaningful for configurations at the beginning of deliberation cycles, i.e., configurations in $SC(s_0)$. This implies for configurations not in $SC(s_0)$, we assume their observation is ignored (and denoted by

$\top$[1]). Then, considering two configurations not in $SC(s_0)$ to be weakly bisimilar is done by only determining the last two conditions of Definition 5.1.3. This relaxation increases the space for differences between agent programming language where a transition can be simulated by a sequence of transitions where internal components of configurations is gradually changed over each transition in the sequence. Formally, we have $observe(s) = \top$ if $s \notin SC(s_0)$. Then, we define the observables of a deliberation cycle as the observables from its first configuration since most other configurations of the deliberation cycle is $\top$:

Definition 5.2.5 (Observables of a deliberation cycle)**.** *Given $c \in RC(s_0)$, $observe(c) = observe(first(c))$.*

Then, we have the following result:

Theorem 5.2.6. *Let s_0 and t_0 be two initial configurations. If there exists a strong bisimulation $\sim \subseteq DC(s_0) \times DC(t_0)$ where, for any $c \sim d$, the following conditions hold:*

1. *$\forall\, s \in c$ where $s \neq last(c)$, $\exists\, t \in d$ such that $t \neq last(d)$, $label(c|s) = label(d|t)$, $observe(s) = observe(t)$ and $\{c' \in DC(s_0) \mid first(c) = first(c'), s \in c'\} \sim \{d' \in DC(t_0) \mid first(d) = first(d'), t \in d'\}$,*

2. *$\forall\, t \in d$ where $t \neq last(d)$, $\exists\, s \in c$ such that $s \neq last(c)$, $label(d|t) = label(c|s)$, $observe(t) = observe(s)$ and $\{c' \in DC(s_0) \mid first(c) = first(c'), s \in c'\} \sim \{d' \in DC(t_0) \mid first(d) = first(d'), t \in d'\}$,*

then, $RC(s_0)$ and $RC(t_0)$ are weakly bisimilar.

Before proving this theorem, let us discuss the intuitive meaning, illustrated in Figure 5.1, of its conditions.

The two conditions are used to establish a weak bisimulation between configurations in c and d. The first condition states that, for any configuration s of c, we can determine

[1] In logic, $\top$ usually denotes a tautology. We use it to indicate that the observation does not produce any information.

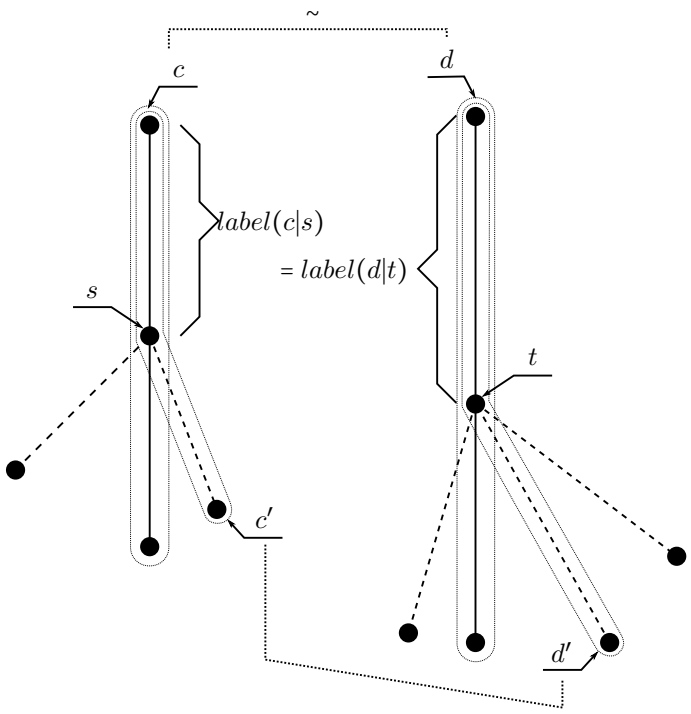

Figure 5.1: Conditions of Theorem 5.2.6.

a configuration t in d such that the label of transitions from $first(c)$ to s is the same as that from $first(d)$ to t, observations of s and t are the same, and for any cycle c' which shares the prefix of c up to s (i.e., $first(c) = first(c')$ and $s \in c'$ [2]), there exists a cycle d' which also shares the prefix of d up to t (i.e., $first(d) = first(d')$ and $t \in d'$) and c' and d' are bisimilar and vice versa. Conversely, the second condition is for determining for any configuration t of d a configuration s of c that satisfies the same condition as above. In the proof, we will relate these configurations to form the weak bisimulation between $RC(s_0)$ and $RC(t_0)$.

Proof.

Firstly, let us construct a binary relation $\cong \subseteq RC(s_0) \times RC(t_0)$ where for any $(s,t) \in RC(s_0) \times RC(t_0)$ we have that $s \cong t$ iff:

[2]Note that all labelled transition systems under consideration are tree-like, i.e., there is at most one path between two configurations.

(Local) $\exists\,(c,d) \in DC(s_0) \times DC(t_0)$ such that $s \in c$, $s \neq last(c)$, $t \in d$, $t \neq last(d)$, $label(c|s) = label(d|t)$, $observe(s) = observe(t)$, $c \sim d$, and

(Global) $\{c' \in DC(s_0) \mid first(c) = first(c'), s \in c'\} \sim \{d' \in DC(t_0) \mid first(d) = first(d'), t \in d'\}$.

Let us now prove that $\cong$ is a weak bisimulation.

Let $s \in RC(s_0)$ and $t \in RC(t_0)$ such that $s \cong t$, $s \xrightarrow{a} s'$, and $a \in A^{\tau}$, we must show that there exists $t' \in RC(t_0)$ such that $t \stackrel{a}{\Longrightarrow} t'$ and $s' \cong t'$. As $s \cong t$, there are $c \in DC(s_0)$ and $d \in DC(t_0)$ satisfying (Local).

Case 1: If $s' \neq last(c)$ and $s = first(c)$, then $t = first(d)$ since the observation of configurations which are after $first(d)$ and before $last(d)$ in d is $\top$ while $observe(s) \neq \top$.

Let c'' be an arbitrary cycle in $\{c' \in DC(s_0) \mid first(c) = first(c'), s' \in c'\}$, we have that s and $s' \in c''$. Then, $c'' \in \{c' \in DC(s_0) \mid first(c) = first(c'), s \in c'\}$. By $s \cong t$ and (Global), there exists $d'' \in \{d' \in DC(t_0) \mid first(d) = first(d'), t \in d'\}$ such that $c'' \sim d''$.

Since $s' \in c''$ and $c'' \sim d''$, by condition (1), there exists $t' \in d''$ such that:

- $label(c''|s') = label(d''|t')$ and $observe(s') = observe(t')$ which imply (Local) for s' and t', and
- $\{c' \in DC(s_0) \mid first(c') = first(c''), s' \in c'\} \sim \{d' \in DC(s_0) \mid first(d') = first(d''), t' \in d'\}$ which implies (Global) for s' and t'.

Hence, $s' \cong t'$. Since t is the first configuration of d, t' is after t in d. As $label(d''|t') = label(c''|s') = label(c|s) \cdot label(a)$ and $label(d|t) = label(c|s)$, the label of the sequence of transitions from t to t' is $label(a)$, hence $t \stackrel{label(a)}{\Longrightarrow} t'$.

Case 2: If $s' \neq last(c)$ and $s \neq first(c)$, then, similar to the previous case, we have that $t \neq first(d)$.

Again, let c'' be an arbitrary cycle in $\{c' \in DC(s_0) \mid first(c) = first(c'), s' \in c'\}$, we have that s and $s' \in c''$. Then, $c'' \in \{c' \in DC(s_0) \mid first(c) = first(c'), s \in c'\}$. By $s \cong t$ and (Global), there exists $d'' \in \{d' \in DC(t_0) \mid first(d) = first(d'), t \in d'\}$ such that $c'' \sim d''$.

Since $s' \in c''$ and $c'' \sim d''$, by condition (1), there exists $t'' \in d''$ such that:

- $label(c''|s') = label(d''|t'')$ and $observe(s') = observe(t'')$ which imply (Local) for s' and t'', and
- $\{c' \in DC(s_0) \mid first(c) = first(c'), s' \in c'\} \sim \{d' \in DC(t_0) \mid first(d) = first(d'), t'' \in d'\}$ which implies (Global) for s' and t''.

Hence, $s' \cong t''$. Furthermore, we also have:

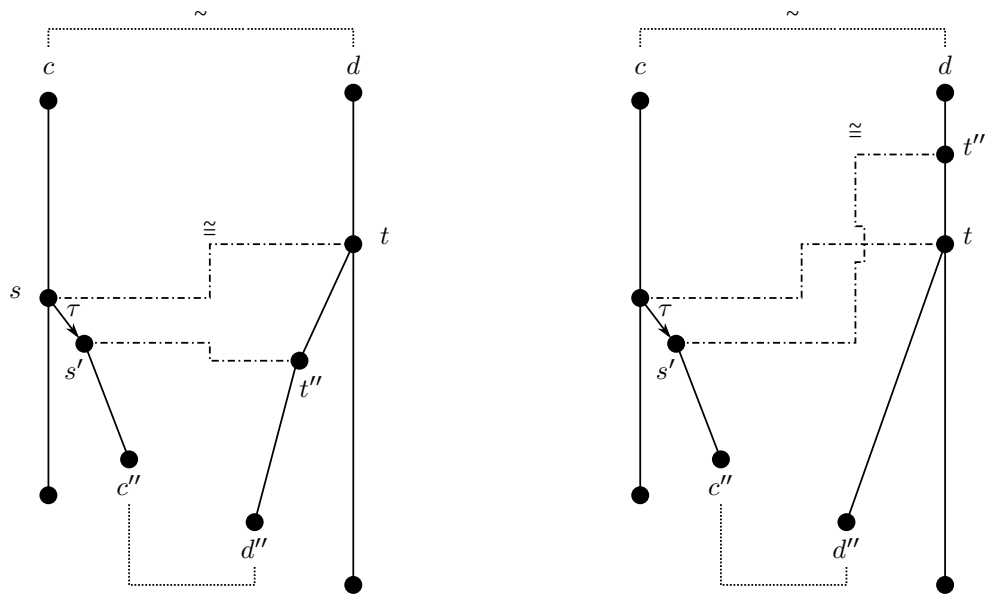

Figure 5.2: There are more than one transitions from t.

- If $a = \tau$ and t'' is after t in d'', as illustrated in Figure 5.2 (a), we select $t' = t''$ and have that $s' \cong t'$ and $t \stackrel{\epsilon}{\Longrightarrow} t'$ (as $label(c''|s') = label(d''|t')$).
- If $a = \tau$ and t'' is before t in d'', as illustrated in Figure 5.2 (b), we select $t' = t$. We have:
  - $label(c''|s') = label(c|s)$ and $label(c''|s') = label(d|t'')$ imply that there are only silent transitions from t'' to t, hence $label(c''|s') = label(d|t)$. Furthermore, since

s' is after s in c'' and t is after t'' in d'', both s' and t are not beginning configurations of a deliberation cycle, i.e., $observe(s') = observe(t) = \top$. Hence, (Local) is true for (s', t).

– For any $e \in \{c' \in DC(s_0) \mid first(c) = first(c'), s' \in c'\}$, $e \in \{c' \in DC(s_0) \mid first(c) = first(c'), s \in c'\}$ since s is before s' in c''. Then, by $s \cong t$ and (Global), there exists $f \in \{d' \in DC(t_0) \mid first(d) = first(d'), t \in d'\}$ such that $e \sim f$. Conversely, for any $f \in \{d' \in DC(t_0) \mid first(d) = first(d'), t \in d'\}$, $f \in \{d' \in DC(t_0) \mid first(d) = first(d'), t'' \in d'\}$ since t'' is before t in d''. Then, by $s' \cong t''$ and (Global), there exists $e \in \{c' \in DC(s_0) \mid first(c) = first(c'), s' \in c'\}$ such that $e \sim f$. Hence, (Global) is true for (s', t)

Hence, $s' \cong t$. Obviously, we also have that $t \stackrel{\epsilon}{\Longrightarrow} t$.

- If a is an external action, t'' must be after t in d'' since $label(c''|s') = label(c|s) \cdot a = label(d|t) \cdot a$. We select $t' = t''$ and have that $s' \cong t'$ and $t \stackrel{a}{\Longrightarrow} t'$.

Case 3: If $s' = last(c)$, then we select $t' = last(d)$. We have $label(c|s') = label(c|s) \cdot label(a) = label(c) = label(d) = label(d|t) \cdot label(a)$ as $c \sim d$ and $s \cong t$; thus, $t \stackrel{label(a)}{\Longrightarrow} t'$.

- Let $c_1 \in DC(s_0)$ be a deliberation cycle such that $c \stackrel{l}{\rightarrow} c_1$. As $c \sim d$, there exists a deliberation cycle $d_1 \in DC(t_0)$ such that $d \stackrel{l}{\rightarrow} d_1$ and $c_1 \sim d_1$. Then, $s' = first(c_1)$ and $t' = first(d_1)$. By $c_1 \sim d_1$ and condition (1) we have that $observe(s') = observe(t'')$ for some $t'' \in d_1$ and $t'' \neq last(d_1)$. As $observe(s') \neq \top$, $t'' = first(d_1) = t'$. Hence $observe(s') = observe(t')$. We also have that $label(c_1|s') = \epsilon = label(d_1|t')$. Therefore, (Local) holds for s' and t'.

- Let $c' \in DC(s_0)$ such that $first(c_1) = first(c')$. Then, we have $c \stackrel{l}{\rightarrow} c'$. Then, $c \sim d$ implies that $d \stackrel{l}{\rightarrow} d'$ and $c' \sim d'$ for some $d' \in DC(t_0)$. Hence, $first(d') = first(d_1)$. By the same argument, it is also straightforward to show that for any $d' \in DC(t_0)$ such that $first(d_1) = first(d')$, there exists $c' \in DC(s_0)$ such that $first(c_1) = first(c')$ and $c' \sim d'$. Therefore, (Global) holds for s' and t'.

Hence, $s' \cong t'$.

Similarly, given $s \in RC(s_0)$ and $t \in RC(t_0)$ such that $s \cong t$ and $t \xrightarrow{a} t'$, we can show that there exists $s' \in RC(s_0)$ such that $s \stackrel{a}{\Longrightarrow} s'$ and $s' \cong t'$. The proof is similar to the one above as the roles of s and t are symmetric. Hence, it is omitted here.

□

5.3 Summary

In this chapter, we recalled the notions of strong bisimulation and weak bisimulation. They are powerful tools for the comparison of expressiveness of programming languages in general and agent programming languages in particular. Two agents behave equivalently if there is a weak bisimulation between their semantics. Here, we show that if there is a so-called cycle-based bisimulation – a strong bisimulation between deliberation cycles in the semantics of the two agents, then we can construct a weak bisimulation between configurations of the two agents. This result will be used in the next two chapters to show that the simulations of Jason and 3APL in **meta-APL** are correct.

Chapter 6

Simulating Jason

In this chapter, we demonstrate the flexibility of **meta-APL** by showing how Jason programs and deliberation strategy can be simulated in **meta-APL**. In particular, we define a translation function for transforming an agent program written in Jason into an agent program in **meta-APL**. Then, we prove that these two programs (the source program in Jason and the target agent in meta-APL) are equivalent using the notion of cycle-based bisimulation.

6.1 Jason

Jason [Bordini et al., 2007] is an interpreter for an extension of the agent programming language AgentSpeak(L) [Rao, 1996]. The main purpose of extending AgentSpeak(L) in Jason is to create a language suitable for the practical needs of the implementation of intelligent agents and multi-agent systems. In particular, as pointed out by [Bordini and Hübner, 2005], the main extensions include: strong negation, default negation (or negation as failure), plan labels, events for handling plan failure and internal actions; although most of them has not been formalised in the semantics. For the purpose of illustrating the simulation of Jason in **meta-APL**, we use the simplified version of Jason presented in [Bordini et al., 2007, Chapter 10] which is accompanied with a formal semantics and covers all

core features and important aspects of the language. In the rest of this chapter, we refer to this simplified version as Jason.

6.1.1 Syntax

An agent program in Jason consists of a belief base and a plan base. The syntax of an agent ag is given below:

$$ag \quad ::= \quad (bs, ps)$$

where bs is a belief base and ps is a plan base.

In the following, we use a to denote an atom as defined in Section 3.2.1. We use b to denote a ground atom, that is:

$$b \quad ::= \quad p(t_1, \ldots, t_n).$$

where p is a predicate of n-arity in the set of predicates *PRED* (see Section 3.2.1) and $t_1, \ldots, t_n$ are ground terms.

The belief base bs is a finite set of beliefs which are ground first order atomic formulas. The syntax of bs is given below:

$$bs \quad ::= \quad b^*$$

The plan base ps is a non-empty finite set of plans. The syntax of a plan base is given below:

$$ps \quad ::= \quad p^+$$

where the syntax of p is defined as:

$$p \quad ::= \quad te : ct \leftarrow h.$$

Here, p consists of a head $te : ct$ where te is a triggering event and ct is a context query, and a plan body h. The triggering event te is either an addition of a belief, a deletion of a

belief, or an addition of a goal. When an agent selects an event to react to, the triggering events of plans are used to determine whether the plans are relevant. The syntax of a triggering event is listed below:

$$te \quad ::= \quad +a \mid -a \mid +!a \mid +?a$$

where a denotes an atom; $+a$ denotes an event of adding a belief a, $-a$ an event of deleting a belief a, $+!a$ an event of adding an achievement goal $!a$, and $+?a$ an event of adding a test goal. The context query ct is a conjunction of atoms and their negation. A relevant plan p can be applied if the context query ct is a logical consequence of the current belief base. The syntax of a context query is listed as follows:

$$ct \quad ::= \quad true \mid a \mid \text{not } a \mid ct_1 \ \& \ ct_2$$

Finally, the plan body h is a sequence of external actions e, subgoals g and belief updates u. Their syntax is given as follows:

$$\begin{aligned} h &::= \epsilon \mid (ea \mid g \mid u \mid); h_1 \\ ea &::= e(t_1, \ldots, t_n) \\ g &::= !a \mid ?a \\ u &::= +b \mid -a \end{aligned} \qquad (n \geq 0)$$

where $ea \in$ *ActionNames* (for performing the external action ea on the environment); the subgoal g is either an achievement goal $!a$ (for generating an event of adding a subgoal) or a test goal $?a$ (for reasoning about the belief base, if the test fails, an event of adding a test goal is generated); and the belief update u is either an addition $+b$ (for adding the belief b into the belief base) or a deletion $-a$ (for deleting beliefs matching a from the belief base).

The syntax presented above does not support events for handling plan failure ("$-!a$" denoting a deletion of an achievement goal and "$-?a$" denoting a deletion of a test goal) and communication actions. We do not consider events for handling plan failure since Jason has not defined formal semantics for them. We also do not consider communication

actions since the purpose of this chapter is to translate single Jason programs into single **meta-APL** programs and hence communication is omitted.

6.1.2 Operational semantics

Informally, a Jason agent runs by generating intentions from its plan base and executing these intentions. The application of plans is triggered by events. An event is a pair $\langle te, i\rangle$ where te is a triggering event and i is either $\top$ or an intention. An event of the form $\langle te, \top\rangle$ is called an *external* event; an event of the form $\langle te, i\rangle$ where i is the intention (which generates the event by performing a subgoal) is called an *internal* event. An intention is a stack of partially instantiated plans where plans in this stack are executed in the order from the top to the bottom. In particular, the intention has the form of $i = [p_1] \ldots [p_n]$ where p_i are partially instantiated plans.

Deliberation cycle of Jason

The generation and execution of intentions follow the deliberation cycle as depicted in Figure 6.1. In this figure, rectangles denote different phases in a deliberation cycle. The double-lined rectangle marks the beginning of the cycle. Arrows illustrate possible transitions from a configuration of a phase to that of another. This deliberation cycle is comprised of the following phases:

1. ProcMsg: this phase processes received messages from other agents and perception from the environment. Since we do not consider communication actions, this phase only updates the belief base of the agent.

2. SelEv: this phase selects an event from the event base to react to. The selection is defined by means of a function S_E which returns an event from a set of events.

3. RelPl: this phase determines all plans from the plan base that are relevant to the selected event. Such a plan has the selected event as its triggering event.

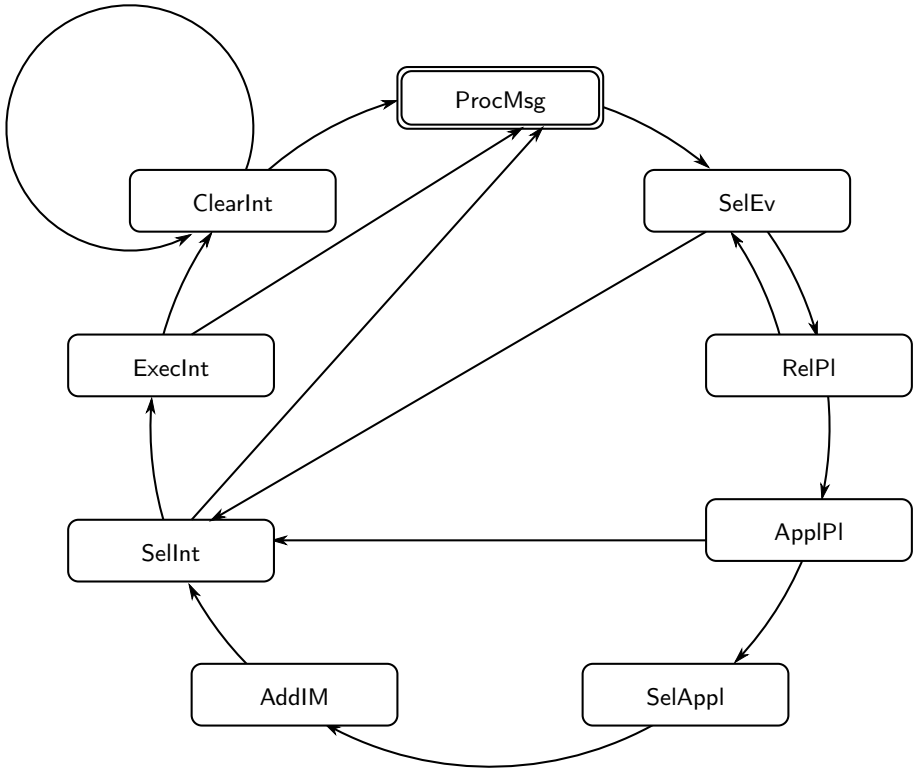

Figure 6.1: The deliberation cycle of Jason.

4. ApplPl: this phase checks which relevant plans are applicable according to the belief base. A relevant plan is applicable if its context query is true in the current belief base of the agent.

5. SelAppl: this phase selects one of the applicable plans to apply. The selection is defined by means of a function S_O[1] which returns a plan from a set of applicable plans.

6. AddIM: this phase applies the selected applicable plan. As a result, a new intended means is added into the set of intentions.

7. SelInt: this phase selects an intention for execution. The selection of intentions is defined by means of a function S_I which returns an intention from a set of intentions.

[1] O stands for options.

8. ExecInt: this phase executes one step of the selected intention.

9. ClrInt: this phase deletes intended means, which has finished, from the set of intentions.

Jason configuration

The operational semantics of Jason is defined by means of transition rules which allow one to derive tree-like transition systems as the semantics of Jason agent programs. Each branch on the tree corresponds to a run of an agent and is comprised of transitions between Jason configurations. A Jason configuration is defined as follows.

Definition 6.1.1 (Jason configurations)**.** *A configuration of a Jason agent is a tuple* $\langle ag, C, T, s \rangle$ *where:*

- $ag = (bs, ps)$ *is an agent program which consists of a belief base* bs *and a plan base* ps *(as defined by the syntax of Jason);*
- C *is a circumstance which is a triple* (I, E, A) *where:*
  - I *is a set of intentions;*
  - E *is a set of events; and*
  - A *is a set of executed external actions;*
- T *is a tuple* (R, Ap, i, ev, p) *which stores temporary information, where:*
  - R *is a set of relevant plans;*
  - Ap *is a set of applicable plans;*
  - i *is a selected intention;*
  - ev *is a selected event; and*
  - p *is a selected applicable plan;*

- $s \in \{\mathit{ProcMsg}, \mathit{SelEv}, \mathit{RelPl}, \mathit{ApplPl}, \mathit{SelAppl}, \mathit{AddIM}, \mathit{SelInt}, \mathit{ExecInt}, \mathit{ClrInt}\}$ *is a phase indicator.*

In a Jason configuration $\langle ag, C, T, s\rangle$ of an agent, only the plan base ps is unchanged during the execution the agent.

Logical consequences are defined as follows. An atom a is a logical consequence of a belief base bs (a set of positive literals) iff there exists $b \in bs$ such that $a\theta = b$ for some most general unifier θ. Then, we write $bs \models a\theta$. Conversely, we define $bs \models$ not a iff $bs \not\models a$. For a conjunction c of atoms and their negation, i.e., $c = l_1 \wedge \ldots \wedge l_n$ where l_i is either an atom or atom's negation, we write that $bs \models c$ iff $bs \models l_i$ for all i.

The set A consists of actions to be carried out in the environment. Therefore, in order to execute an external action, it is added into A. This addition tells the effector to perform the added action in the environment.

Elements of T hold temporary information during the execution of an agent. For example, when the agent selects and event to react to, it stores this event into ev which will be used in subsequent phases. For convenience, we adopt the following subscript notation to refer to components of ag, C and T:

- ag_{bs} is the belief base bs of the agent ag,
- C_I, C_E and C_A refer to I, E and A, respectively, and
- T_R, T_{Ap}, T_i, T_e and T_p refers to R, Ap, i, ev and p, respectively. To indicate that no intention has been selected, we write $T_i = \bot$. Similarly, we write $T_e = \bot$ and $T_p = \bot$ to indicate that no event and no applicable plan is selected, respectively.

In the operational semantics of Jason, plans from ps relevant to a selected event ev are those whose triggering events match ev. Firstly, Jason defines a function $TrEv$ which extracts the triggering event from a plan $TrEv(te : ct \leftarrow h) = te$, and $Ctxt$ which extracts the context query from a plan $Ctxt(te : ct \leftarrow h) = ct$. Then, the set of these relevant plans

is given as follows:

$$RelPlans(ps, ev) = \{(p, \theta) \mid p \in ps \wedge \theta \text{ is a substitution s.t. } ev = TrEv(p) \mid \theta\}$$

Then, plans from a set R of relevant plans are applicable if their contexts are the logical consequences of the belief base bs. The set of applicable plans is given as follows:

$$ApplPlans(bs, R) = \{(p, \theta_1\theta_2) \mid \exists (p, \theta_1) \in R \wedge \theta_2 \text{ is a substitution s.t. } bs \models (Ctxt(p)\theta_1)\theta_2\}$$

where $\theta_1\theta_2$ denotes the composition θ_1 and θ_2.

In the beginning, a Jason agent has no intentions, events or executed external actions. It also does not have any temporary information. Let $ag = (bs, ps)$ be an agent program in Jason, then the initial configuration of this agent is given as $\langle ag, C_0, T_0, \mathsf{ProcMsg}\rangle$ where $C_0 = (\varnothing, \varnothing, \varnothing)$ and $T_0 = (\varnothing, \varnothing, \bot, \bot, \bot)$.

Transition rules

An agent in Jason runs by transiting from one configuration to another, starting from the initial configuration. The transitions between configurations are specified by transition rules.

At the beginning of a cycle, a Jason agent is in the ProcMsg phase where its belief base is updated according to perception received from the environment. This update is carried out by the *belief update function* which is called buf. The transition rule to update the belief base is given as follows:

$$\frac{\begin{array}{rcl} ag'_{bs} & = & buf(env, ag_{bs}) \\ C'_E & = & C_E \cup \{\langle +b, \top\rangle \mid b \in ag'_{bs} \setminus ag_{bs}\} \cup \{\langle -b, \top\rangle \mid b \in ag_{bs} \setminus ag'_{bs}\} \end{array}}{\langle ag, C, T, \mathsf{ProcMsg}\rangle \rightarrow \langle ag', C', T, \mathsf{SelEv}\rangle} \quad \text{(ProcMsg)}$$

In this rule, apart from changes in the belief base, events about these changes (addition and deletion of beliefs) are also added into the event base.

In the SelEv phase, the agent selects an event from the event base. If this is possible,

i.e., the event base is not empty, the selection is given by the following transition rule:

$$\frac{C_E \neq \varnothing \wedge T'_e = S_E(C_E) \wedge C'_E = C_E \setminus \{T'_e\}}{\langle ag, C, T, \mathsf{SelEv}\rangle \rightarrow \langle ag, C', T', \mathsf{RelPl}\rangle} \quad \text{(SelEv-1)}$$

Otherwise, the agent skips the selection and moves to the SelInt phase to select an intention for execution:

$$\frac{C_E = \varnothing}{\langle ag, C, T, \mathsf{SelEv}\rangle \rightarrow \langle ag, C, T, \mathsf{SelInt}\rangle} \quad \text{(SelEv-2)}$$

In the RelPl phase, the agent finds all plans from the plan base which are relevant to the selected event. If there are such plans, the agent moves to the next phase to determine applicable plans from the relevant ones:

$$\frac{RelPlans(ag_{ps}, T_e) \neq \varnothing \wedge T'_R = RelPlans(ag_{ps}, T_e)}{\langle ag, C, T, \mathsf{RelPl}\rangle \rightarrow \langle ag, C, T', \mathsf{ApplPl}\rangle} \quad \text{(Rel-1)}$$

Otherwise, the agent returns to the previous phase (i.e., SelEv) to select another event:

$$\frac{RelPlans(ag_{ps}, T_e) = \varnothing}{\langle ag, C, T, \mathsf{RelPl}\rangle \rightarrow \langle ag, C, T, \mathsf{SelEv}\rangle} \quad \text{(Rel-2)}$$

This means if the event base has no event which has relevant plans, eventually, the agent uses (SelEv-2) to go to the SelInt phase. If the event base has some event which has relevant plans, eventually, the agent uses (Rel-1) to go to the ApplPl phase.

In the ApplPl phase, the agent determines the set of applicable plans from T_R which are relevant to the selected event T_e. If this set is not empty, it is stored in T_{Ap}. This is given by the following transition rule:

$$\frac{ApplPlans(ag_{bs}, T_R) \neq \varnothing \wedge T'_{Ap} = ApplPlans(ag_{bs}, T_R)}{\langle ag, C, T, \mathsf{ApplPl}\rangle \rightarrow \langle ag, C, T', \mathsf{SelAppl}\rangle} \quad \text{(Appl-1)}$$

Otherwise, the agent moves directly to the SelInt phase:

$$\frac{ApplPlans(ag_{bs}, T_R) = \varnothing}{\langle ag, C, T, \mathsf{ApplPl}\rangle \rightarrow \langle ag, C, T, \mathsf{SelInt}\rangle} \quad \text{(Appl-2)}$$

Note that if all relevant plans are not applicable, the agent does not go back to the SelEv phase to select a different event. Instead, it carries on to the SelInt phase.

In the SelAppl phase, the agent selects one of the applicable plans in T_{Ap} to apply. The selected applicable plan is temporarily stored in T_p for the next phase:

$$\frac{S_O(T_{Ap}) = (p, \theta) \wedge T'_p = (p, \theta)}{\langle ag, C, T, \mathsf{SelAppl}\rangle \rightarrow \langle ag, C, T', \mathsf{AddIM}\rangle} \quad \text{(SelAppl)}$$

If the selected event T_e is internal, this selected applicable plan is put on top of the intention which generates the selected event:

$$\frac{T_e = \langle e, i\rangle \wedge T_p = (p, \theta) \wedge C'_I = C_I \cup \{i[p\theta]\}}{\langle ag, C, T, \mathsf{AddIM}\rangle \rightarrow \langle ag, C', T, \mathsf{SelInt}\rangle} \quad \text{(IntEv)}$$

Otherwise, the selected applicable plan forms a new intention:

$$\frac{T_e = \langle e, \top\rangle \wedge T_p = (p, \theta) \wedge C'_I = C_I \cup \{[p\theta]\}}{\langle ag, C, T, \mathsf{AddIM}\rangle \rightarrow \langle ag, C', T, \mathsf{SelInt}\rangle} \quad \text{(ExtEv)}$$

In both transition rules above, the resulting configurations are in the SelInt phase where the agent selects an intention for execution. If the intention base is not empty, the intention selection is given by the following transition rule:

$$\frac{C_I \neq \varnothing \wedge T'_i = S_I(C_I)}{\langle ag, C, T, \mathsf{SelInt}\rangle \rightarrow \langle ag, C, T', \mathsf{ExecInt}\rangle} \quad \text{(SelInt-1)}$$

Otherwise, the agent finishes the current cycle and starts a new one as follows:

$$\frac{C_I = \varnothing}{\langle ag, C, T, \mathsf{SelInt}\rangle \rightarrow \langle ag, C, T, \mathsf{ProcMsg}\rangle} \quad \text{(SelInt-2)}$$

In the ExecInt phase, the agent executes the selected intention. This means executing the first action in the top plan of the intention. Depending on the type of this action, we have the following transition rules specifying the effect of its execution.

If the action is an external action, we have:

$$\frac{\begin{array}{rcl} T_i & = & i[head \leftarrow ea; h] \\ C'_I & = & C_I \setminus \{T_i\} \cup \{i[head \leftarrow h]\} \\ C'_A & = & C_A \cup \{ea\} \end{array}}{\langle ag, C, T, \mathsf{ExecInt}\rangle \rightarrow \langle ag, C', T, \mathsf{ClrInt}\rangle} \quad \text{(Action)}$$

Note that $C'_A = C_A \cup \{ea\}$ means that the agent sends the action to the effector so that it will be performed on the environment.

If the action is a subgoal, we have:

$$\frac{\begin{array}{rcl} T_i & = & i[head \leftarrow !g; h] \\ C'_I & = & C_I \setminus \{T_i\} \\ C'_E & = & C_E \cup \{\langle +!g, T_i\rangle\} \end{array}}{\langle ag, C, T, \mathsf{ExecInt}\rangle \rightarrow \langle ag, C', T, \mathsf{ClrInt}\rangle} \quad \text{(AchvGl)}$$

If the action is a test goal and the test goal is true with respect to the belief base, we have:

$$\frac{\begin{array}{rcl} T_i & = & i[head \leftarrow ?g; h] \\ ag_{bs} & \models & g\theta \\ C'_I & = & C_I \setminus \{T_i\} \cup \{i[(head \leftarrow h)\theta]\} \end{array}}{\langle ag, C, T, \mathsf{ExecInt}\rangle \rightarrow \langle ag, C', T, \mathsf{ClrInt}\rangle} \quad \text{(TestGl-1)}$$

If the test goal is not true, we have

$$\frac{\begin{array}{rcl} T_i & = & i[head \leftarrow ?g; h] \\ ag_{bs} & \not\models & g \\ C'_E & = & C_E \cup \{\langle +?g, T_i\rangle\} \\ C'_I & = & C_I \setminus \{T_i\} \end{array}}{\langle ag, C, T, \mathsf{ExecInt}\rangle \rightarrow \langle ag, C', T, \mathsf{ClrInt}\rangle} \quad \text{(TestGl-2)}$$

If the action is to add a belief, we have

$$\frac{\begin{array}{rcl} T_i & = & i[head \leftarrow +b; h] \\ ag'_{bs} & = & ag_{bs} \cup \{b\} \\ C'_E & = & C_E \cup \{\langle +b, \top\rangle\} \\ C'_I & = & C_I \setminus \{T_i\} \cup \{i[head \leftarrow h]\} \end{array}}{\langle ag, C, T, \mathsf{ExecInt}\rangle \rightarrow \langle ag', C', T, \mathsf{ClrInt}\rangle} \quad \text{(AddBel)}$$

Recall that b denotes a ground atom.

If the action is to delete beliefs, we have

$$\frac{\begin{array}{rcl} T_i & = & i[head \leftarrow -at; h] \\ ag'_{bs} & = & ag_{bs} \setminus \{b \mid \exists\theta : b = at\theta\} \\ C'_E & = & C_E \cup \{\langle -at, \top\rangle\} \\ C'_I & = & C_I \setminus \{T_i\} \cup \{i[head \leftarrow h]\} \end{array}}{\langle ag, C, T, \mathsf{ExecInt}\rangle \rightarrow \langle ag', C', T, \mathsf{ClrInt}\rangle} \quad \text{(DelBel)}$$

Recall that at denotes an atom which may have variables. Therefore, the effect of the action $-at$ is to delete all beliefs which match with at.

After executing the selected intention, the agent is in the ClrInt phase. In this phase, the agent cleans the executed intention if its plan is empty.

If the executed intention contains only one plan which is empty, this intention is deleted from the intention base:

$$\frac{\begin{array}{rcl} T_i & = & [head \leftarrow \epsilon] \\ C'_I & = & C_I \setminus \{T_i\} \end{array}}{\langle ag, C, T, \mathsf{ClrInt}\rangle \rightarrow \langle ag, C', T, \mathsf{ProcMsg}\rangle} \quad \text{(ClrInt-1)}$$

If the executed intention has more than one plan and the top plan is empty, this top plan is removed from the intention:

$$\frac{\begin{array}{rcl} T_i & = & i[head' \leftarrow !g; h][head \leftarrow \epsilon] \\ \theta & \text{is} & \text{a substitution s.t. } +!g = TrEv(head \leftarrow \epsilon) \mid \theta \\ C'_I & = & C_I \setminus \{T_i\} \cup \{i[(head' \leftarrow h)\theta]\} \end{array}}{\langle ag, C, T, \mathsf{ClrInt}\rangle \rightarrow \langle ag, C', T, \mathsf{ClrInt}\rangle} \quad \text{(ClrInt-2)}$$

In the final case, when the executed intention is not empty, no cleaning is needed, the agent starts a new cycle as follows:

$$\frac{T_i \neq [head \leftarrow \epsilon] \wedge T_i \neq i[head \leftarrow \epsilon]}{\langle ag, C, T, \mathsf{ClrInt}\rangle \rightarrow \langle ag, C, T, \mathsf{ProcMsg}\rangle} \quad \text{(ClrInt-3)}$$

6.1.3 Selections in a deliberation cycle

A deliberation cycle of a Jason agent is a sequence of transitions $s_0 \xrightarrow{a_1} s_1 \ldots \xrightarrow{a_k} s_k$ where two end configurations s_0 and s_k are of the phase ProcMsg and transitions are derived from the transition rules of Jason operational semantics. We will characterise a deliberation cycle of a Jason agent in terms of selections (of an event, plan, or intention) performed during the cycle.

The first selection is for an event which is made from a configuration whose phase indicator is SelEv. At this configuration, there is either a transition labelled with (SelEv-1)

or another with (SelEv-2). The latter transition corresponds to the case when the event base is empty, thus, no event is selected in this deliberation cycle. Otherwise, the former transition selects an event from the event base and outputs a configuration with the RelPl phase where relevant plans from the plan base are determined. If no such plan exists, the next transition is (Rel-2) which returns to the SelEv phase in order to select another event, and then (SelEv-1) is repeated. This repetition only terminates when either an event which has some relevant plans is selected, where there is a transition (Rel-2) following (SelEv-1) and the selected event is called a relevant event, or there is no event left to select, where there is a transition (SelEv-2). Hence, we define:

Definition 6.1.2. *Let* $c = s_0 \xrightarrow{a_1} s_1 \ldots \xrightarrow{a_k} s_k$ *be a deliberation cycle of a Jason agent* ag*. Then,*

- c *selects a relevant event* ev *if there is a transition* a_i *= (Rel-1) for some* $1 \leq i \leq k$ *where* $ev = T_e$ *and* $s_{i-1} = (ag, C, T, \mathsf{RelPl})$*,*
- c *does not select any relevant event if there is a transition* a_i *= (SelEv-2) for some* $1 \leq i \leq k$*.*

The second selection is for an applicable plan. It is made from a configuration of the SelAppl phase by a transition label (SelAppl). In order to arrive at the SelAppl phase, there must be a transition (Appl-1) from a configuration of the ApplPl phase just before (SelAppl) where the set of applicable plans is determined to be non-empty. Otherwise, there is a transition labelled (Appl-2) instead of (Appl-1) which means that no applicable plan is selected in this deliberation cycle. Hence, we define:

Definition 6.1.3. *Let* $c = s_0 \xrightarrow{a_1} s_1 \ldots \xrightarrow{a_k} s_k$ *be a deliberation cycle of a Jason agent* ag*. Then,*

- c *selects an applicable plan* ap *if there is a transition* a_i *= (SelAppl) for some* $1 \leq i \leq k$ *where* $ap = T_p$ *and* $s_i = (ag, C, T, \mathsf{AddIM})$*,*
- c *does not select any applicable plan if there is a transition* a_i *= (Appl-2) for some* $1 \leq i \leq k$*.*

The last selection is for an intention to execute. It is made from a configuration of the SelInt phase. If there is an intention to select, the selection for an intention is carried

out by a transition labelled with (SelInt-1) from this configuration. Otherwise, there is a transition (SelInt-2) from this configuration instead of (SelInt-1). Hence, we define:

Definition 6.1.4. *Let* $c = s_0 \xrightarrow{a_1} s_1 \ldots \xrightarrow{a_k} s_k$ *be a deliberation cycle of a Jason agent* ag*. Then,*

- c *selects an intention* int *to execute if there is a transition* a_j *= (SelInt-1) for some* $1 \leq j \leq k$ *where* $int = T_i$ *and* $s_j = (ag, C, T, \mathsf{ExecInt})$*,*
- c *does not select any intention to execute if there is a transition* a_j *= (SelInt-2) for some* $1 \leq j \leq k$*.*

6.2 Translation

We define a translation function to translate a Jason agent program into a **meta-APL** agent program such that two agents are equivalent under the notion of weak bisimulation.

6.2.1 Outline of the translation

We define the translation function based on correspondences between phases of Jason's deliberation cycles and **meta-APL**'s deliberation cycles. These correspondences are illustrated as dashed, two-ended arrows in Figure 6.2.

The first correspondence is between the ProcMsg phase of Jason's deliberation cycle and the Sense phase of **meta-APL**'s deliberation cycle where agents update their belief bases according to perception received from the environment. Note that ProcMsg generates events about changes in a belief base and so does Sense (see the transition rule Sense in Section 4.3.1). Furthermore, completed intentions are cleared just before the ProcMsg phase of Jason's deliberation cycles while **meta-APL**'s operational semantics defines that the Sense phase follows the Exec phase immediately. This means it is necessary for the translation to clear completed intentions as soon as possible after Exec. This is done by suitable meta rules which delete completed intentions. Finally, right after the ProcMsg phase, Jason's deliberation cycle has the SelEv phase which eventually selects an event

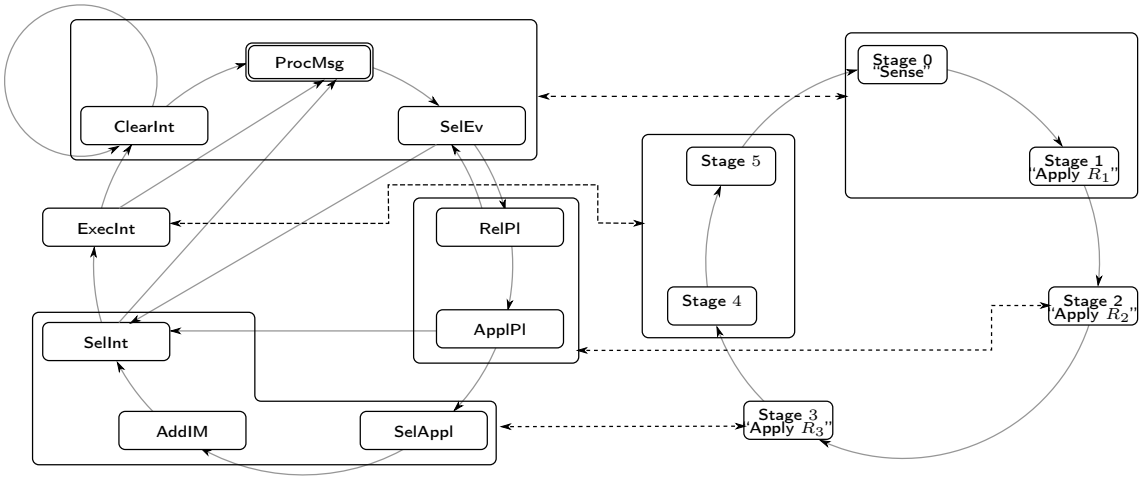

Figure 6.2: The correspondence between Jason's and **meta-APL**'s deliberation cycles.

relevant to some plan (if a non-relevant event is selected, the Jason agent has to return to SelEv to select another event). We will also simulate this by means of meta rules which which look for relevant events specified by plans in the plan base of the Jason agent. The meta rules for cleaning completed intention and selecting relevant events form the rule set R_1 in the translation.

The next correspondence is about generating relevant and applicable plans. In Jason's deliberation cycle, these are done in two phases RelPl and ApplPl where relevant plans and applicable plans are generated from the plan base and the selected event. They can be simulated by means of object-level rules in **meta-APL** where each plan in the Jason plan base is translated into an object-level rule. These object-level rules comprise the next rule set R_2.

Then, the third correspondence is between the selection of an applicable plan to add into the intention base and the selection of an intention for execution. In Jason, these are done by a combination of three phases SelAppl, AddIM and SelInt. We simulate them by a meta rule which promotes one of the new plan instances generated by the object level rules in R_2 to be intended and another meta rule which selects one of intentions to be

executed in this cycle. These meta rules comprise the rule set R_3.

We define a translation function tr_{Jason}, illustrated in Figure 6.3 that translates a Jason agent program into a **meta-APL** one. The function is defined in terms of three component functions:

- tr_{bel} which translates beliefs into atoms,
- tr_{rel} which extracts relevant events from plans,
- tr_{plan} which translates plans into object-level rules.

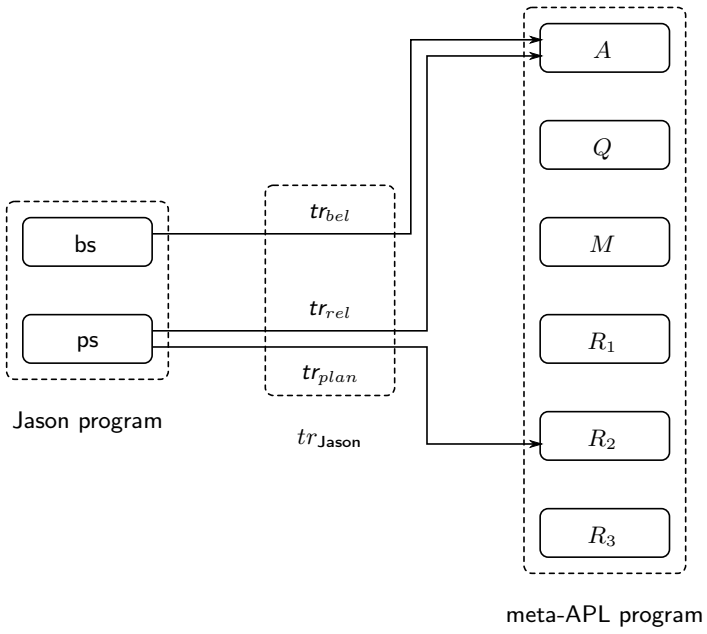

Figure 6.3: The translation function tr_{Jason}.

The result of tr_{Jason} is an agent program which includes an initial atom base A, a set Q of clauses for defining additional queries, a set M of macros for defining additional meta actions, and three rule sets R_1, R_2 and R_3. The elements Q, M, R_1 and R_3 are common to all Jason agent programs. In contrast, A and R_2 depends on the initial belief base and the plan base of a Jason agent program. In particular, A is obtained by translating beliefs in

the initial belief base and extracting relevant events from plans in the plan base; and R_2 is obtained by translating plans in the plan base into object-level rules.

6.2.2 The static part of the translation

Defining additional queries

The set Q contains additional queries which will be used to construct meta rules and object-level rules in the translation. In particular, we define queries: "relevant-event" which checks if an atom is relevant; "selected-event" which retrieves the selected event of the current deliberation cycle; "trigger-event" which checks if an event is relevant; "plan-at" and "intention-at" which check if a plan instance or an intention is created at a cycle, respectively; and "executable-intention" which checks if an intention has no subgoal. These queries are defined below:

$$\text{relevant-event}(E) \leftarrow \text{atom}(\_, relevantEvent(E)) \tag{6.1}$$

$$\text{selected-event}(I) \leftarrow \text{cycle}(N), \text{atom}(\_, selectedEvent(I, N)) \tag{6.2}$$

$$\text{trigger-event}(I) \leftarrow \text{atom}(I, E), \text{relevant-event}(E), \text{not justification}(\_, I) \tag{6.3}$$

$$\text{plan-at}(I, N) \leftarrow \text{plan}(I, \_), \text{cycle}(I, N) \tag{6.4}$$

$$\text{intention-at}(I, N) \leftarrow \text{plan-at}(I, N), \text{state}(I, \texttt{intended}) \tag{6.5}$$

$$\text{executable-intention}(I) \leftarrow \text{state}(I, \texttt{intended}), \text{not subgoal}(I, \_) \tag{6.6}$$

The query relevant-event is defined by the query (6.1) where an atom E is relevant if there is an an instance of the atom $relevantEvent(E)$. Recall that in Jason an event is relevant if it can be unified with the triggering event of a plan in the plan base. Therefore, in the translation, we transform triggering events of plans in the plan base into instances of atoms $relevantEvent(E)$. This transformation is defined by a translation function call tr_{rel} defined in Section 6.2.3.

The query selected-event is defined by clause (6.2) where we first get the number N of the current deliberation cycle by the query cycle(N) and then check if there is an instance

of the atom $selectedEvent(I, N)$. The existence of such an instance means that an event with id I is selected in cycle N. This will be encoded by the meta rule (6.12) in R_1.

The query trigger-event is defined by clause (6.3) where we use the query relevant-event to check if an event is relevant and the query justification to check that the event has not been used to generate an intention.

The queries plan-at and intention-at are defined by clauses (6.4) and (6.5) where we query the cycle at which a plan instance or an intention is created. They are used to form meta rule (6.13) in R_3.

The query executable-intention is defined by clause (6.6) where we use the query state to check if a plan instance is an intention, i.e., it has the flag `intended`, and the query subgoal to check if this plan instance has no subgoal.

Defining additional meta-actions

We define in the translation two additional meta actions which simulate actions of adding and deleting belief of Jason, i.e., actions of the form $+b$ and $-a$. Therefore, the transitions (AddBel) and (DelBel) in Jason will be simulated by the transition (EXEC-META) of these two additional meta actions, respectively. These meta actions take the side effects of $+b$ and $-a$ into account where events of belief addition and belief deletion are generated into the event base. They are defined in M as follows:

$$\texttt{add-belief}(B) = \texttt{add-atom}(I, belief(B)), \texttt{add-atom}(J, +belief(B)) \quad (6.7)$$

$$\texttt{del-belief}(A) = \texttt{delete-atom}(I, belief(A)), \texttt{add-atom}(J, -belief(A)) \quad (6.8)$$

In macro (6.7), the meta action `add-belief`(B) add an instance of the atom $belief(B)$ into the atom base in order to simulate that a belief of B is added into the belief base of the Jason agent. Besides, an instance of the event $+belief(B)$ describing the event of adding a belief is also added.

Similarly, in macro (6.8), the meta action `del-belief`(A) removes instances of atoms $belief(A)$. Also, an instance of the event $-belief(B)$ describing the event of removing

beliefs is also added.

Note that both additional atoms $+belief(B)$ and $-belief(A)$ are added to simulate the belief addition and deletion events in Jason. The role of these events will be finished when they are selected (and deleted) in the operational semantics of Jason (see rule (SelEv-1)). In our translation, the role of these atoms $+belief(B)$ and $-belief(A)$ are finished when they are selected (see rule (6.12)).

Defining meta rules for R_1

R_1 contains meta rules which are responsible for removing non-intended plan instances from the previous cycle, clearing completed intentions, and selecting a relevant event for generating a plan instance in this deliberation cycle. These meta rules are defined below:

$$\text{plan}(I, \_), \text{not state}(I, \texttt{intended}) \rightarrow \texttt{delete-plan}(I) \tag{6.9}$$

$$\text{executable-intention}(I), \text{plan}(I, \epsilon), \text{justification}(I, J), \text{not subgoal}(\_, J) \rightarrow \texttt{delete-atom}(J) \tag{6.10}$$

$$\text{executable-intention}(I), \text{plan}(I, \epsilon), \text{justification}(I, J), \text{subgoal}(K, J), \text{substitution}(I, S) \rightarrow \texttt{set-substitution}(K, S), \texttt{delete-atom}(J) \tag{6.11}$$

$$\text{cycle}(N), \text{not selected-event}(\_), \text{trigger-event}(I) \rightarrow \texttt{add-atom}(selectedEvent(I, N)) \tag{6.12}$$

The meta rule (6.9) is for deleting unintended plan instances. First, it queries plan instances which do not have the flag `intended` by the query "plan" and "state". Then, it deletes these plan instances.

The two next meta rules simulate the transition rules (ClrInt-1) and (ClrInt-2) for clearing completed intentions, respectively. First, they query completed intentions, i.e., executable intentions whose plans are empty. Recall that an intention is executable if it does not have any subgoal. If a completed intention with id I is for reacting to an external event, i.e., not a subgoal of another intention, the meta rule (6.10) clears the completed

intention with id I by removing the instance of this external event. In other words, this is a translation for the transition rules (ClrInt-1) where an intention is deleted from the intention base. Otherwise, if a completed intention with id I is for reacting to an an internal event, i.e., a subgoal of another intention with id K, the meta rule (6.11) clears the completed intention by first extending the substitution of the intention with id K with that of the intention with id I and removing the instance of the internal event. This is a translation for the transition rule (ClrInt-2) where the top plan is removed from the intention and the substitution (obtained by unifying the triggering event of the top plan and the achievement goal in the next plan in the intention) is applied to this next plan. Here, some variables in the triggering event of the top plan may be instantiated as the result of executing the top plan (such as by some test goals); these instantiations are then propagated into the next plan by applying the substitution to the next plan.

Finally, the meta rule (6.12) is for selecting a relevant event, i.e., it simulates the transition (SelEv-1) in Jason. Here, we assume that the Jason function S_E selects events nondeterministically, and that (6.12) relies on this assumption. It first queries the number N of the current cycle, makes sure that no relevant event has been selected by using the query "selected-event" defined by (6.2) and queries for a relevant event with id I. The selection is done in (6.12) by adding an instance of the atom $selectedEvent(I, N)$. Other selection functions S_E can be encoded by modifying (6.12).

Defining meta rules for R_3

R_3 contains meta rules which select one plan instance to become an intention, select one intention to execute in this deliberation cycle, and revise a test action to a corresponding subgoal action if it is scheduled to be executed. These meta rules are as follows:

$$\text{cycle}(N), \text{plan-at}(I, N), \text{not intention-at}(\_, N) \rightarrow \texttt{set-state}(I, \texttt{intended}) \quad (6.13)$$

$$\text{not state}(\_, \texttt{scheduled}), \text{executable-intention}(I), \text{not state}(I, \texttt{failed}) \rightarrow \texttt{set-state}(I, \texttt{scheduled}) \quad (6.14)$$

$$\text{state}(I, \texttt{scheduled}), \text{plan}(I, ?q; P), \text{not } q \rightarrow \texttt{set-plan}(I, !(+test(q)); P) \quad (6.15)$$

The meta rule (6.13) sets a new intention by first querying a plan instance with id I which is generated in the current cycle via the query "plan-at". Therefore, it simulates the effect of the sequence of two transitions (SelAppl) and either (IntEv) or (ExtEv), depending on whether the selected event is a subgoal or not. This meta rule also makes sure that no intention has been generated in the current cycle using the query "intention-at". Then, this meta rule sets the flag `intended` of the plan instance with id I.

The meta rule (6.14) selects an intention for execution by looking for an executable intention with id I, i.e., having no subgoal. Then, this meta rule sets the flag `scheduled` of the plan instance with id I. This is equivalent to the transition (SelInt-1) in Jason.

The meta rule (6.15) revises a test action of an intention which is selected to be executed into a subgoal action when the test action fails. This implements the way where Jason deals with failed tests (defined in the transition rule (TestGl-2)) where a failed test goal gives rise to an event of the form $+?q$. In our translation, such an event is translated into $+test(q)$.

6.2.3 Component translation functions

Translating a belief tr_{bel}

Each belief in the belief base is wrapped in the predicate $belief$ by tr_{bel} as follows:

$$tr_{bel}(b) = belief(b)$$

Extracting relevant events by tr_{rel}

The triggering event of a plan in Jason determines which event is relevant to the plan. We extract this by tr_{rel} as follows:

$$tr_{rel}(te : ct \leftarrow h) = relevantEvent(tr_{event}(te))$$

where the translation of the triggering event te depends on the type of te and is given as follows:

$$\begin{aligned} tr_{event}(+a) &= +belief(a) \\ tr_{event}(-a) &= -belief(a) \\ tr_{event}(+!a) &= +goal(a) \\ tr_{event}(+?a) &= +test(a) \end{aligned}$$

Translating a plan by tr_{plan}

Each plan in a plan base is translated into an object-level rule, i.e., $tr_{plan}(te : ct \leftarrow h)$ gives the following rule:

$$atom(I, tr_{event}(te)) : \text{selected-event}(I), tr_{query}(ct) \rightarrow tr_{body}(h)$$

where I is a fresh variable which does not appear in $te : ct \leftarrow h$; the translation of the context ct is defined as follows:

$$\begin{aligned} tr_{query}(\top) &= \top \\ tr_{query}(a) &= belief(a) \\ tr_{query}(\text{not } a) &= \text{not } belief(a) \\ tr_{query}(ct_1 \& ct_2) &= tr_{query}(ct_1), tr_{query}(ct_1) \end{aligned}$$

and the translation of the plan body h is defined defined inductively on the length of h as follows:

$$
\begin{aligned}
tr_{body}(\epsilon) &= \epsilon \\
tr_{body}(ea) &= ea \quad \text{if } ea \text{ is an external action} \\
tr_{body}(!a) &= !(+goal(a)) \\
tr_{body}(?a) &= ?(belief(a)) \\
tr_{body}(+b) &= \texttt{add-belief}(b) \\
tr_{body}(-a) &= \texttt{del-belief}(a) \\
tr_{body}(h_1; h_2) &= tr_{body}(h_1); tr_{body}(h_2)
\end{aligned}
$$

The translation function tr_{Jason}

Finally, we combine tr_{bel}, tr_{rel} and tr_{plan} to define the translation function tr_{Jason}. Given a Jason program $ag = (bs, ps)$, we define $tr_{\text{Jason}}(ag) = (A, Q, M, R_1, R_2, R_3)$ where Q, M, R_1, R_3 are defined in Section 6.2.2 and

- $A = \{tr_{bel}(b) \mid b \in bs\} \cup \{tr_{rel}(p) \mid p \in ps\}$, and
- $R_2 = \{tr_{plan}(p) \mid p \in ps\}$.

Note that, in the above description of the translation from Jason to **meta-APL**, we have mentioned how transitions in Jason's operational semantics are simulated. Figure 6.4 summarises these simulations. In this summary, we don't mention the cases of (SelEv-2), (Rel-2), (Appl-2) and (SelInt-2) as their simulations are more involved. In particular, for (SelEv-2) and (Rel-2), we record all relevant events in atoms of the form $relevantEvent(e)$ as the plan base of a Jason agent is translated by the function tr_{rel}. This helps the translation agent in **meta-APL** avoid selecting non-relevant event. Later in the proof, we also ignore relevant events when comparing configurations of the Jason and **meta-APL** agents since they do not create new intentions, and hence, do not contribute to the behaviour of the agents via actions they performed. For (Appl-2) and (SelInt-2), they are implicitly

Transition	Simulated by
ProgMsg	SENSE
SelEv-1	META-APPLY-1 which applies (6.12)
Rel-1, Appl-1	OBJ-APPLY-1's which apply $tr_{plan}(r)$ where r's are both relevant and applicable
SelAppl, IntEv, ExtEv	META-APPLY-1 which applies (6.13), non-selected plan instances are cleaned by (6.9) in the next cycle
SelInt-1	META-APPLY-1 which applies (6.14)
Action	EXEC-EA
AchvGl	EXEC-GOAL
TestGl-1	EXEC-TEST-1
TestGl-2	META-APPLY-1 which applies (6.15)
AddBel, DelBel	EXEC-META which executes additional meta actions (6.7) and (6.8)
ClrInt-1	META-APPLY-1 which applies (6.10)
ClrInt-2	META-APPLY-1 which applies (6.11)
ClrInt-3	NEW-CYCLE

Figure 6.4: The simulation of Jason transitions in the translation.

simulated by the transitions (OBJ-APPLY-2) and (META-APPLY-2), respectively. When (Appl-2) is enabled, this means no plan in the plan base of the Jason agent is applicable, this also means none of their translations by the function tr_{plan} is applicable. Hence, in the phase when object level rules in R_2 are considered to be applied, the transition (OBJ-APPLY-2) is enabled and it simulates (Appl-2), while no (OBJ-APPLY-1) can be performed. When (SelInt-2) is enabled, this means there is no intention to select; therefore, the meta rule (6.14) is not applicable. Then, in the phase when meta rules in R_3 are considered to be applied, only (META-APPL-2) is applicable and it simulates (SelInt-2).

6.2.4 Simulating selections

A deliberation cycle of a **meta-APL** agent is a sequence of transitions $s_0 \xrightarrow{a_1} s_1 \ldots \xrightarrow{a_k} s_k$ where configurations s_0 and s_k have the Sense phase, i.e., their phase counter is 0. In the translation of Jason, the selections of a relevant event, an applicable plan and an intention to execute of a Jason deliberation cycle are simulated by applying the meta rule (6.12) in R_1, the meta rule (6.13) in R_3, and the meta rule (6.14) in R_3, respectively. These meta rules are applied by transition labels (META-APPLY-1). Hence, we have the following definition:

Definition 6.2.1. *Let $c = s_0 \xrightarrow{a_1} s_1 \ldots \xrightarrow{a_k} s_k$ be a deliberation cycle of a translation of a Jason agent in **meta-APL**. Then,*

- *c selects a relevant event ev if there is a transition a_i = (META-APPLY-1), for some $1 \leq i \leq k$, of applying the meta rule* (6.12) *where $I = id(a)$ for some atom instance a of s_i and $ev = atom(a)$.*
- *c does not select any relevant event if there is no transition (META-APPLY-1) of applying* (6.12) *in the cycle.*
- *c selects an applicable plan ap if there is a transition a_i = (META-APPLY-1), for some $1 \leq i \leq k$, of applying the meta rule* (6.13) *where $I = id(p)$ for some plan instance p of s_i and $ap = plan(p)$.*
- *c does not select any applicable plan if there is no transition (META-APPLY-1) of applying* (6.13) *in the cycle.*
- *c selects an intention int to execute if there is a transition a_i = (META-APPLY-1), for some $1 \leq i \leq k$, of applying the meta rule* (6.14) *where $I = id(p)$ for some plan instance p of s_i and $int = plan(p)$.*
- *c does not select any intention to execute if there is no transition (META-APPLY-1) of applying* (6.14) *in the cycle.*

6.3 Equivalence of $tr_{\textbf{Jason}}$

In this section, we show that the translation into **meta-APL** of a Jason agent simulates the behaviour of the Jason agent under the notion of weak bisimulation. To this end, we show that there is a strong bisimulation between deliberation cycles of the Jason agent and that of the translation. Furthermore, this bisimulation satisfies the two conditions specified in Theorem 5.2.6 and, therefore, its existence entails a weak bisimulation between two agents.

6.3.1 Observations

First of all, let us define observable properties of configurations of Jason agents and **meta-APL** agents with respect to the translation tr_{Jason}. Given a Jason configuration $s = \langle ag, C, T, p\rangle$, we stipulate that observation of s is possible when s has the ProcMsg phase. Then, observation of s consists of beliefs in the belief base, relevant events in the event base, and incomplete intentions in the intention base as follows:

$$\begin{array}{lcll} observe(s) & = & \top & \text{if } p \neq \mathsf{ProcMsg} \\ observe(s) & = & (Bs, Es, Is) & \text{if } p = \mathsf{ProcMsg} \end{array}$$

where Bs, Es, Is are defined as follows

- $Bs = \{belief(b) \mid b \in ag_{bs}\}$;
- $Es = \{tr_{trigger}(te, i) \mid \langle te, i\rangle \in C_E \wedge RelPlans(ag_{ps}, \langle te, i\rangle) \neq \varnothing\}$ where the function $tr_{trigger}$ converts Jason triggering events into a interleave sequence (starting with a subgoal) of **meta-APL**-like subgoals and **meta-APL**-like intentions, and is defined below:
  - $tr_{trigger}(te, \top) = tr_{event}(te)$;
  - $tr_{trigger}(+!g, [te : ct \leftarrow !g'; h]) = tr_{event}(+!g) \rightarrow tr_{body}(h) \rightarrow tr_{event}(te)$;
  - $tr_{trigger}(+!g, i[te : ct \leftarrow !g'; h]) = tr_{event}(+!g) \rightarrow tr_{body}(h) \rightarrow tr_{trigger}(te, i)$.

- $Is = \{tr_{int}(i) \mid i \in C_I\}$ where the function tr_{int} converts Jason intention into an interleaved sequence (starting with an intention) of **meta-APL**-like subgoals and **meta-APL**-like intentions, and is defined below:
  - $tr_{int}([te : ct \leftarrow h]) = tr_{body}(h)$;
  - $tr_{int}(i[te : ct \leftarrow h]) = tr_{body}(h) \rightarrow tr_{trigger}(te, i)$.

Given a **meta-APL** configuration $t = \langle A, \Pi, \rho, n \rangle$, we stipulate that observation of t is possible if the value of the stage counter is 0 (i.e., of the Sense phase). Then, observation of t consists of instances of beliefs and relevant events in the atom base, and incomplete intentions in the intention base as follows:

$$\begin{array}{lcll} observe(t) & = & \top & \text{if } \rho \neq 0 \\ observe(t) & = & (Bs, Es, Is) & \text{if } \rho = 0 \end{array}$$

where Bs, Es, Is are defined as follows:

- $Bs = \{belief(b) \mid \exists a \in A : \text{atom}(a) = belief(b)\}$.
- $Es = \{cv_{event}(a, t) \mid a \in A, RelPlans(ag_{ps}, \langle tr^{-1}_{event}(\text{atom}(a)), \top \rangle) \neq \varnothing \wedge \neg \exists p \in \Pi : \text{id}(a) \in \text{justs}(p)\}$ where the conversion functions cv_{event} and cv_{int} simplify instances of events and intentions and are defined as follows:
  - $cv_{event}(a, t) = \text{atom}(a)$ if $\neg \exists p \in \Pi : \text{id}(p) = \text{par}(a)$;
  - $cv_{event}(a, t) = \text{atom}(a) \rightarrow cv_{int}(p, t)$ if $\exists p \in \Pi : \text{id}(p) = \text{par}(a)$;
  - $cv_{int}(p, t) = \text{plan}(p)\theta \rightarrow cv_{event}(a, t)$ where $\theta = \text{subs}(p)$ and $a \in A$ such that $\text{id}(a) \in \text{justs}(p)$;

  Recall that functions $\text{atom}(a)$, $\text{plan}(p)$ and subs(p) were defined in Page 70.
- $Is = \{j \mid \exists i \in \Pi : (\neg \exists a \in A : \text{id}(p) = \text{par}(a)) \wedge j = clear(cv_{int,subst}(i, t)) \wedge j \neq \epsilon\}$ where the function $cv_{int,subst}(i, t)$ is defined similar to $cv_{int}(i, t)$ except keeping the substitutions along with plan bodies as follows:

- $cv_{int,subst}(p,t) = (\text{plan}(p), \text{subs}(p)) \rightarrow cv_{event,subst}(a,t)$ where $a \in A$ such that $\text{id}(a) \in \text{justs}(p)$;
- $cv_{event,subst}(a,t) = \text{atom}(a)$ if $\neg\exists p \in \Pi : \text{id}(p) = \text{par}(a)$;
- $cv_{event,subst}(a,t) = \text{atom}(a) \rightarrow cv_{int,subst}(p,t)$ if $\exists p \in \Pi : \text{id}(p) = \text{par}(a)$;

and the functions $clear$ is used for clearing completed intentions and is defined – similarly to Jason's transition rules (ClrInt-1) and (ClrInt-2) – as follows:

$$
\begin{aligned}
clear((\epsilon,\theta) \rightarrow e) &= \epsilon \\
clear((\epsilon,\theta) \rightarrow e_0 \rightarrow (\pi_0,\theta_0) \rightarrow e_1) &= \pi_0\theta_0\theta \rightarrow e_1 \\
clear((\epsilon,\theta) \rightarrow e_0 \rightarrow (\pi_0,\theta_0) \rightarrow e_1 \rightarrow (\pi_1,\theta_1) \ldots \rightarrow e_n) &= \pi_0\theta_0\theta \rightarrow e_1 \rightarrow \pi_1\theta_1 \ldots \rightarrow e_n \\
clear((\pi,\theta) \rightarrow e_0 \rightarrow (\pi_1,\theta_1) \rightarrow \ldots \rightarrow e_n) &= \pi\theta \rightarrow e_0 \rightarrow \pi_1\theta_1 \ldots \rightarrow e_n \text{ if } \pi \neq \epsilon
\end{aligned}
$$

6.3.2 Equivalence

Theorem 6.3.1. *Given a Jason agent $ag = (bs, ps)$, let $(A, Q, M, R_1, R_2, R_3) = tr_{Jason}(ag)$ be its translation in Meta-APL, we have that (bs, ps) and (A, Q, M, R_1, R_2, R_3) are weakly bisimilar.*

Proof.

Let s_0 be the initial configuration of the Jason agent (bs, ps).

Let t_0 be the initial configuration of the translated agent (A, Q, M, R_1, R_2, R_3).

We construct in this proof a strong bisimulation, denoted by ~, between deliberation cycles of ag and $tr_{\text{Jason}}(ag)$, which satisfies conditions (1) and (2) of Theorem 5.2.6. This strong bisimulation ~ is constructed inductively with the help of an auxiliary binary relation $\sim_{\mathsf{Sense}}$ between configurations of phases ProcMsg and Sense in $RC(s_0)$ and $RC(t_0)$, respectively.

Let $c \in DC(s_0)$ be a deliberation cycle of ag and $d \in DC(t_0)$ be a deliberation cycle of $tr_{\text{Jason}}(ag)$.

We say that:

- c and d select the same relevant event if c selects a relevant event $\langle e, i\rangle$ and d selects a relevant event a where $tr_{event}(e)$ = atom(a),
- c and d select the same applicable plan if c selects an applicable plan $head \leftarrow h$ and d selects an applicable plan p where $tr_{body}(h)$ = plan(p),
- c and d select the same intention to execute if c selects an intention $i[head \leftarrow h]$ and d selects an intention p where $tr_{body}(h)$ = plan(p).

Then, we say that c and d select bisimilar items if they select (i) the same relevant event or no relevant event, (ii) the same applicable plan or no applicable plan, and (iii) the same intention to execute or no intention to execute.

Given two configurations $s \in RC(s_0)$ and $t \in RC(t_0)$, we define the set of pairs of deliberation cycles which select bisimilar items and start from s and t, respectively, as follows:

$$eq(s,t) = \{(c,d) \in DC(s_0) \times DC(t_0) \mid first(c) = s, first(d) = t, \\ c \text{ and } d \text{ select bisimilar items}\}$$

Below, we define binary relations:

- $\sim^k$, where $k \in \mathbb{N}$, between deliberation cycles of the Jason agent and the **meta-APL** agent (intuitively, if we number deliberation cycles in a run starting from $1, 2, \ldots$, then $\sim^k$ relates k^{th} deliberation cycles);
- $\sim^k_{\mathsf{Sense}}$, where $k \in \mathbb{N}$, between configurations of the Jason agent and the **meta-APL** agent (intuitively, $\sim^k_{\mathsf{Sense}}$ relates the beginning configurations of $(k+1)^{th}$ deliberation cycle); and
- $\sim$ between deliberation cycles of the Jason agent and the **meta-APL** agent.

These binary relation is defined inductively as follows:

$$\sim^0 = \varnothing$$

$$\sim^0_{\mathsf{Sense}} = \{(s_0, t_0)\}$$

$$\sim^{n+1} = \{(c,d) \in DC(s_0) \times DC(t_0) \mid \exists\, s \sim^n_{\mathsf{Sense}} t : (c,d) \in eq(s,t)\}$$

$$\sim^{n+1}_{\mathsf{Sense}} = \{(s,t) \in RC(s_0) \times RC(t_0) \mid \exists\, c \sim^{n+1} d : last(c) = s \wedge last(d) = t\}$$

$$\sim = \bigcup_{n \geq 0} \sim^n$$

In the following, we establish and prove Claims 6.3.2, 6.3.3, 6.3.4, 6.3.5, and 6.3.6. The results from these claims will be used to prove that the binary relation $\sim$ constructed above satisfies the conditions of Theorem 5.2.4.

Claim 6.3.2. *Let $s \in RC(s_0)$ be a configuration in the ProcMsg phase of ag and $t \in RC(t_0)$ be a configuration in the Sense phase of $tr_{Jason}(ag)$, if $observe(s) = observe(t)$ and $(c,d) \in eq(s,t)$, then $observe(last(c)) = observe(last(d))$.*

Proof. As $(c,d) \in eq(s,t)$, c and d select the same items, i.e., they select the same event, the same applicable plan, and the same intention to execute. We show that $observe(last(c)) = observe(last(d))$ by analysing the change of configurations along c and d.

If a relevant event e is selected by c, then phases RelPl and ApplPl generates all relevant plans and applicable plans from the selected events. Then, d also selects the same event e which generates all plan instances which correspond to applicable plans generated in c.

If c selects an applicable plan ap in SelAppl, then d also selects the corresponding plan instance and sets it to become an intention.

If c selects the intention i to execute, then d also selects the corresponding intention i to execute. Then, they both perform the same actions and produce the same effect on the environment (if the action is an external one) or on the mental state of the internal state (if the action is an internal action).

Thus, changes occurred between s and $last(c)$ are equivalent to those between t and $last(d)$:

- For any new (or deleted) belief in $last(c)$, it must be caused by either a new perception or the effect of performing a belief update action. Equivalently, its corresponding atom instance is also new in $last(d)$ because we have the same new perception or the same belief update action performed resulting the same effect to the atom base.

- For any new intention in $last(c)$ which is created by a plan p, then the corresponding intention is also created in $last(d)$ by $tr_{plan}(p)$.

Therefore, $observe(last(c)) = observe(last(d))$. □

Claim 6.3.3. *Given* $(s,t) \in \sim^n_{Sense}$, *then* $observe(s) = observe(t)$.

Proof. The base case is trivial.

In the induction step, assume $s \sim^{n+1}_{\mathsf{Sense}} t$. Then, there exists $c \sim^{n+1} d$ such that $last(c) = s$ and $last(d) = t$. Then, there exists $s' \sim^n_{\mathsf{Sense}} t'$ such that $(c,d) \in eq(s',t')$. By induction hypothesis, we have that $s' \sim^n_{\mathsf{Sense}} t'$ implies $observe(s') = observe(t')$. Then, by Claim 6.3.2, we have that $observe(last(c)) = observe(last(d))$, i.e., $observe(s) = observe(t)$. □

Claim 6.3.4. *If* $(c,d) \in eq(s,t)$, *then* $label(c) = label(d)$.

Proof. As $(c,d) \in eq(s,t)$, they select bisimilar items.

- If c contains a transition corresponding to the execution of an external action a, as other transitions are silent, then, $label(c) = a$. Since c and d select bisimilar items, d also executes a. Thus, $label(d) = a$. Hence, $label(c) = label(d)$.

- If c does not contain any transition corresponding to the execution of an external action, all transitions in c are silent. Then, $label(c) = \epsilon$. Since c and d select bisimilar items, d does not execute any external action. Thus $label(d) = \epsilon$. Hence, $label(c) = label(d)$.

□

Claim 6.3.5. *For any $s \sim^n_{\mathsf{Sense}} t$ and any cycle c from s, there exists a cycle d from t such that $(c, d) \in eq(s, t)$ and $observe(last(c)) = observe(last(d))$.*

Proof. We shall construct d along transitions of c.

If c selects a relevant event ev, this event is either already in the event base prior to c or a new event (caused by changes in the belief base according to the update by the transition (ProcMsg) at the beginning of c). In the former case, as $s \sim^n_{\mathsf{Sense}} t$, by Claim 6.3.3, $observe(s) = observe(t)$. Thus, there is an atom instance a such that atom$(a) = tr_{event}(e)$ in t. In the latter case, changes in the belief base after the transition (SENSE) give rise to atom instances of events about these changes by the definition of the transition rule (SENSE). Therefore, there is also a new atom instance a such that atom$(a) = tr_{event}(e)$ after the phase of applying meta rules in R_1 from t. Therefore, there is a deliberation cycle d_e from t which applies the meta rule (6.12) to a and generates an instance of the atom $selectedEvent(\mathrm{id}(a), N)$ where N is the current cycle counter. Then d is constructed from the beginning of d_e to the configuration obtained by applying the meta rule (6.12). Note that during this beginning of d_e, the meta rules (6.9), (6.10) and (6.11) may have been applied. (6.9) clears intentions corresponding to applicable plans from the previous cycle which are not selected. (6.10) clears intentions which are completely executed and are not justified by some subgoals; this implements the transition rules (ClrInt-1) of Jason. Finally, (6.11) clears intentions which are completely executed and are justified by some subgoals; this implements the transition rules (ClrInt-2) of Jason where the substitution of the complete intentions are used to extend the substitution of the direct parental intentions. Although in Jason, this is done at the end of the previous cycle of c, this does not effect the equivalence between observations at the beginning configurations of c and d as in the way we define the function $observe(t)$, we ignore the intentions removed by (6.9), (6.10) and (6.11) already.

Conversely, if c does not select any relevant event, there are no deliberation cycles from t which apply the meta rule (6.12). Let d_e be an arbitrary deliberation cycle from t. Then

d is constructed from the beginning of d_e to the configuration obtained by the transition labelled (META-APPLY-2) in the phase of applying meta rules in R_1.

If c selects an applicable plan ap, this applicable plan is one of the applicable plans aps generated by the selected event. Since d (constructed so far) also has the same beliefs (as $observe(s) = observe(t)$) and selects the same event as c, we also obtain plan instances by applying object-level plans which are the translation of the applicable plans aps. Hence, there is a deliberation cycle d_a extending d and containing the transition (META-APPLY-1) of applying the meta rule (6.13) to the plan instance by applying the translation of ap. Then, the construction of d continues with d_e until the transition (META-APPLY-1) where (6.13) is applied.

If c does not select any applicable plan, then there is no applicable plan generated in this cycle because either no relevant event is selected or the selected relevant event has no applicable plan. In both cases, no object-level rules in R_2 can be applied on d (constructed so far), hence, there is no cycle which extends d and has the transition (META-APPLY-2) of applying the meta rule (6.13). Let d_a be an arbitrary cycle extending d, the construction of d continues with d_e until the transition (META-APPLY-2), where the phase of applying object-level rules in R_2 completes.

If c selects an intention int to execute, this intention either exists at the beginning of c or is generated from the selected applicable plan in c. In both cases, the equivalent intention i of int exists in the last configuration of d constructed so far as they have the same intention at the beginning of c and d (as $observe(s) = observe(t)$) and select the same applicable plan. Hence, there exists a deliberation cycle d_i extending d and containing the transition (META-APPLY-1) of applying the meta rule (6.14) to the intention i equivalent to int, i.e., $tr_{int}(i) = int$. Then, the construction of d completes by d_i. As d_i selects the same intention to execute as c, it executes the same action and causes equivalent changes to the environment (in case of an external action) or to the atom base (otherwise). However, if the intention int begins with a test goal which will fail according to the belief base, there

is a transition (META-APPLY-1) of applying the meta rule (6.15) in d_i, which replaces the test action $?belief(b)$ by a subgoal action $! + test(b)$. Hence, by executing this subgoal action, a new atom instance of $+test(b)$ is generated. This simulates the effect of failing a test goal in Jason as described by the transition (TestGl-2).

However, if c does not select any intention to execute, this means that there is no intention at the beginning of c and no applicable plan is selected in c. Then, as $observe(s) = observe(t)$, d has no executable intentions at the beginning and also no new executable intention is generated in d, as constructed so far. Hence, any cycle that extends d_i cannot select an intention to execute. We complete the construction of d by d_i. Then, in d, changes to the atom base are made in phases Sense and of applying R_1 to add new instances of events.

According to the construction of d, we have that c and d select bisimilar items. Hence, $(c, d) \in eq(s, t)$. Then, by Claim 6.3.2, $observe(last(c)) = observe(last(d))$. □

Applying an analogous argument, we can also show the following result:

Claim 6.3.6. *For any $s \sim^n_{Sense} t$ and any cycle d from t, there exists a cycle c from s such that $(c, d) \in eq(s, t)$ and $observe(last(c)) = observe(last(d))$.*

Finally, we show that $\sim$ defined above is a strong bisimulation satisfying Conditions (1) and (2) of Theorem 5.2.6.

Let $c \sim d$, then there exists $n > 0$ such that $c \sim^n d$, then there exists $s \sim^{n-1}_{\mathsf{Sense}} t$ such that $(c, d) \in eq(s, t)$. Note that in the following argument, we ignore the condition on observations as it is straightforward by the result of Claim 6.3.3.

We show that $\sim$ is a bisimulation:

- We have that

$$\begin{aligned} observe(c) &= observe(first(c)) = observe(s) \\ observe(d) &= observe(first(d)) = observe(t) \end{aligned}$$

By Claim 6.3.3, we also have $observe(s) = observe(t)$. Hence, $observe(c) = observe(d)$.

- Assume that $c \xrightarrow{l} c'$ where $l = label(c)$, we have $last(c) = first(c')$. Obviously, $c \sim^n d$ implies $last(c) \sim^n_{\mathsf{Sense}} last(d)$ by the definition of $\sim^n_{\mathsf{Sense}}$. By Claim 6.3.5, there is a cycle d' from $last(d)$ such that $(c', d') \in eq(last(c), last(d))$, i.e., $c' \in \sim^{n+1} d'$. Then, we have that $c' \sim d'$. As $first(d') = last(d)$ and $label(c) = label(d) = l$ (by Claim 6.3.4), we also have that $d \xrightarrow{l} d'$.

- Similarly, we also have that if $d \xrightarrow{l} d'$ where $l = label(d)$, by Claim 6.3.6, there is a cycle c' such that $c \xrightarrow{l} c'$ and $c' \sim d'$.

We show that $\sim$ satisfies conditions (1) and (2) of Theorem 5.2.6:

- For (1): let s' be a configuration along c and $s' \neq last(c)$. Note that c fixes the selections of a relevant event, an applicable plan, and an intention to be executed.

  In the following, we denote $K = \{c' \in DC(s_0) \mid first(c) = first(c'), s' \in c'\}$ and $H = \{d' \in DC(t_0) \mid first(d) = first(d'), t' \in d'\}$.

  - If $s' = first(c)$, we select $t' = first(d)$.

    As $c \sim^n d$, $first(c) \sim^{n-1}_{\mathsf{Sense}} first(d)$, i.e., $s' \sim^{n-1}_{\mathsf{Sense}} t'$. By Claim 6.3.3, $observe(s') = observe(t')$.

    For any $c' \in K$, by Claim 6.3.5, there is a d' in $DC(t_0)$ such that $first(d') = t'$ and $(c', d') \in eq(s', t')$. Then, $c' \sim^n d'$, i.e., $c' \sim d'$. Thus, $d' \in H$.

    Similarly, we have for any $d' \in H$ there exists $c' \in K$ such that $c' \sim d'$.

    Thus, $K \sim H$.

  - If $s' \neq first(c)$ and appears in c before the relevant event is selected, cycles through s' correspond to all possible selections of relevant events, applicable plans to apply and intentions to execute. We choose t' to be a configuration between $first(d)$ and the configuration immediately after the transition (META-APPLY-1) for (6.12) (hence $t' \neq last(d)$). Then, cycles through t' correspond to all possible selections of relevant events, applicable object-level rules and intentions. Hence, we have that $observe(s') = observe(t') = \top$.

As no external action is selected at s' and t', $label(c'|s') = label(d'|t') = \epsilon$.

Let $c' \in K$, then $first(c) = first(c')$. By Claim 6.3.5, there exists $d'' \in DC(t_0)$ such that $first(d'') = first(d)$ and $(c', d'') \in eq(first(c), first(d))$.

* If $t' \in d''$ (i.e., $t' \in$ both d and d'', then we select $d' = d''$, hence $d' \in H$ and $(c', d') \in eq(first(c), first(d))$. Thus, by definition of $\sim^n$ and $\sim$, we obtain $c' \sim^{n+1} d'$ and $c \sim d$.
* If $t' \notin d''$, we construct d' which has the same selections as d'' yet passing t'.

  Let t''_1 and t'_1 be the configurations right before the transition (META-APPLY-1) for (6.12) in d'' and d, respectively. Then, atom instances representing events in t''_1 are the same as in t'_1. Thus, there is a deliberation cycle d'_1 which passes t'_1 (hence also t') and selects the same event as d''.

  Similarly, let t''_2 and t'_2 be the configurations right before the transition (META-APPLY-1) for (6.13) in d'' and d'_1, respectively. Then, new plan instances representing applicable plans in t''_2 are the same as in t'_2. Thus, there is a deliberation cycle d'_2 which passes t'_2 (hence also t') and selects the same applicable plan as d''.

  Similarly, let t''_3 and t'_3 be the configurations right before the transition (META-APPLY-1) for (6.14) in d'' and d'_2, respectively. Then, plan instances representing intentions in t''_3 are the same as in t'_3. Thus, there is a deliberation cycle d'_3 which passes t'_3 (hence also t') and selects the same intention to execute as d''.

  Then, we select $d' = d'_3$ and have that $(c', d') \in eq(first(c), (d))$ and $first(d) = first(d')$. As $t' \in d'$, $d' \in H$.

Thus, for any $c' \in K$ there exists $d' \in H$ such that $c' \sim d'$.

Similarly, we have for any $d' \in H$ there exists $c' \in K$ such that $c' \sim d'$. Hence, $K \sim H$.

– Similarly, if s' appears in c after the relevant event is selected but before the applicable plan is selected to apply, cycles through s' correspond to all possible selections of applicable plans to apply and intentions to execute. We choose t' to be the configuration in d resulting from the transition (META-APPLY-1) of applying the meta rule (6.12) in R_1 (hence $t' \neq last(d)$). This transition selects the relevant event, which is the translation of the selected event in c. Then, cycles through t' correspond to all possible selections of applicable plans and intentions. Hence, we have that $observe(s') = observe(t') = \top$, $label(c|s') = label(d|t') = \epsilon$. Then, applying the same argument as the previous case, we also have $K \sim H$.

– Similarly, if s' appears in c after the applicable plan is selected but before the intention is selected to execute, cycles through s' correspond to all possible selections of intentions to execute. We choose t' to be the configuration in d resulting from the transition (META-APPLY-1) of applying the meta rule (6.13) in R_3 which selects an applicable plan in d (hence $t' \neq last(d)$). Then, cycles through t' correspond to all possible selections of intentions. Hence, we have that $observe(s') = observe(t') = \top$, $label(c|s') = label(d|t') = \epsilon$. Then, applying the same argument as the previous case, we also have $K \sim H$.

– Similarly, if s' appears in c after the intention is selected but before it is executed, cycles through s' correspond to the execution of the selected intention. We choose t' as the configuration in d resulting from the transition (META-APPLY-1) of applying the meta rule (6.14) in R_3 which selects an intention to execute (hence $t' \neq last(d)$). Then, cycles through t' correspond to the execution of the selected intention. Hence, we have that $observe(s') = observe(t') = \top$, $label(c|s') = label(d|t') = \epsilon$. Then, applying the same argument as the previous case, we also have $K \sim H$.

– Similarly, if s' appears in c after the intention is executed, through s' correspond

to the clearing of empty intentions. We choose t' to be the configuration in d just after the transition which executes first step a of the selected intention (hence $t' \neq last(d)$ since there is at least a transition label (NEW-CYCLE) in d). Then, d is only one cycle through t'. Hence, we have that $observe(s') = observe(t') = \top$, $label(c|s') = label(d|t') = a$ if a is an external action or ϵ otherwise. Then, applying the same argument as the previous case, we also have $K \sim H$.

- For (2): let t' be a configuration a long d such that $t' \neq last(d)$. Applying a similar argument as for (1), we can also show that there exists $s' \in c$ such that $label(c|s') = label(d|t')$ and $\{c' \in DC(s_0) \mid first(c) = first(c'), s' \in c'\} \cong \{d' \in DC(t_0) \mid first(d) = first(d'), t' \in d'\}$..

Thus, by Theorem 5.2.6, we have that ag and $tr_{\text{Jason}}(ag)$ are weakly bisimilar.

□

6.4 Summary

In this chapter, we presented how to simulate Jason agents in **meta-APL**. We first reviewed the syntax and operational semantics of Jason. Then, we defined a translation which produces for each agent program in Jason a **meta-APL** program. The idea behind the translation is that each deliberation cycle of the Jason agent is simulated by a deliberation cycle of the **meta-APL** agent. Therefore, selections of relevant events, applicable plans and intentions to execute are simulated. Then, we showed that the Jason agent and the **meta-APL** agent operate equivalently according to the notion of weak bisimulation.

Chapter 7

Simulating 3APL

In this chapter, we show how to simulate 3APL agents in **meta-APL**. 3APL is a BDI-based agent programming language. However, it differs from Jason in important respects, including providing support for declarative goals and plan revision. We first review the syntax and the operational semantics of 3APL. Then, we define a translation function to transform a 3APL agent program into a **meta-APL** agent program. Finally, we prove that these two programs are equivalent under the notion of weak bisimulation.

7.1 3APL

In 3APL, agents are considered as individuals which have a state comprised of mental elements such as beliefs and goals. These mental elements provide a basis for making decisions about their behaviour. There are several versions of 3APL [Hindriks et al., 1999; Dastani et al., 2003b, 2005]. In the first version of 3APL presented in [Hindriks et al., 1999], goals are procedural where a goal is a sequence of basic actions such as external actions and belief tests, achievement goals are similar to function calls of imperative programming languages, and control statements include the conditional choice operator (if-then-else) and the iteration operator (while-do). Inspired by Dribble [van Riemsdijk et al., 2003], 3APL was extended in [Dastani et al., 2003b] to incorporate both declarative and proce-

dural goals. This extended version of 3APL also incorporates reflective features which allow programmers to specify how agents should revise their goals and plans. However, there are no deliberation cycles defined for this version of 3APL. In the latest implementation of an interpreter for 3APL, namely the 3APL platform [Dastani et al., 2005], some features of 3APL (such as goal revision) were removed and a deliberation cycle is defined, as depicted in Figure 7.1.

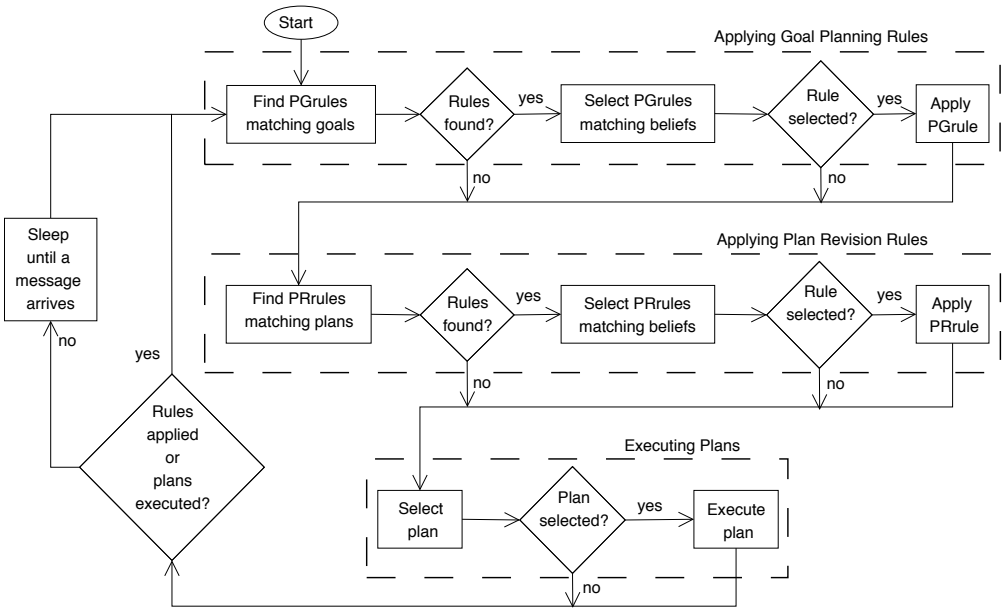

Figure 7.1: The implemented deliberation cycle in 3APL platform [Dastani et al., 2005].

In this chapter, we show how to simulate the version of 3APL which is implemented in the 3APL platform. To simplify the presentation, we omit communication actions, two plan constructs "if - then - else" and "while - do". Before introducing the simulation, we first review the syntax and the operational semantics of 3APL.

7.1.1 3APL Syntax

An agent program in 3APL consists of a set of capabilities, a initial belief base, a initial goal base, a set of goal planning rules (PG-rules) and a set of plan revision rules (PR-rules).

The syntax of an agent ag is as follows:

$$ag \quad ::= \quad (Cap, \sigma, \gamma, PG, PR)$$

where Cap denotes the set of capabilities, σ the initial belief base, γ the initial goal base, PG is the set of goal planning rules and PR is the set of plan revision rules.

Terms and atoms

Given a set *PRED* of predicates, a set *FUNC* of function and a set *VAR* of variables, a term t and an atom a in 3APL have the following syntax:

$$t \quad ::= \quad X \mid f([t(,t)^*])$$
$$a \quad ::= \quad p([t(,t)^*])$$

where $X \in$ *VAR*, $p \in$ *PRED* and $f \in$ *FUNC*. As usual, ground terms do not contain variables. In the following, we use a to denote an atom and b to denote a ground atom.

Beliefs

The belief base σ is a finite set of ground atoms and Horn clauses. The syntax of σ is defined as follows:

$$\sigma \quad ::= \quad bel^*$$
$$bel \quad ::= \quad b. \mid a \text{ :- } lit(,lit)^*.$$
$$lit \quad ::= \quad a \mid \text{not } a$$

Here, negation in a Horn clause is interpreted as negation as failure.

Goals

The goal base γ is a finite set of atoms or conjunctions of atoms. The syntax of γ is defined as follows:

$$\gamma \quad ::= \quad \kappa^*$$
$$\kappa \quad ::= \quad a \mid \kappa_1 \text{ and } \kappa_2$$

The difference between two individual goals "κ_1" and "κ_2" and the conjunctive goal "κ_1 and κ_2" is that "κ_1" and "κ_2" can be achieved individually at different times while "κ_1 and κ_2" is achieved if both "κ_1" and "κ_2" are achieved at the same time.

Capabilities

The set Cap of capabilities defines belief update actions bu of the agent ag for adding and deleting beliefs. Given a set *PRED* of predicates, a set *FUNC* of function and a set *VAR* of variables, the syntax of Cap is as follows:

$$
\begin{array}{lcl}
Cap & ::= & bu^* \\
bu & ::= & (\mathit{wff}, belup, lit^*) \\
\mathit{wff} & ::= & \top \mid lit \mid \mathit{wff} \text{ and } \mathit{wff} \mid \mathit{wff} \text{ or } \mathit{wff} \\
belup & ::= & a
\end{array}
$$

Each belief update action is defined in Cap by a belief query (or the precondition) *wff*, a name $belup$, and a list of literals (which are called postconditions). In this list, a positive literal of the form b indicates that b is added into the belief base, a negated literal of the form not a indicates that atoms matching a are deleted from the belief base.

Note that in the precondition *wff* of a belief update action, the "or" operator can be omitted without reducing the expressiveness of 3APL. However, we do need the "or" operator for the case of test actions (see below). In order for postconditions to be definable in terms of add and delete lists of literals, the following assumptions need to be made:

- Horn clauses are never changed by any action;
- literals in the belief base (or in the postcondition of any belup) are not defined by Horn clauses (they never occur as a head of a clause)

Plans

Plans are sequences of external actions, belief update actions, abstract plans and test actions. The syntax of plans are given below:

$$
\begin{array}{lll}
h & ::= & \epsilon \mid act \mid h_1; h_2 \\
act & ::= & ea \mid belup \mid absplan \mid \textit{wff}? \\
ea & ::= & e(t(,t)^*) \\
absplan & ::= & a
\end{array}
$$

where $absplan$ is an abstract plan. An abstract plan can be seen as a place holder for a plan during execution. Abstract plans cannot be executed directly and should be modified into plans by means of plan revision rules (see below).

Plan revision rules

The rule set PR contains plan revision rules which are used to modify intentions. The syntax of PR is as follows:

$$
\begin{array}{lll}
PR & ::= & r^* \\
r & ::= & h_1 \leftarrow \textit{wff} \ \mathbin{¦} \ h_2
\end{array}
$$

Given a PR rule $h_1 \leftarrow \textit{wff} \ ¦ \ h_2$, h_1 is called the head of the rule, *wff* is called the query of the rule, and h_2 is called the body of the rule.

Goal planning rules

The rule set PG contains goal planning rules which are used to create intentions for realising a goal derived from the goal base or reacting. The syntax of PG is as follows:

$$
\begin{array}{lll}
PG & ::= & p^* \\
p & ::= & \kappa \leftarrow \textit{wff} \ ¦ \ h \quad \mid \quad \leftarrow \textit{wff} \ ¦ \ h
\end{array}
$$

Given a PG rule $\kappa \leftarrow \textit{wff} \ ¦ \ h$, κ is called the goal of the rule, *wff* is called the query of the rule, and h is called the body of the rule. Similarly, given a PG rule $\leftarrow \textit{wff} \ ¦ \ h$, we say that its goal is empty, its query is *wff* and its body is h.

To simplify the presentation of the simulation, we omit two plan constructs "if *wff* then h_1 else h_2" and "while *wff* do h_1" which form part of the syntax of 3APL as defined in [Dastani et al., 2005]. However, the omission of these constructs does not reduce the expressiveness of 3APL. A plan "if *wff* then h_1 else h_2" can be transformed into an abstract plan $abs_{\text{if } wff \text{ then } h_1 \text{ else } h_2}$ together with the following two plan revision rules:

$$abs_{\text{if } wff \text{ then } h_1 \text{ else } h_2} \leftarrow wff \ \dagger \ h_1$$
$$abs_{\text{if } wff \text{ then } h_1 \text{ else } h_2} \leftarrow not(wff) \ \dagger \ h_2$$

where $not(wff)$ is defined inductively as follows:

- $not(a)$ = not a;
- $not(\text{not } a) = a$;
- $not(wff_1 \text{ and } wff_2) = not(wff_1)$ or $not(wff_2)$; and
- $not(wff_1 \text{ or } wff_2) = not(wff_1)$ and $not(wff_2)$.

A plan "while *wff* do h_1" can be similarly transformed into an abstract plan $abs_{\text{while } wff \text{ do } h_1}$ together with the following two plan revision rules:

$$abs_{\text{while } wff \text{ do } h_1} \leftarrow wff \ \dagger \ h_1; abs_{\text{while } wff \text{ do } h_1}$$
$$abs_{\text{while } wff \text{ do } h_1} \leftarrow not(wff) \ \dagger \ \epsilon$$

7.1.2 3APL Operational semantics

Given an agent defined in 3APL, it operates by generating new intentions from goal planning rules, modifying existing intentions by plan revision rules, and executing existing intentions. In this section, we review 3APL operational semantics presented in [Dastani et al., 2005] and extended with transition rules for steps "Execute plan" and "Apply PGrule" in 3APL deliberation cycle as depicted in Figure 7.1.

Phases in 3APL deliberation cycle

In order to present 3APL operational semantics as defined by the deliberation defined in 3APL platform, we define the following phases corresponding to the main steps in the 3APL deliberation cycle, indicated by the dashed boxes in Figure 7.1:

1. Message: this phase processes percepts from the environment and updates the belief base and the goal base (as some goals might be achieved after the belief base is updated).

2. Apply-PG: this phase applies a goal planning rule and produces a new intention for the intention base.

3. Apply-PR: this phase applies a plan revision rule which modifies the plan of an intention.

4. Execute-Int: this phase executes an intention.

Then, the relation between these phases is depicted in Figure 7.2. where arrows illustrate possible transitions from a phase to another as presented in Figure 7.1.

3APL configuration

The operational semantics of 3APL is defined by a set of transition rules that go from one configuration to another. In [Dastani et al., 2005], a configuration is defined to consist of an agent id, a belief base, a goal base, an intention base and the state of the agent's environment represented by a set of ground atoms. This definition is not enough to define the operational semantics with ordered steps in the deliberation cycle as depicted in Figure 7.1. We therefore extend each 3APL configuration with a phase indicator to specify which phase the configuration is in. As we are only interested in systems of a single agent in this thesis, the agent id are not needed and will be omitted.

A 3APL configuration is defined as follows:

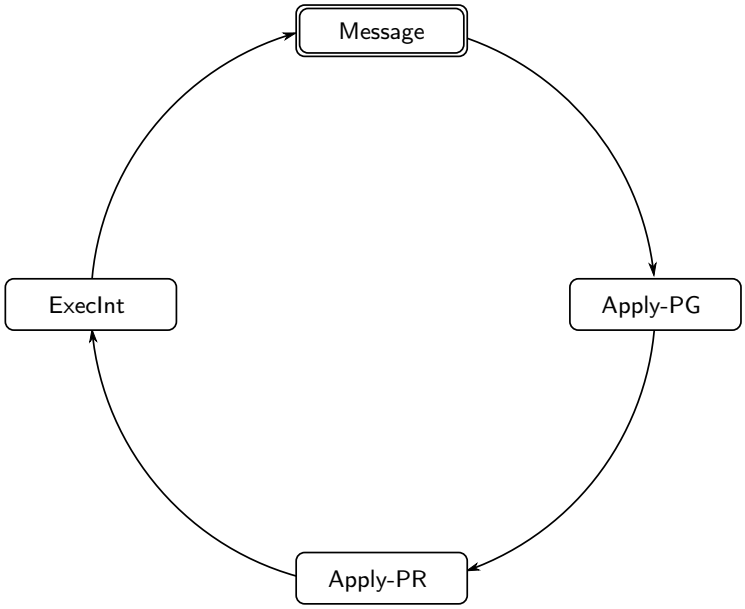

Figure 7.2: The deliberation cycle of 3APL platform.

Definition 7.1.1 (Extended 3APL configuration)**.** *An extended 3APL configuration is a tuple* $\langle \sigma, \gamma, I, ph, E \rangle$ *where:*

- σ *is a belief base which consists of ground atoms and Horn clauses;*
- γ *is a goal base which consists of atoms and conjunctions of atoms;*
- I *is a intention base which consists of intentions; each intention is a pair of* (κ, h) *where* κ *is a goal and* h *is a partially instantiated plan;*
- $ph \in \{$*Message*, *Apply-PG*, *Apply-PR*, *Execute-Int*$\}$ *is a phase indicator;*
- E *is a state of the agent's environment which is a set of ground atoms.*

In the following, we refer to extended 3APL configurations as 3APL configurations.

Let $C = \langle \sigma, \gamma, I, ph, E \rangle$ be a 3APL configuration. In 3APL, the query of a PG rule or a PR rule is evaluated with respect to the belief base σ as in Prolog. If the query succeeds, the result of the query is a substitution θ and we write $\sigma \vDash_B \mathit{wff} \mid \theta$. This evaluation is defined inductively as follows:

- $\sigma \models_B a \mid \theta$ iff:
  - there exists $b \in \sigma$ such that $b = a \mid \theta$ for some substitution θ; or
  - there exists a Horn clause $a' \text{ :- } l_1, \ldots, l_n$ such that $a' = a \mid \theta'$ and $\sigma \models_B (l_1\theta' \text{ and } \ldots \text{ and } l_n\theta') \mid \theta$;
- $\sigma \models_B \text{not } a \mid \varnothing$ iff $\sigma \not\models_B a \mid \theta'$ for any θ';
- $\sigma \models_B wff_1 \text{ and } wff_2 \mid \theta$ iff $\sigma \models_B wff_1 \mid \theta'$ and $\sigma \models_B wff_2\theta' \mid \theta$; and
- $\sigma \models_B (wff_1 \text{ or } wff_2) \mid \theta$ iff $\sigma \models_B wff_1 \mid \theta$ or $\sigma \models_B wff_2 \mid \theta$.

The goal κ of a PG rule is evaluated against the goal base γ of the configure C by looking for a goal κ' in γ where κ' entails κ. If the evaluation succeeds, the result is a substitution θ and we write $\gamma \models_G a \mid \theta$. The definition of this evaluation is given below:

- $\gamma \models_G a \mid \theta$ iff there exists $\kappa = (a_1 \text{ and } \ldots \text{ and } a_n) \in \gamma$ such that $\{a_1, \ldots, a_n\} \models_B a \mid \theta$;
- $\gamma \models_G (\kappa_1 \text{ and } \kappa_2) \mid \theta$ iff there exists $\kappa = (a_1 \text{ and } \ldots \text{ and } a_n) \in \gamma$ such that $\{a_1, \ldots, a_n\} \models_B \kappa_1 \mid \theta'$ and $\{a_1, \ldots, a_n\} \models_B \kappa_2\theta' \mid \theta$.

Given a 3APL agent $ag = (Cap, \sigma, \gamma, PG, PR)$ and the initial state of the environment E_0, the initial configuration of ag is $\langle \sigma, \gamma, \varnothing, \text{Message}, E_0 \rangle$. This means that the initial belief base is σ, the initial goal base is γ, the initial intention base is empty, there is no temporary information and the initial phase is Message.

Transition rules of 3APL

In the following, we review the transition rules for 3APL. The transition rules for applying PG rules and executing intentions are based from [Dastani et al., 2005]. As there is no transition rule describing the application of PR rules in [Dastani et al., 2005], we use the transition rule for applying PR rules from Dastani et al. [2003b].

The Message phase is the first phase in a deliberation cycle of a 3APL agent. Here, the agent updates its beliefs according to percepts from the environment. It is assumed that

this update is modelled by a function $update(E, \sigma)$ which returns an updated belief base from a given belief base σ and a state E of the environment. The transition rule to update the belief base is given below:

$$\frac{\begin{array}{rcl}\sigma' & = & update(E, \sigma) \\ \gamma' & = & \gamma \setminus \{\kappa \in \gamma \mid \sigma' \vDash_B \kappa \mid \theta\}\end{array}}{\langle \sigma, \gamma, I, \mathsf{Message}, E\rangle \rightarrow \langle \sigma', \gamma', I, \mathsf{Apply\text{-}PG}, E\rangle} \quad \text{(Update)}$$

In this transition rule, since the update of the belief base can achieve certain goals, achieved goals are also removed from the goal base.

In the Apply-PG phase, the agent applies a goal planning rule $p \in PG$.

Given a goal planning rule $p = \kappa \leftarrow \mathit{wff} \mid h$, we define $head(p) = \kappa$ which denotes the head of the rule p, $query(p) = \mathit{wff}$ which denotes the query of the rule p and $body(p) = h$ which denotes the body of the rule p. p is relevant to some goals in γ if $head(p)$ is derivable from γ. Similarly, given a reactive goal planning rule $p = \leftarrow \mathit{wff} \mid h$, we define $head(p) = \top$, $query(p) = \mathit{wff}$ and $body(p) = h$.

A PG rule can be applied iff it is relevant to a goal in the goal base and its query is evaluated to true against the belief base. In other words, a PG rule is applied in the Apply-PG phase iff $head(p)$ is derivable from the goal base γ and $query(p)$ is derivable from the belief base σ. This application generates a new intention from $body(p)$. The transition of applying p is given below:

$$\frac{\begin{array}{l}\exists p \in PG : \gamma \vDash_G head(p) \mid \theta \wedge \sigma \vDash_B query(p)\theta \mid \theta' \\ I' = I \cup \{(head(p)\theta, body(p)\theta\theta')\}\end{array}}{\langle \sigma, \gamma, I, \mathsf{Apply\text{-}PG}, E\rangle \rightarrow \langle \sigma, \gamma, I', \mathsf{Apply\text{-}PR}, E\rangle} \quad \text{(Apply-PG-1)}$$

If there are no such PG rules, no PG rule is applied in the Apply-PG phase. The transition in this case is as follows:

$$\frac{\neg(\exists p \in PG : \gamma \vDash_G head(p) \mid \theta \wedge \sigma \vDash_B query(p)\theta \mid \theta')}{\langle \sigma, \gamma, I, \mathsf{Apply\text{-}PG}, E\rangle \rightarrow \langle \sigma, \gamma, I, \mathsf{Apply\text{-}PR}, E\rangle} \quad \text{(Apply-PG-2)}$$

In the Apply-PR phase, the agent applies a plan revision rule $r \in PR$ to modify an intention.

Similar to goal planning rules, given a plan revision rule $r = h_1 \leftarrow \mathit{wff} \mid h_2$, we define $head(r) = h_1$ to denote the head of the rule r, $query(r) = \mathit{wff}$ to denote the query of the rule r and $body(r) = h_2$ to denote the body of the rule r. Then, a rule $r \in PR$ is relevant to an intention in I when $head(r)$ is unifiable with a prefix of the plan of the intention.

A PR rule r can be applied iff it is relevant to the plan of an intention in the intention base and its query is evaluated to be true against the belief base. In other words, $head(r)$ is unifiable with a prefix of the plan of an intention in the intention base I and $query(r)$ is derivable from the belief base σ. This application replaces the prefix with $body(r)$. The transition is given below:

$$\frac{\begin{array}{l}\exists r \in PR, (\kappa, h_1; h) \in I : head(r) = h_1 \mid \theta \wedge \sigma \models_B query(r)\theta \mid \theta' \\ I' = I \setminus \{(\kappa, h_1; h)\} \cup \{(\kappa, body(r)\theta\theta'; h)\}\end{array}}{\langle \sigma, \gamma, I, \mathsf{Apply\text{-}PR}, E\rangle \rightarrow \langle \sigma, \gamma, I', \mathsf{ExecInt}, E\rangle} \quad \text{(Apply-PR-1)}$$

If there are no such PR rules, no PR rule is applied in the Apply-PR phase. The transition for this case is as follows:

$$\frac{\neg(\exists r \in PR, (\kappa, h_1; h) \in I : head(r) = h_1 \mid \theta \wedge \sigma \models_B query(r)\theta \mid \theta')}{\langle \sigma, \gamma, I, \mathsf{Apply\text{-}PR}, E\rangle \rightarrow \langle \sigma, \gamma, I, \mathsf{ExecInt}, E\rangle} \quad \text{(Apply-PR-2)}$$

In the Execute-Int phase, the agent executes the first action of the an intention in the intention base. Here, only executable intentions are eligible for execution. This means they must start with either an external action, a belief update action, or a test action which succeeds with respect to the current belief base. In other words, intentions that start with an abstract plan or a test action which fails with respect to the current belief base are not executed. The execution then depends on the type of the action to be executed.

If this action is an external action, it is performed on the environment. [Dastani et al., 2005] assumes that the effect of ea is modelled by a function $G_{ea}(E)$ which returns a new state of the environment state after ea is executed. The transition of executing ea is given

below:

$$\frac{\begin{array}{lcl} \exists i & = & (\kappa, ea; h) \in I \\ \gamma & \models_G & \kappa \\ E' & = & G_{ea}(E) \\ I' & = & I \setminus \{i\} \cup \{(\kappa, h)\} \end{array}}{\langle \sigma, \gamma, I, \textsf{Execute-Int}, E\rangle \rightarrow \langle \sigma, \gamma, I', \textsf{Message}, E'\rangle} \quad \text{(Exec-EA)}$$

If the action is a belief update action, performing this action changes the belief base according to the declaration of the action. Let $(\mathit{wff}, belup, L)$ be a belief update action defined in Cap. We define $pre(belup) = \mathit{wff}$ to denote the precondition of $belup$, $del(belup) = \{\text{not } a \in L\}$ to denote beliefs to be removed by $belup$ and $add(belup) = \{a \in L\}$ to denote beliefs to be added by $belup$. The transition of applying $belup$ is given below:

$$\frac{\begin{array}{lcl} \exists i & = & (\kappa, belup; h) \in I \\ \gamma & \models_G & \kappa \\ I' & = & I \setminus \{i\} \cup \{(\kappa, h)\} \\ \sigma' & = & \sigma \setminus \{b \in \sigma \mid \sigma \models_B pre(belup) \mid \theta, \exists b' \in del(belup) : b'\theta = b\} \cup \\ & & \{b \mid \sigma \models_B pre(belup) \mid \theta, \exists b' \in add(belup) : b = b'\theta\} \\ \gamma' & = & \gamma \setminus \{\kappa \in \gamma \mid \sigma' \models_B \kappa \mid \theta\} \end{array}}{\langle \sigma, \gamma, I, \textsf{Execute-Int}, E\rangle \rightarrow \langle \sigma', \gamma', I', \textsf{Message}, E\rangle} \quad \text{(Exec-BU)}$$

If the action is a test action, performing this action tests the belief base. The transition is given below:

$$\frac{\begin{array}{lcl} \exists i & = & (\kappa, \mathit{wff}?; h) \in I \\ \gamma & \models_G & \kappa \\ \sigma & \models & \mathit{wff} \mid \theta \\ I' & = & I \setminus \{i\} \cup \{(\kappa, h)\theta\} \end{array}}{\langle \sigma, \gamma, I, \textsf{Execute-Int}, E\rangle \rightarrow \langle \sigma, \gamma, I', \textsf{Message}, E\rangle} \quad \text{(Exec-Test)}$$

If there are no executable intentions, no intention is executed in the ExecInt phase. The

transition rule for this case is as follows:

$$\frac{\begin{array}{c}\forall i \in I : (i = (\kappa, h) \wedge \gamma \not\models_G \kappa) \vee \\ i = (\kappa, absplan; h) \vee \\ (i = (\kappa, \mathit{wff}?; h) \wedge \sigma \not\models_B \mathit{wff})\end{array}}{\langle \sigma, \gamma, I, \mathsf{Execute\text{-}Int}, E\rangle \rightarrow \langle \sigma, \gamma, I, \mathsf{Message}, E\rangle} \quad \text{(No-Exec)}$$

where $absplan$ denotes an abstract plan.

7.1.3 Selections in a 3APL deliberation cycle

A deliberation cycle of a 3APL agent is a sequence of transitions $s_0 \xrightarrow{a_1} s_1 \ldots \xrightarrow{a_k} s_k$ where only s_0 and s_k are of the Message phase and transitions are derived from the transition rules of 3APL operational semantics. In this section, we analyse the characteristics of the 3APL deliberation cycle based on the selections which are made within the cycle.

The 3APL deliberation cycle contains at most three selections. The first selection is for a goal planning rule which is made in a configuration with phase Apply-PG. From this configuration, there are two possible transitions, one labelled with (Apply-PG-1) and the other labelled with (Apply-PG-2). The former transition corresponds to the application of a goal planning rule whose head is derivable from the goal base and its belief query is a logical consequence of the belief base. In the latter case, no goal planning rule is applied. Hence, we define:

Definition 7.1.2. *Let $c = s_0 \xrightarrow{a_1} s_1 \ldots \xrightarrow{a_k} s_k$ be a deliberation cycle of a 3APL agent ag. Then,*

- *c selects a goal planning rule (p, θ) if there is a transition a_i = (Apply-PG-1) for some $1 \leq i \leq k$ where p is applied by a_i and $s_i = (\sigma, \gamma, I, \mathsf{Apply\text{-}PG})$,*
- *c does not select a goal planning rule if there is a transition a_i = (Apply-PG-2) for some $1 \leq i \leq k$.*

The second selection is for a plan revision rule which is made in a configuration with phase Apply-PR. There are two possible transitions from this configuration, one labelled

with (Apply-PR-1) which means that a plan revision rule is applied and the other labelled with (Apply-PR-2) which means that no plan revision rule is applied in this deliberation cycle. Hence, we define:

Definition 7.1.3. *Let* $c = s_0 \xrightarrow{a_1} s_1 \ldots \xrightarrow{a_k} s_k$ *be a deliberation cycle of a 3APL agent* ag*. Then,*

- *c selects a plan revision rule* (p, θ) *if there is a transition* a_i *= (Apply-PR-1) for some* $1 \leq i \leq k$ *where* p *is applied by* a_i *and* $s_i = (\sigma, \gamma, I, \mathsf{Apply\text{-}PR})$*,*
- c *does not select a plan revision rule if there is a transition* a_i *= (Apply-PR-2) for some* $1 \leq i \leq k$*.*

The last selection is for an intention to execute which is made in a configuration with phase ExecInt. There are two possible transitions from this configuration. One is labelled with (Exec-EA), (Exec-BU) or (Exec-Test), which means that an intention is executed and the other transition is labelled with (No-Exec) which means that no intention is executed in this deliberation cycle. Hence, we define:

Definition 7.1.4. *Let* $c = s_0 \xrightarrow{a_1} s_1 \ldots \xrightarrow{a_k} s_k$ *be a deliberation cycle of a 3APL agent* ag*. Then,*

- c *selects an intention* int *to execute if there is a transition* a_i *= (Exec-EA), (Exec-BU) or (Exec-Test) for some* $1 \leq i \leq k$ *where* int *is executed by* a_i *and* $s_i = (\sigma, \gamma, I, \mathsf{ExecInt})$*,*
- c *does not select an intention to execute if there is a transition* a_i *= (No-Exec) for some* $1 \leq i \leq k$*.*

7.2 Translation

In this section, we construct a translation function to translate a 3APL agent program into a **meta-APL** agent program such that two agents are equivalent under the notion of weak bisimulation.

7.2.1 Outline of the translation

Before diving into the details of the translation, we first present an outline of the translation function. As in the translation of Jason into **meta-APL**, this translation is also defined by following the correspondences between phases in deliberation cycles of 3APL agents and **meta-APL** agents. Figure 7.3 illustrates these correspondences as dashed, two-ended arrows.

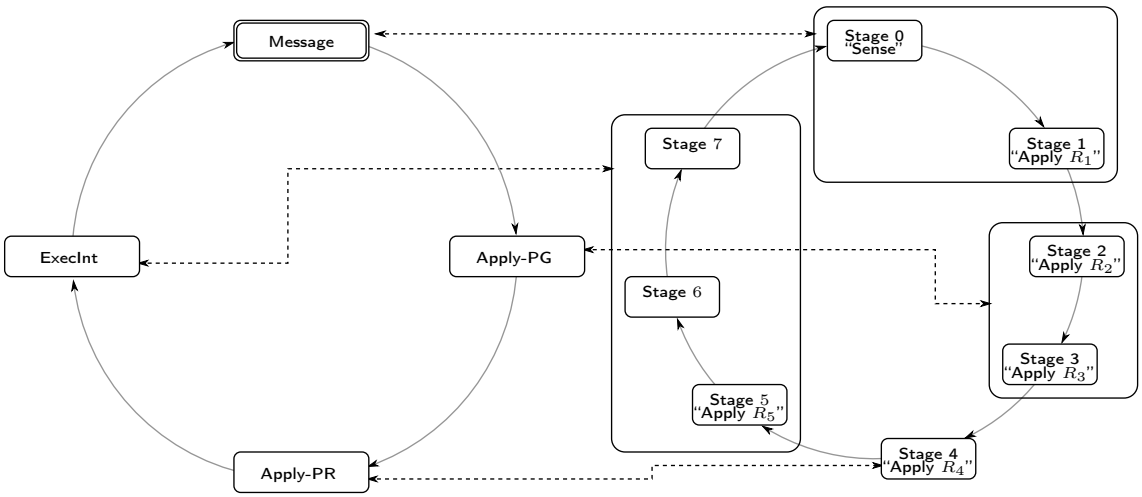

Figure 7.3: The correspondence between 3APL's and **meta-APL**'s deliberation cycles.

The first correspondence is between the Message phase of the 3APL deliberation cycle and the Sense phase of **meta-APL**'s deliberation cycles. In these phases, agents update their belief bases according to percepts received from the environment. Furthermore, 3APL agents also remove achieved goals and intentions for achieved goals due to changes in the belief base. Since this removal does not happen in the phase Sense of **meta-APL** agents, it is necessary to have meta rules which check for achieved goals and then delete them. These meta rules are collected in a rule set R_1 which is active after the phase Sense.

The next correspondence involves applying goal planning rules for generating new intentions. In the 3APL deliberation cycle, this is done in the Apply-PG phase where a goal planning rule with both goals and queries evaluating to be true is applied to generate a

new intention in the order of PR rules' occurrence. The application of the PG rule can be simulated in **meta-APL** by means of object-level rules which are the translations of goal planning rules. They are collected in a rule set R_2. Here, in order to simulate the selection and application of one goal planning rules, we define meta rules which set only one of the new plan instances generated by rules in R_2 to be an intention. These meta rules form the rule set R_3.

The third correspondence involves applying plan revision rules for repairing existing intentions. In the 3APL deliberation cycle, this is done in the Apply-PR phase where one of the plan revision rules with heads matching with a prefix and their queries are evaluated to be true is applied and its application in the Apply-PR phase replaces the matched prefix with the body part of the rule. We simulate the selection and application of PR rules in **meta-APL** by meta rules which are the translations of plan revision rules. The selection of one plan revision rules is encoded in the translations of plan revision rules so that only one of them can be applied in a cycle. These meta rules form the rule set R_4.

The last correspondence involves executing an intention. In the 3APL deliberation cycle, this is done in the ExecInt phase. We simulate this execution by a meta rule which picks an intention and sets its `scheduled` flag. This meta rule is defined in a rule set R_5.

Given the correspondences between a 3APL deliberation cycle and a **meta-APL** deliberation cycle, we now define a translation function tr_{3APL} as illustrated in Figure 7.4 which converts a 3APL agent program into a **meta-APL** one. This function is defined in terms of the following component functions:

- tr_{act} which translates belief update actions into meta actions;
- tr_{fact} and $tr_{clauses}$ which translates beliefs from the belief base into atoms and clauses;
- tr_{goal} which translates goals into atoms;
- tr_P which translates goal planning rules into object-level rules;
- tr_R which translates plan revision rules into meta rules.

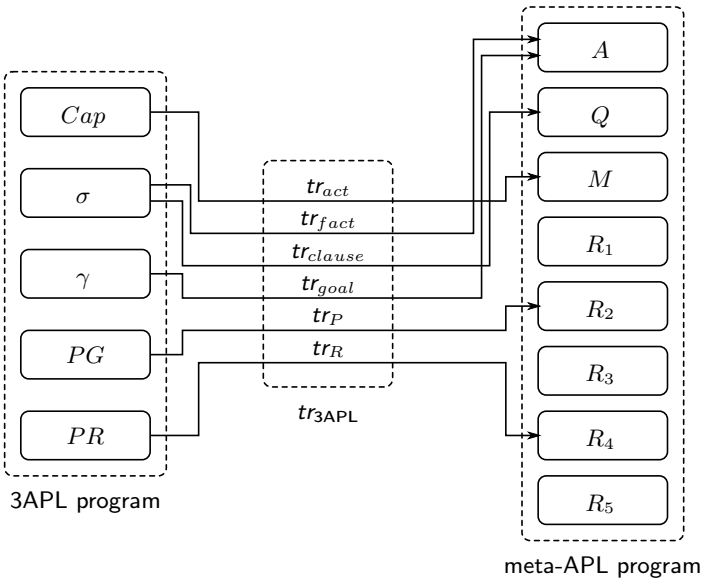

Figure 7.4: The translation function tr_{3APL}.

Given a 3APL agent program, the translation function tr_{3APL} returns a **meta-APL** agent program contains an initial atom base A, a set Q including, a set M of user-defined macros and five rule sets R_1, R_2, R_3, R_4 and R_5. The user-defined queries in Q, the the meta rule sets $R1$, R_3 and R_5 are static and common to all 3APL agent programs.

7.2.2 The static part of the translation

Defining additional common queries

The set Q_{static} contains additional queries which are used in meta rules and object-level rules in the translation. In particular, we have a query $belief(b)$ which checks if there is an instance of an atom $belief(b)$, a query $abstractPlan$ which checks if a plan starts with a subgoal action, and a query $failedTest$ which checks if a plan starts with a failed test action. These queries are defined as follows:

$$belief(B) \leftarrow \text{atom}(I, belief(B)) \tag{7.1}$$

$$belief(not(B)) \leftarrow \text{not } belief(B) \tag{7.2}$$

$$belief(and(B_1, B_2)) \leftarrow belief(B_1), belief(B_2) \tag{7.3}$$

$$belief(or(B_1, B_2)) \leftarrow belief(B_1) \tag{7.4}$$

$$belief(or(B_1, B_2)) \leftarrow belief(B_2) \tag{7.5}$$

$$goal(G) \leftarrow \text{atom}(I, goal(G')), subgoal(G, G') \tag{7.6}$$

$$subgoal(G, G). \tag{7.7}$$

$$subgoal(and(G_1, G_2), G) \leftarrow subgoal(G_1, G), subgoal(G_2, G) \tag{7.8}$$

$$subgoal(G, and(G_1, G_2)) \leftarrow subgoal(G, G_1) \tag{7.9}$$

$$subgoal(G, and(G_1, G_2)) \leftarrow subgoal(G, G_2) \tag{7.10}$$

$$abstractPlan(!\_; \_). \tag{7.11}$$

$$failedTest(?q; \_) \leftarrow \text{not } q \tag{7.12}$$

Defining meta rules for R_1

The set R_1 contains meta rules to remove goals which are believed, non-intended plan instances from the last cycle and completed intentions. These meta rules are defined below:

$$goal(G), belief(G) \rightarrow \texttt{delete-atom}(goal(G)) \tag{7.13}$$

$$\text{plan}(I, P), \text{not state}(I, \texttt{intended}) \rightarrow \texttt{delete-plan}(I) \tag{7.14}$$

$$\text{plan}(I, \epsilon), \text{state}(I, \texttt{intended}) \rightarrow \texttt{delete-plan}(I) \tag{7.15}$$

Here, the meta rule (7.13) simulates the effect of deleting achieving goals, i.e., the deletion of goals which are achieved during the transition (Update) of 3APL or the transition (Exec-BU) in the previous cycle. Then, the meta rule (7.14) removes plan instances generated in the last cycle but not promoted to be intentions as only one of these plan instances will be selected to become an intention as specified by (7.16) below. Hence, together with (7.16), (7.14) simulates the transition (Apply-PG-1) of 3APL. Note that these

non-intended plan instances won't effect the observation since we shall define not to observe non-intended plan instances. Finally, (7.15) takes care of completely executed intentions. Although 3APL does not delete completely executed intentions, this will not effect the establishment of a strong bisimulation later in this chapter since we will define not to observe empty intentions.

Defining meta rules for R_3

The set R_3 contains a meta rule which will set one of the plan instances – generated by object-level rules in R_2 in the same cycle – to become an intention. The meta rule is defined below:

$$cycle(N), \text{plan}(I, P), \text{not state}(I, \texttt{intended}), \text{not } atom(\_, selectedPG(\_, N)) \\ \rightarrow \texttt{set-state}(I, \texttt{intended}), \texttt{add-atom}(\_, selectedPG(I, N)) \quad (7.16)$$

In this meta rule, we use an atom $selectedPG$ to keep track of which plan instances are set to be intentions at each cycle. The condition of the meta rule (7.16) checks for a new plan instance and no plan instance has been set to be an intention in the current cycle yet. Then, the meta rule sets the plan instance to be an intention and records that an intention has been set in this cycle by adding an instance of the atom $selectedPG$.

As stated in the previous section, (7.16) (together with (7.14)) simulates the transition (Apply-PG-1) of 3APL.

Defining meta rules for R_5

The set R_5 contains a meta rule which is responsible for selecting an intention to execute. It simulates the Select-Int phase in deliberation cycles of 3APL agents where only intentions starting with external actions, belief update actions and test actions (that evaluate to *true* against the belief base) are eligible to be selected. The meta rule is as follows:

$$\text{not state}(\_, \texttt{scheduled}), \text{plan}(I, P), \text{state}(I, \texttt{intended}),$$
$$\text{not } abstractPlan(P), \text{not } failedTest(P)$$
$$\rightarrow \texttt{set-state}(I, \texttt{scheduled}) \tag{7.17}$$

In this meta rule, the condition not state(_, `scheduled`) is used to check if no intention has been selected for execution in this cycle. Then, the rule selects one of intentions in the intention base to be executed by setting the flag `scheduled` of the intention. Here, the two tests not $abstractPlan(P)$ and not $failedTest(P)$ make sure that only eligible intentions are selected. $abstractPlan$ and $failedTest$ are two user-defined queries. Together with the transitions (EXEC-EA), (EXEC-TEST-1) and (EXEC-META), this meta rule simulates the transitions (Exec-EA), (Exec-Test) and (Exec-BU) of 3APL, respectively.

7.2.3 Component translation functions

Translating belief update actions by tr_{act}

Each belief update action $bu = (\phi, belup, \{l_1, \ldots, l_n\})$ is translated into a macro ma where $tr_{act}(bu)$ gives:

$$belup = ?belief(tr_{query}(\phi)); tr_{bact}(l_1); \ldots; tr_{bact}(l_n)$$

where $tr_{query}(\phi)$ translate each query ϕ into a query in **meta-APL** as follows where a is an atom:

$$\begin{aligned}
tr_{query}(a) &= a \\
tr_{query}(not\ a) &= not(a) \\
tr_{query}(\phi_1 \text{ and } \phi_2) &= and(tr_{query}(\phi_1), tr_{query}(\phi_2)) \\
tr_{query}(\phi_1 \text{ or } \phi_2) &= or(tr_{query}(\phi_1), tr_{query}(\phi_2))
\end{aligned}$$

and $tr_{bact}(l)$ translates each literal into a meta action of adding or deleting an atom instance as follows where a is an atom:

$$\begin{aligned}
tr_{bact}(a) &= \texttt{add-atom}(belief(a)) \\
tr_{bact}(\text{not } a) &= \texttt{delete-atom}(belief(a))
\end{aligned}$$

In this translation, the query *wff* of bu forms the first action of ma which is evaluated against the belief base, the list of literals $\{l_1, \ldots, l_n\}$ of bu defines the rest of ma for adding and deleting corresponding atoms.

Furthermore, note that the syntax of *wff* is more general than that of goals κ, therefore, we shall see that tr_{query} is also used to translate goal expressions in PG rules, queries in PG and PR rules and test actions in plan bodies of PG and PR rules.

Translating beliefs by tr_{fact} and tr_{clause}

Given a belief base σ, we write σ_{fact} to denote the set of facts in σ. The translation function tr_{fact} wraps each fact in σ_{fact} into an atom in **meta-APL** as follows:

$$tr_{fact}(a) = belief(a)$$

Given a belief base σ, we write σ_{clause} to denote the set of Horn clauses in σ. The translation function tr_{clause} transforms each clause in σ_{clause} into a user-defined query as follows:

$$tr_{clause}(a \text{ :- } l_1, \ldots, l_n) = belief(a) \leftarrow belief(tr_{query}(l_1)), \ldots, belief(tr_{query}(l_1))$$

Translating goals by tr_{goal}

The translation tr_{goal} translates each goal in 3APL into an atom instance of the goal in **meta-APL**. This translation is defined as follows:

$$tr_{goal}(\kappa) = goal(tr_{query}(\kappa))$$

Translating goal planning rules by tr_P

Each goal planning rule is translated into an object-level rule by the translation function tr_P. Let $r_1 = g \leftarrow wff \mid h$ and $r_2 = \leftarrow wff \mid h$, the results of $tr_p(r_1)$ and $tr_P(r_2)$ are defined

as:

$$tr_p(r_1) = goal(tr_{query}(g)) : belief(tr_{query}(wff)) \rightarrow tr_{plan}(h)$$
$$tr_p(r_2) = goal(tr_{query}(top)) : belief(tr_{query}(wff)) \rightarrow tr_{plan}(h)$$

where top is a special atom which stands for empty goal, the translation function tr_{plan} is defined inductively on h as follows:

- $tr_{plan}(\epsilon) = \epsilon$,
- $tr_{plan}(ea) = ea$ where ea is an external action,
- $tr_{plan}(ap) = !ap$ where ap is an abstract plan,
- $tr_{plan}(wff?) = ?belief(tr_{query}(wff))$,
- $tr_{plan}(belup) = belup$ where $belup$ is a belief update action, and
- $tr_{plan}(h_1; h_2) = tr_{plan}(h_1); tr_{plan}(h_2)$.

In this translation, the head g of a goal planning rule p is directly transformed into the reason part of an object-level rule $tr_P(p)$; the test wff of p becomes the context query of $tr_P(p)$; and that body of p is translated into **meta-APL** where external actions are kept unchanged, abstract plans are translated into subgoal action, test actions are converted into test actions of **meta-APL** and belief update actions are converted into mental meta actions of the same names.

Translating plan revision rules by tr_R

Each plan revision rule $r = h_1 \leftarrow wff \ \vdots \ h_2$ is translated into a meta rule by tr_R where tr_R gives:

$$\begin{aligned} &\text{cycle}(N), \text{not atom}(\_, selectedPR(\_, N)), \text{plan}(I, tr_{plan}(h_1); X), \text{state}(I, \texttt{intended}), \\ &\quad belief(tr_{query}(wff)) \rightarrow \texttt{set-plan}(I, tr_{plan}(h_2); X); \texttt{add-atom}(selectedPR(I, N)) \end{aligned} \tag{7.18}$$

In this translation, we use an atom $selectedPR$ to keep track of which intentions are revised at a cycle. In the context query, cycle(N) and not atom(_, $selectedPR$(_, N)) check that no intention has been revised in the current deliberation cycle. In the rest of the context query, we then use plan($I, tr_{plan}(h_1); X$) to look for a plan instance with the body matching with the translation of h_1 and the translation of *wff* to match the condition when the rule can be applicable. When the translation of r is applied, the body of the intention is replaced with $tr_{plan}(h_2); X$ and an instance of $selectedPR(I, N)$ is added into the atom base.

Here, the application of a meta rule (7.18) simulates the transition (Apply-PR-1) of 3APL.

7.2.4 The translation function tr_{3APL}

Finally, we combine the above component translation functions to define tr_{3APL}. Given a 3APL program $ag = (Cap, \sigma, \gamma, PG, PR)$, $tr_{\text{3APL}}(ag) = (A, Q, M, R_1, R_2, R_3, R_4, R_5)$ where Q_{static}, R_1, R_3 and R_5 are defined in Section 7.2.2 and:

- $Q = Q_{static} \cup \{tr_{clause}(b) \mid b \in \sigma_{clause}\}$,
- $M = \{tr_{act}(belup) \mid belup \in Cap\}$,
- $A = \{tr_{fact}(b) \mid b \in \sigma_{fact}\} \cup \{top\}$,
- $R_2 = \{tr_P(p) \mid p \in PG\}$, and
- $R_4 = \{tr_R(r) \mid r \in PR\}$.

Figure 7.5 summaries the simulation of transitions in 3APL's operational semantics. Note that although the meta rule (7.15) does not simulate any 3APL's transition, it shall not effect the equivalence of the translation. The reason is that (7.15) is used to clean completed intentions. As these completed intentions are empty, they do not contribute to the behaviour of the agent any more, and hence, they are not considered in the equivalence between the configurations of a 3APL agent and its translation in **meta-APL**.

Transition	Simulated by
Update	SENSE and META-APPLY-1 which applies (7.13) to remove achieved goals
Apply-PG-1	OBJ-APPLY-1 which applies the translation of PG rules, META-APPLY-1 which applies (7.16) to select one, and META-APPLY-1 which applies (7.14) to clean non-selected ones in the next cycle
Apply-PG-2	OBJ-APPLY-2 when applying object rules in R_2
Apply-PR-1	META-APPLY-1 which applies the translation of a PR rule (7.18)
Apply-PR-2	META-APPLY-2 when applying meta rules in R_4
Exec-EA	META-APPL-1 which applies (7.17) to select one intention, and EXEC-EA
Exec-BU	META-APPL-1 which applies (7.17) to select one intention, EXEC-META of the corresponding macro defined by the function tr_{act}, and META-APPLY-1 which applies (7.13) to remove achieved goals
Exec-Test	META-APPL-1 which applies (7.17) to select one intention, and EXEC-TEST-1
No-Exec	NEW-CYCLE

Figure 7.5: The simulation of 3APL transitions in the translation.

7.3 Simulating selections

In this section, let us discuss the simulation of selections of a goal planning rule, a plan revision rule and an intention for execution in 3APL deliberation cycle in the translation into **meta-APL**. Recall that a **meta-APL** deliberation cycle is a sequence of transitions $s_0 \xrightarrow{a_1} s_1 \ldots \xrightarrow{a_k} s_k$ where only s_0 and s_k are of the Sense phase. In our translation of 3APL into **meta-APL**, these selections are simulated by transitions labelled with (META-APPLY-1) in a **meta-APL** deliberation cycle where suitable meta rules are applied. In particular, the selection of a goal planning rule is simulated by applying the meta rule (7.16) in R_3; the selection of a plan revision rule is simulated by applying the meta rule $tr_R(r)$ for some $r \in PR$; finally, the selection of an intention for execution is simulated by applying the

meta rule (7.17) in R_5 in R_5. Therefore, we have the following definition:

Definition 7.3.1. *Let $c = s_0 \xrightarrow{a_1} s_1 \ldots \xrightarrow{a_k} s_k$ be a deliberation cycle of the translation $tr_{3APL}(ag)$ of an 3APL agent ag. Then,*

- *c selects an object-level rule r if, in c, there is a transition a_i = (META-APPLY-1) , for some $1 \leq i \leq k$, of applying the meta rule* (7.16) *where $I = id(j)$, j is a plan instance in s_i and is generated by applying the object-level rule r.*
- *c does not select an object-level rule if, in c, there is no transition (META-APPLY-1) of applying the meta rule* (7.16).
- *c selects a meta rule r to repair some intention if, in c, there is a transition a_i = (META-APPLY-1), for some $1 \leq i \leq k$, of applying the meta rule* (7.18) *$r = tr_R(r')$ for some plan revision rule r'.*
- *c does not select a meta rule to repair some intention if, in c, there is no transition (META-APPLY-1) of applying a meta rule* (7.18).
- *c selects an intention j of a goal a to execute if, in c, there is a transition a_i = (META-APPLY-1), for some $1 \leq i \leq k$, of applying the meta rule* (7.17) *where $I = id(j)$, int is an intention of s_i and a is a justification of int in s_i (i.e., $id(a) \in justs(j)$).*
- *c does not select a plan revision rule if, in c, there is no transition (META-APPLY-1) of applying the meta rule* (7.17).

7.4 Equivalence by $tr_{\mathbf{3APL}}$

In this section, we show that our translation of a 3APL agent in Meta-APL is equivalent to the 3APL agent one under the notion of weak bisimulation. The proof is similar to the simulation of Jason in Theorem 6.3.1 where we show that there is a strong bisimulation between deliberation cycles of the 3APL agent and that of its translation in **meta-APL**.

Furthermore, we will show that this strong bisimulation satisfies the two conditions in Theorem 5.2.6 which entails a weak bisimulation between states of two agents.

7.4.1 Observations

Similar to the simulation of Jason in **meta-APL**, we first define observable properties of 3APL and **meta-APL** configurations in the translation tr_{3APL}.

Let $s = \langle \sigma, \gamma, I, ph, E \rangle$ be a 3APL configuration where σ is a belief base, γ is a goal base and Ps is an intention base, T is a temporal storage and ph is a phase indicator. We stipulate that s is observable when it is of the Message phase. Then, the observation of s includes beliefs in the belief base, goals in the goal base and intentions in the intention base.

$$\begin{aligned} observe(s) &= \top && \text{if } ph \neq \textsf{Message} \\ observe(s) &= (\sigma', \gamma', I') && \text{if } ph = \textsf{Message} \end{aligned}$$

where

- $\sigma' = \{tr_{fact}(b) \mid b \in \sigma_{fact}\}$;
- $\gamma' = \{goal(tr_{query}(\kappa)) \mid \kappa \in \gamma\}$;
- $I' = \{tr_{plan}(h) \mid \exists i : i = (\kappa, h) \in I, \gamma \vDash_G \kappa, h \neq \epsilon\}$.

Let $t = \langle A, \Pi, \rho, n \rangle$ be a **meta-APL** configuration, we also stipulate that t is observable if its counter is 0, i.e., t is of the Sense phase. Then, it is possible to observe beliefs in the atom base, goals in the atom base, and uncompleted intentions in the intention base. We define the observations of **meta-APL** configurations in the translation of 3APL as follows:

$$\begin{aligned} observe(t) &= \top && \text{if } \rho \neq 0 \\ observe(t) &= (Bs, Gs, Is) && \text{if } \rho = 0 \end{aligned}$$

where Bs, Es, Is are defined as follows:

- $Bs = \{belief(b) \mid \exists a \in A : \text{atom}(a) = belief(b)\}$;

- $Gs = \{goal(g) \mid \exists a \in A : \text{atom}(a) = goal(g), t \not\models belief(g)\}$;
- $Is = \{\pi\theta \mid \exists p \in \Pi, a \in A : \text{plan}(p) \neq \epsilon, \text{subs}(p) = \theta, \text{id}(a) \in \text{justs}(p), \text{atom}(a) = goal(g), t \not\models belief(g)\}$.

7.4.2 Equivalence theorem

Theorem 7.4.1. *Given a 3APL agent* $ag = (Cap, \sigma, \gamma, PG, PR)$*, let* $(A, Q, M, R_1, R_2, R_3, R_4, R_5) = tr_{3APL}(ag)$ *be its translation in* ***meta-APL****, we have that* ag *and* $tr_{3APL}(ag)$ *are weakly bisimilar.*

Proof.

Let s_0 be the initial configuration of $(Cap, \sigma, \gamma, PG, PR)$.

Let t_0 be the initial configuration of $(A, Q, M, R_1, R_2, R_3, R_4, R_5)$.

Similar to the proof of Theorem 6.3.1, we construct a strong bisimulation between cycles of ag in $DC(s_0)$ and cycles of $tr_{3\text{APL}}(ag)$ in $DC(t_0)$ which satisfies condition (1) and (2) of Theorem 5.2.6.

Let $c \in DC(s_0)$ be a deliberation cycle of ag and d be a deliberation cycle of $tr_{3\text{APL}}(ag)$. We say that:

- c and d select the same goal planning rule if c selects a goal planning rule r and d selects an object-level rule r' and $r' = tr_P(r)$.
- c and d select the same plan revision rule if c selects a plan revision rule r and d selects an meta rule r' to repair some intention and $r = tr_R(r')$.
- c and d select the same intention for execution if c selects an intention (g, h) and d selects an intention i for a goal a where $h = tr_{int(i)}$ and $goal(g) = \text{atom}(a)$.

Then, we say that c and d select bisimilar items if they select the same goal planning rule or no goal planning rule, the same plan revision rule or no plan revision rule, and the same intention for execution or no intention.

Given two configurations $s \in RC(s_0)$ and $t \in RC(t_0)$, we define the set of pairs of cycles which select bisimilar items and start from s and t, respectively, as follows:

$$eq(s,t) = \{(c,d) \in DC(s_0) \times DC(t_0) \mid first(c) = s, first(d) = t, \\ c \text{ and } d \text{ select bisimilar items}\}$$

Then, we define a relation $\sim$ as follows:

$\sim^0 = \varnothing$

$\sim^0_{\text{Start}} = \{(s_0, t_0)\}$

$\sim^{n+1} = \{(c,d) \in DC(s_0) \times C(t_0) \mid \exists\, s \sim^n_{\text{Start}} t : (c,d) \in eq(s,t)\}$

$\sim^{n+1}_{\text{Start}} = \{(s,t) \in RC(s_0) \times RC(t_0) \mid \exists\, c \sim^{n+1} d : last(c) = s \text{ and } last(d) = t\}$

$\sim = \bigcup_{n \geq 0} \sim^n$

In the following, we establish and prove a series of claims which will be used to prove that the relation $\sim$ constructed above satisfies the two conditions of Theorem 5.2.6.

Claim 7.4.2. *Let $s \in SC(s_0)$ be a configuration in the Message phase of ag and $t \in SC(t_0)$ be a configuration in the phase of Sense of $tr_{Jason}(ag)$, if $observe(s) = observe(t)$ and $(c,d) \in eq(s,t)$, then $observe(last(c)) = observe(last(d))$.*

Proof. As $(c,d) \in eq(s,t)$, c and d select bisimilar items, i.e., they select the same goal planning rule, the same plan revision rule, and the same intention for execution. We show that $observe(last(c)) = observe(last(d))$ are effectively equivalent by analysing the change of configurations along c and d.

If a goal planning rule $r \in PG$ is applied in c, a new plan (g,h) is generated in c as well. Since c and d apply the same goal planning rule, this mean $tr_P(r) \in R_2$ is applied in d and generates a new plan instance i such that plan$(i) = tr_{plan}(h)$, and there must be an atom instance a such that atom$(a) = goal(g)$ before $tr_P(r)$ is applied and id$(a) \in$ justs(i) after $tr(r)$ is applied.

If a plan revision rule $r' \in PR$ is applied in c (after r), it repairs some plan (g_1, h_1) to (g_1, h_1'). As c and d apply the same plan revision rule, $tr_R(r_1) \in R_4$ is also applied within d to repair an intention i_1 where plan(i_1) = $tr_{plan}(h_1)$ and its justification is an atom instance of g_1. Then, the result of applying $tr_R(r_1)$ is to repair the body of i_1 such that plan(i_1) = $tr_{plan}(h_1')$.

Finally, if an intention (g_2, h_2) is selected for execution in c, d selects an intention i_2 where plan(i_2) = $tr_{plan}(h_2)$ and its justification is an atom instance of g_2 as c and d select the same intention for execution. If h_2 starts with an external action, so is plan(i_2) and hence the effects to the environment of executing (g_2, h_2) and i_2 are the same. If h_2 starts with a belief update action bu, plan(i_2) starts with a mental meta action corresponding to bu, and hence the effects to the belief base of executing (g_2, h_2) and i_2 are the same. If h_2 starts with a test action, plan(i_2) starts with an equivalent test action, and hence the effects of executing (g_2, h_2) and i_2 to these intentions themselves are the same.

In total, changes between s and $last(c)$ are equivalently occurred between t and $last(d)$:

- For any new (or deleted) belief in $last(c)$, it must be caused by either perception or the effect of performing a belief update action. Equivalently, its corresponding atom instance is also new in $last(d)$ because we have the same new perception or the same belief update action performed resulting the same effect to the atom base.

- For any deleted goal in $last(c)$, it must be caused by perception or the effect of preforming a belief update action which makes the goal achieved. Then, the same perception or the equivalent mental meta action is received or performed in d, which generates the same change to the belief base and hence achieved the same goals.

- For any new intention in $last(c)$ which is created by a goal planning rule r, then the corresponding intention is also created by $tr_P(r)$ in $last(d)$.

- For any intention in $last(c)$ which is repaired by a plan revision rule r or executed, then the corresponding intention is also repaired by $tr_R(r)$ or executed in $last(d)$

Therefore, $observe(last(c)) = observe(last(d))$. □

Claim 7.4.3. *For all* $s \sim^{n}_{Start} t$, $observe(s) = observe(t)$.

Proof. We prove by induction on n. The base case, where $n = 0$, is trivial.

In the induction step, assume $s \sim^{n+1}_{\mathsf{Start}} t$. Then, there exists $c \sim^{n+1} d$ such that $last(c) = s$ and $last(d) = t$. Then, there exists $s' \sim^{n}_{\mathsf{Start}} t'$ such that $(c, d) \in eq(s', t')$. By induction hypothesis, we have that $s' \sim^{n}_{\mathsf{Sense}} t'$ implies $observe(s') = observe(t')$. Then, by Claim 7.4.2, we have that $observe(last(c)) = observe(last(d))$, i.e., $observe(s) = observe(t)$. □

Claim 7.4.4. *If* $(c, d) \in eq(s, t)$, *the* $label(c) = label(d)$.

Proof. As $(c, d) \in eq(s, t)$, they select bisimilar items.

- If c contains a transition corresponding to the execution of an external action a, as other transitions are silent, then, $label(c) = a$. Since c and d select bisimilar items, d also executes a. Thus, $label(d) = a$. Hence, $label(c) = label(d)$.
- If c does not contain any transition corresponding to the execution of an external action, all transitions in c are silent. Then, $label(c) = \epsilon$. Since c and d bisimilar items, d does not execute any external action. Thus $label(d) = \epsilon$. Hence, $label(c) = label(d)$.

□

Claim 7.4.5. *For any* $s \sim^{n}_{Start} t$ *and* $c \in DC(s_0)$ *such that* $first(c) = s$, *there exists* $d \in DC(t_0)$ *such that* $first(d) = t$, $(c, d) \in eq(s, t)$ *and* $observe(last(c)) = observe(last(d))$.

Proof. We shall construct d along with transitions in c.

The first transition in c is labelled with (Message) where the belief base of s is updated with respect to perception received from the environment. Together with this update, some goals are achieved which leads to the update of the goal base and the intention base accordingly. The result of this transition is a configuration s_1 where the phase indicator of s_1 is Apply-PG.

For any cycle d_1 starting from t, the first transition is labelled with (SENSE) where beliefs in the atom base, i.e., instances of atoms $belief(b)$, are also updated with respect to perception received from the environment. As $observe(s) = observe(t)$, after the update, the set of beliefs in the atom base is equivalent to the belief base of s_1. Also in d_1, achieved goals and intentions of achieved goals are removed from the atom base and the plan base by applying the meta rule (7.13) in R_1. Let t_1 be the first configuration in d_1 where the phase counter of t_1 is 2. Therefore, we have that beliefs and goals in the atom bases of t_1 are equivalent to beliefs and goals in the belief base and the goal base of s_1. We construct the first part of d as the sequence of transitions from t to t_1 in d_1.

If c selects a goal planning rule r, $head(r)$ is derived from the goal base of s_1 and $query(r)$ is a logical consequence of the belief base of s_1. Let s_2 be the configuration in c labelled with Apply-PR. Then, in s_2, there is a new intention generated by r. This also means $tr_P(r)$ is also applicable with respect to beliefs and goals in t_1. Therefore, there is a cycle d_2 from t_1 where $tr_P(r)$ is applied when rules from R_2 are applied and is selected by the meta rule (7.16) of R_3. Let t_2 be the first configuration in d_2 such that the phase counter of t_2 is 4, then only the plan instance generated by $tr_P(r)$ is set to be an intention. This intention is equivalent to the new intention in s_2. We continue constructing d from t_1 by the sequence of transitions from t_1 to t_2 in d_2.

If c does not select any goal planning rule, we have that for any rule r, either $head(r)$ is not derivable from the goal base of s_1 or $query(r)$ does not hold in the belief base of s_1. This also means $tr_P(r)$ is not applicable with respect to the atom base of t_1. Therefore, from t_1, there is only a transition labelled with (META-APPLY-2) to a configuration t_2. Here, we extend d from t by this transition to t_2.

If c selects a plan revision rule r, $head(r)$ is matched with a prefix of an intention in the intention base of s_2 and $query(r)$ is a logical consequence of the belief base of s_2. Let s_3 be the configuration in c labelled with ExecInt. Then, in s_3, an intention is repaired by r. This also means that $tr_R(r)$ is also applicable with respect to plan instances in the plan

base of t_2 and beliefs in the belief base of t_2. Therefore, there is a cycle d_3 from t_2 where $tr_R(r)$ is applied when rules from R_4 are applied. Let t_3 be the first configuration in d_3 such that the phase counter of t_3 is 5. Then, only one intention is repaired by $tr_R(r)$. This intention is equivalent to the repaired intention in s_3. We extend d from t_2 by the sequence of transitions in d_3 to t_3.

If c does not select any plan revision rule, we have that for any plan revision rule r, either $head(r)$ does not match with any prefix of an intention from the intention base of s_1 or $query(r)$ does not hold in the belief base of s_1. This also means $tr_P(r)$ is not applicable with respect to the atom base and the plan base of t_1. Therefore, from t_2, there is only a transition labelled with (META-APPLY-2) to a configuration t_3. Here, we extend d from t_2 by this transition to t_3.

If c selects an intention (g, h) for execution, then (g, h) is an intention in s_3 which does not start with a test action which will fail or an abstract plan. Then there is an intention i in the plan base of t_3 whose justification is an atom instance a in the atom base of t_3 such that $goal(g)$ = atom(a) and tr_{int}(plan(i)) = h. Then, i does not start with a test action which will fail or a subgoal action. Therefore, there is a cycle d_4 from t_3 which applies the meta rule (7.17) which selects i for execution. Then, we complete the construction of d by extending it from t_3 to the last configuration of d_4.

According to the construction of d, we have that c and d are select bisimilar items. Hence, $(c, d) \in eq(s, t)$. Then, by Claim 7.4.2, $observe(last(c)) = observe(last(d))$. □

Similar to the proof of Claim 7.4.6, we can also show the following result:

Claim 7.4.6. *For any $s \sim^n_{Start} t$ and $d \in DC(t_0)$ such that $first(d) = t$, there exists $c \in DC(s_0)$ such that $first(c) = s$, $(c, d) \in eq(s, t)$ and $observe(last(c)) = observe(last(d))$.*

Let us now return to the proof that $\sim$ is a strong bisimulation which satisfies conditions (1) and (2) of Theorem 5.2.6.

First, show that $\sim$ is a bisimulation. Let $c \sim d$, then there is $n > 0$ such that $c \sim^n d$. Thus,

there also exists $s \sim^{n-1}_{\text{Start}} t$ such that $(c, d) \in eq(s, t)$.

- We have

  $$observe(c) = observe(first(c)) = observe(s)$$
  $$observe(d) = observe(first(d)) = observe(t)$$

  By Claim 7.4.3, we have $observe(s) = observe(t)$, hence $observe(s) = observe(t)$.

- Assume that $c \xrightarrow{l} c'$ where $l = label(c)$, we have $last(c) = first(c')$. Obviously, $last(c) \sim^{n}_{\text{Start}} last(d)$ by the definition of $\sim^{n}_{\text{Start}}$. By Claim 7.4.5, there is a cycle d' from $last(d)$ such that $(c', d') \in eq(last(c), last(d))$, i.e., $c' \in \sim^{n+1} d'$. Then, we have that $c' \sim d'$. As $first(d') = last(d)$ and $label(c) = label(d) = l$ (by Claim 7.4.4), we also have that $d \xrightarrow{l} d'$.

- Similarly, we also have that if $d \xrightarrow{l} d'$, by Claim 7.4.6, there is a cycle c' such that $c \xrightarrow{l} c'$ and $c' \sim d'$.

We show that $\sim$ satisfies conditions (1) and (2) of Theorem 5.2.6:

- For (1): let s' be a configuration along c and $s \neq last(c)$. Note that c fixes the selections of a goal planning rule, a plan revision rule and an intention for execution.

  In the following, we denote $K = \{c' \in DC(s_0) \mid first(c) = first(c'), s' \in c'\}$ and $H = \{d' \in DC(t_0) \mid first(d) = first(d'), t' \in d'\}$.

  - If $s' = first(c)$, we select $t' = first(d)$.

    As $c \sim^{n} d$, $first(c) \sim^{n-1}_{\text{Start}} first(d)$, i.e., $s' \sim^{n-1}_{\text{Start}} t'$. By Claim 7.4.3, $observe(s') = observe(t')$.

    For any $c' \in K$, by Claim 7.4.5, there is a d' in $DC(t_0)$ such that $first(d') = t'$ and $(c', d') \in eq(s', t')$. Then, $c' \sim^{n} d'$, i.e., $c' \sim d'$. Thus, $d' \in H$.

    Similarly, we have for any $d' \in H$ there exists $c' \in K$ such that $c' \sim d'$.

    Thus, $K \sim H$.

– If $s' \neq first(c)$ and is before a goal planning rule is selected in c, cycles through s' correspond to all possible selections of goal planning rules, plan revision rules and intentions for execution. Then, we choose t' between $first(d)$ and before the transition (Apply-PG-1), (thus, $t' \neq last(d)$). Hence, cycles through t' correspond to all possible selections of goal planning rules, plan revision rules and intentions for execution.

As no external action is selected at s' and t', $label(c'|s') = label(d'|t') = \epsilon$.

Let $c' \in K$, then $first(c) = first(c')$. By Claim 7.4.5, there exists $d'' \in DC(t_0)$ such that $first(d'') = first(d)$ and $(c', d'') \in eq(first(c), first(d))$.

* If $t' \in d''$ (i.e., $t' \in$ both d and d'', then we select $d' = d''$, hence $d' \in H$ and $(c', d') \in eq(first(c), first(d))$. Thus, by definition of $\sim^n$ and $\sim$, we obtain $c' \sim^{n+1} d'$ and $c \sim d$.

* If $t' \notin d''$, we construct d' which has the same selections as d'' yet passing t'.

  Let t_1'' and t_1' be the configurations right before the transition (META-APPLY-1) for (7.16) in d'' and d, respectively. Then, atom instances representing beliefs and goals in t_1'' are the same as in t_1'. Thus, there is a deliberation cycle d_1' which passes t_1' (hence also t') which selects the same PG rule as d''.

  Similarly, let t_2'' and t_2' be the configurations right before the transition (META-APPLY-1) for (7.18) in d'' and d_1', respectively. Then, atom instances representing beliefs and goals and plan instances representing intentions in t_2'' are the same as in t_2'. Thus, there is a deliberation cycle d_2' which passes t_2' (hence also t') which selects the same PR rule as d''.

  Similarly, let t_3'' and t_3' be the configurations right before the transition (META-APPLY-1) for (7.17) in d'' and d_2', respectively. Then, plan instances representing intentions in t_3'' are the same as in t_3'. Thus, there is a de-

liberation cycle d'_3 which passes t'_3 (hence also t') which selects the same intention to execute as d''.

Then, we select $d' = d'_3$ and have that $(c', d') \in eq(first(c), (d))$ and $first(d) = first(d')$. As $t' \in d'$, $d' \in H$.

Similarly, we have for any $d' \in H$ there exists $c' \in K$ such that $c' \sim d'$. Hence, $K \sim H$.

– Similarly, if s' after a goal planning rule r_1 is selected but before a plan revision rule is selected, cycles through s' correspond to the selection of r_1, all possible selections of plan revision rules, intentions for execution. Then, we choose t' to be the configuration in d resulting from the transition (META-APPLY-1) of applying the meta rule (7.16) in R_3 (thus, $t' \neq last(d)$). This transition selects the plan instance generated by $tr_P(r)$ in d to be an intention. Hence, cycles through t' correspond to the selection of the goal planning rule r_1, all possible selections of plan revision rules and intentions for execution. Thus, we have that $label(c'|s') = label(d'|t') = \epsilon$.

Similar to the previous case, we can also prove that $K \sim H$.

– Similarly, if s' is the selection of a goal planning rule r_1, and the selection of a plan revision rule r_2, but before the selection of a intention for execution in c, cycles through s' correspond to the selections of r_1 and r_2, and all possible selections of an intention for execution. Then, we choose t' to be the configuration in d resulting from the transition (META-APPLY-1) of applying a meta rule in R_4 (thus, $t' \neq last(d)$). Hence, cycles through t' correspond to the selection of $tr_P(r_1)$ and $tr_R(r_2)$, and all possible selections of an intention for execution. Hence, we have that $label(c'|s') = label(d'|t') = \epsilon$.

Similar to the previous case, we can also prove that $K \sim H$.

– Similarly, if s' is after the selections of a goal planning rule r_1, a plan revision rule r_2 and an intention i for execution, cycles through s' correspond to the

execution of the selected intention. Then, we choose t' to be the configuration resulting from the transition (META-APPLY-1) of applying the meta rule (7.17) which selects the corresponding intention of i in d. Hence, cycles through t' correspond to the selections of $tr_P(r_1)$ and $tr_R(r_2)$, and the corresponding intention of i. Hence, we have that $label(c'|s') = label(d'|t') = \epsilon$.

Similar to the previous case, we have for any $d' \in H$ there exists $c' \in K$ such that $c' \sim d'$. Hence, $K \sim H$.

- For (2): let t' be a configuration a long d such that $t' \neq last(d)$. Applying a similar argument as for (1), we can also show that there exists $s' \in c$ such that $label(c|s') = label(d|t')$ and $\{c' \in DC(s_0) \mid first(c) = first(d), s' \in c'\} \sim \{d' \in DC(t_0) \mid first(d) = first(d'), t' \in d'\}$.

Thus, by Theorem 5.2.6, we have that ag and $tr_{\text{3APL}}(ag)$ are weakly bisimilar. □

7.5 Summary

In this chapter, we presented our simulation of 3APL agents in **meta-APL**. We showed that 3APL agents and their translations in **meta-APL** are equivalent under the notion of weak bisimulation. Here, we define a translation function so that each deliberation cycle of 3APL is simulated by a deliberation cycle of the translated agent in **meta-APL**. While the idea of simulating deliberation cycles of 3APL is similar to simulation of Jason, features of 3APL such as declarative goals and plan revision present new challenges. The success of simulating 3APL shows that **meta-APL** is a flexible language for simulating other agent programming languages with different sets of features.

Chapter 8

Conclusion and future work

This thesis has developed a BDI-based agent programming language **meta-APL** which supports procedural reflection. In this final chapter, we present an evaluation of the agent programming language **meta-APL**, then provide a summary of the contributions of the thesis, and finally draw directions and suggestions for future work.

8.1 Evaluation of meta-APL

The main purpose of the agent programming language **meta-APL** proposed in this thesis is to allow programmers to encode agent programs and deliberation strategies in the same language. In order to design such an agent programming language, we follow the approach proposed by des Rivières and Smith [1984] in including procedural reflection features in **meta-APL**. In particular, **meta-APL** includes predefined queries and meta actions which enable reasoning about the metal attitudes and intentions of the agents as well as modifying them and manipulating the execution state of intentions.

While it is the best to have **meta-APL** evaluated by agent programmers (i.e., the actual users of the language), we here provide an initial and preliminary evaluation of **meta-APL**:

- As mentioned at the beginning of the thesis (Section 1.2), once **meta-APL** is defined,

the main objectives are (i) to show that typical examples of deliberation strategies can be encoded in **meta-APL** and (ii) to highlight the flexibility of **meta-APL** by showing how to simulate agent programs in state-of-the-art BDI agent programming languages.

Regarding encoding typical deliberation strategies, Section 3.4 has provided the encoding of three typical deliberation strategies. The first deliberation strategy, namely parallel execution, executes intentions of an agent program in an interleaving fashion where each deliberation cycle selects a top-level executable intention to execute non-deterministically. This deliberation strategy is mainly used when defining formal operational semantics of most state-of-the-art agent programming languages such as AgentSpeak(L), Jason, 3APL, 2APL, etc. In contrast, the second deliberation strategy, called non-interleaved execution, executes intentions of an agent program in a non-interleaving fashion where each intention is executed until completion before another intention is selected to executed. This deliberation strategy is used in the interpreter of the agent programming language Jason to execute intentions generated by plans which are accompanied with the atomic flag. Finally, the third deliberation strategy, namely the round-robin strategy, executes intentions in a round-robin fashion so that every intention has the chance to be executed. This is the deliberation strategy that is implemented in the Jason interpreter.

Regarding the flexibility of **meta-APL**, we have simulated two state-of-the-art BDI agent programming languages, Jason and 3APL, in Chapters 6 and 7, respectively. For each of these agent programming languages, we define a translation function to translate each agent program in the source language into another agent program in **meta-APL**. These translation functions are defined in a modular manner, and we believe it will be easy to adapt for other BDI agent programming languages. In particular, initial mental attitudes of the source agents are translated into initial atoms of the target agents in **meta-APL**. Objects which define how intentions are generated,

such as plans in Jason and PG rules in 3APL, are translated into object level rules in **meta-APL**. To this end, the translation at the object level from Jason to **meta-APL** is rather straightforward. As 3APL has PR rules to revise intentions, they serve as a reflective feature which modifies the internal representation of intentions, therefore, they are translated into meta rules which use meta action `set-plan` to revise the plan body of intentions in **meta-APL**. Furthermore, these translations also include meta rules to implement the deliberation strategies of Jason and 3APL which are defined in their formal operational semantics. In particular, in terms of selecting intentions to execute, the operational semantics of both languages adopt the parallel execution strategy; therefore, we find the similarity between the second meta rule of the implementation of this strategy in Section 3.4 and the two meta rules (6.14) and (7.17) in the translations of Jason and 3APL, respectively. The translations also include other meta rules to implement the selection of which intention to adopt in a deliberation cycle. Selection strategies such as those implicit in the formal operational semantics of Jason and 3APL, where a single intention is adopted randomly, can be straightforwardly implemented as a meta rule of **meta-APL** where the rule queries for a plan instance which is generated in the current cycle and sets it to be an intention. Here, we also see a similarity between the meta rules (6.13) and (7.16) which implements the above idea.

- From the use of **meta-APL** in Section 3.4 as well as Chapters 6 and 7, we would argue that the encoding typical deliberation strategies to select intention to execute (such as parallel execution and non-interleaved execution) and to selection intention to adopt (such as randomly adopting an intention) are fairy easy and straightforward. Furthermore, we also show that common features such as the atomic flag in Jason and the round-robin strategy in the Jason interpreter can also be implemented in **meta-APL**. Therefore, we hope that **meta-APL** will be flexible enough to encode other deliberation strategies in a straightforward way.

- At the object level, the straightforward translation (from beliefs, goals and events in Jason and 3APL to atoms in **meta-APL** and from plans in Jason and PG rules in 3APL into object level rules in **meta-APL**) shows that **meta-APL** is as simple to use as other BDI agent programming languages such as Jason and 3APL.

Obviously, the above evaluation are still preliminary and subjective. Therefore, a meaningful and valuable direction for the future work is to have an extensive and in-depth evaluation of **meta-APL**. In particular, we will look at other deliberation strategies in existing agent programming languages and find out how to implement them in **meta-APL**.

8.2 Summary of Contributions

In this thesis, we have argued that there is a need for a BDI-based agent programming language which enables deliberation strategies to be programmed in the language itself. A deliberation strategy specifies how an agent selects plans for adoption and execution. We gave a brief review of existing agent programming languages and analysed the current support within these languages for programming different deliberation strategies. Here, we argued that existing languages provide a limited support. For example, KAs in PRS [Georgeff and Lansky, 1987], goal planning rules in 3APL [Hindriks et al., 1999; Dastani et al., 2005] and plans in Jason [Bordini et al., 2007] only allow the agent programmer to specify conditions when plans are applicable. With the exception of PRS, the selection when there are more than one applicable plans is carried out by selection functions which are implementable in the host languages of the interpreters of these agent programming languages. The support is even more limited for programming different strategies of selecting plans for execution where only Jason leaves a limited space for customising the default strategy. We also argued that a promising approach to this open problem is to develop an agent programming language which supports procedural reflection [des Riv-

ières and Smith, 1984]. In Chapter 2, we gave a brief overview of procedural reflection and analysed the current appearance of procedural reflection in several agent programming languages.

Then, in Chapters 3 and 4 we presented the syntax and the operational semantics of **meta-APL**, a BDI-based agent programming language which supports procedural reflection. **Meta-APL** is a very simple rule-based language. However it contains all the components necessary to implement a wide variety of BDI-based agent programming language features, deliberation cycles and commitment strategies, including beliefs, procedural and deliberative goals, events, plan selection and intention scheduling, blind commitment and various forms of open minded commitment. We conjecture that the built-in features of meta-APL at least approximate a "core specification" of what constitutes a BDI agent programming language. From this perspective, the key distinguishing feature of BDI languages (compared to other rule based languages) are plans and intentions (persistent plan instances). Everything else is definable in terms of these primitives.

We demonstrated the flexibility of **meta-APL** by showing how to simulate Jason and 3APL (referred to as "source languages" in what follows) in **meta-APL**. The equivalence of these simulations is based on the notion of strong and weak bisimulations [Milner, 1989] which have been used for multi-agent systems in [Hindriks, 2001]. In Chapter 5, we defined a notion of strong bisimulation between deliberation cycles of two agent programs (also referred to as cycle-based bisimulation). We showed that if such a cycle-based bisimulation exists, it gives rise to a weak bisimulation between configurations of the agent in the source languages and the agent in **meta-APL**; hence the two agents have equivalent behaviours.

Based on the result of cycle-based bisimulation, we presented the simulations of Jason and 3APL in **meta-APL** in Chapters 6 and 7, respectively. In each chapter, we gave a detailed review of the corresponding source agent programming language including its syntax, its semantics and the deliberation cycle that its interpreter implements. We

then analysed the key steps and the order that these steps appear in the deliberation cycle of the source language. Based on this analysis, we defined a translation function which translates agent programs in the source language into an agent program in **meta-APL**. In general, such a translation will convert elements in the agent program in the source language into parts of the agent program in **meta-APL**. The other parts of the agent program in **meta-APL** are static and common to all agent programs in the source language. They are used to implement the deliberation strategy which is implemented in the deliberation cycle of the source language. This is natural as all agent programs in the source language share the same deliberation strategy. Then, at the end of both chapters, we presented the proof to show that the agent in the source language and its translation in **meta-APL** have equivalent behaviours. The proof is carried out by constructing a bisimulation between deliberation cycle of the agent program in the source language and its translation and show that this bisimulation is a cycle-based bisimulation.

8.3 Future work

The results of this thesis have raised a number of potential directions of research for future work. In this section, we highlight two of them, namely: implementation of an interpreter for **meta-APL**, and application of **meta-APL** in verification of heterogeneous multi-agent systems.

Currently, no interpreter has been implemented for **meta-APL**. Obviously, it would be useful and interesting to implement an interpreter for **meta-APL** where agent programs in **meta-APL** can be executed. This will be a practical tool for the agent designer and the agent programmer to design and to encode agent programs of an application domain as well as to examine different deliberation strategies in order to look for a "best" one in the application domains.

Another promising direction of research is to use **meta-APL** for verifying heterogeneous systems of multiple agents. The idea here is to translate agent programs in other

languages into agent programs in **meta-APL** with equivalent behaviours, and to define a translation from meta-APL programs into the specification language of a model-checker. The results of this thesis showed that it is possible to translate Jason and 3APL programs into meta-APL. Preliminary experiments have been carried out where agent programs of Jason and 3APL are translated into **meta-APL** using the translations defined in this thesis and then these agent programs in **meta-APL** are encoded in Maude [Clavel et al., 1996]. Their properties are checked in the Maude LTL model checker. However, this direction also requires further extensions of **meta-APL** such as adding features for communication between agents and extending the simulation results in **meta-APL** to other agent programming languages.

Finally, an important direction of future work is to carry out an in-depth and objective evaluation of **meta-APL**. One way to do this is to use **meta-APL** to simulate other agent programming languages as well as their deliberations strategies. By investigating a wide range of agent programming languages and deliberation strategies, it would be possible to show the limits of **meta-APL**, and precisely specify the border between deliberation strategies that can be defined and those that cannot be defined in **meta-APL**. Through these activities, the comparison of the implementation of different deliberation strategies in **meta-APL** and other programming languages (such as the ones that implement interpreters of other agent programming languages) will provide a qualitative evaluation about the design objectives of **meta-APL** such as its simplicity and ease of use.

Bibliography

José Júlio Alferes, Antonio Brogi, João Alexandre Leite, and Luís Moniz Pereira. Evolving Logic Programs. In *Logics in Artificial Intelligence (JELIA)*, pages 50–61, 2002.

Howard Barringer, Michael Fisher, Dov M. Gabbay, Graham Gough, and Richard Owens. METATEM: A Framework for Programming in Temporal Logic. In J. W. de Bakker, Willem P. de Roever, and Grzegorz Rozenberg, editors, *REX Workshop*, volume 430 of *Lecture Notes in Computer Science*, pages 94–129. Springer, 1989. ISBN 3-540-52559-9.

Howard Barringer, Michael Fisher, Dov M. Gabbay, Graham Gough, and Richard Owens. METATEM: An Introduction. *Formal Asp. Comput.*, 7(5):533–549, 1995.

Fabio Bellifemine, Federico Bergenti, Giovanni Caire, and Agostino Poggi. JADE - A Java Agent Development Framework. In Bordini et al. [2005], pages 125–147. ISBN 0-387-24568-5.

Rafael H. Bordini and Jomi Fred Hübner. BDI Agent Programming in AgentSpeak Using *Jason* (Tutorial Paper). In Francesca Toni and Paolo Torroni, editors, *CLIMA*, volume 3900 of *Lecture Notes in Computer Science*, pages 143–164. Springer, 2005. ISBN 3-540-33996-5.

Rafael H. Bordini, Ana L. C. Bazzan, Rafael de Oliveira Jannone, Daniel M. Basso, Rosa Maria Vicari, and Victor R. Lesser. AgentSpeak(XL): efficient intention selection in

BDI agents via decision-theoretic task scheduling. In *AAMAS*, pages 1294–1302. ACM, 2002.

Rafael H. Bordini, Mehdi Dastani, Jürgen Dix, and Amal El Fallah-Seghrouchni. *Multi-Agent Programming: Languages, Platforms and Applications*, volume 15 of *Multiagent Systems, Artificial Societies, and Simulated Organizations*. Springer, 2005. ISBN 0-387-24568-5.

Rafael H. Bordini, Lars Braubach, Mehdi Dastani, Amal El Fallah-Seghrouchni, Jorge J. Gómez-Sanz, João Leite, Gregory M. P. O'Hare, Alexander Pokahr, and Alessandro Ricci. A Survey of Programming Languages and Platforms for Multi-Agent Systems. *Informatica (Slovenia)*, 30(1):33–44, 2006.

R.H. Bordini, J.F. Hübner, and M. Wooldridge. *Programming multi-agent systems in AgentSpeak using Jason*. Wiley-Interscience, 2007.

Michael E. Bratman. *Intention, Plans, and Practical Reason*. CSLI Publications, 1999.

Manuel Clavel, Steven Eker, Patrick Lincoln, and José Meseguer. Principles of Maude. *Electr. Notes Theor. Comput. Sci.*, 4:65–89, 1996.

Philip R. Cohen and Hector J. Levesque. Intention is Choice with Commitment. *Artif. Intell.*, 42(2-3):213–261, 1990.

Mehdi Dastani. 2APL: a practical agent programming language. *Autonomous Agents and Multi-Agent Systems*, 16(3):214–248, 2008.

Mehdi Dastani, Frank S. de Boer, Frank Dignum, and John-Jules Ch. Meyer. Programming agent deliberation: an approach illustrated using the 3APL language. In *AAMAS*, pages 97–104. ACM, 2003a. ISBN 1-58113-683-8.

Mehdi Dastani, Birna van Riemsdijk, Frank Dignum, and John-Jules Ch. Meyer. A Programming Language for Cognitive Agents Goal Directed 3APL. In Mehdi Dastani, Jür-

gen Dix, and Amal El Fallah-Seghrouchni, editors, *PROMAS*, volume 3067 of *Lecture Notes in Computer Science*, pages 111–130. Springer, 2003b. ISBN 3-540-22180-8.

Mehdi Dastani, M. Birna van Riemsdijk, and John-Jules Ch. Meyer. Programming Multi-Agent Systems in 3APL. In Bordini et al. [2005], pages 39–67. ISBN 0-387-24568-5.

Frank S. de Boer, Koen V. Hindriks, Wiebe van der Hoek, and John-Jules Ch. Meyer. A verification framework for agent programming with declarative goals. *J. Applied Logic*, 5(2):277–302, 2007.

Louise A Dennis and Berndt Farwer. Gwendolen: A BDI language for verifiable agents. In *AISB 2008 Convention Communication, Interaction and Social Intelligence*, 2008.

Louise A. Dennis, Berndt Farwer, Rafael H. Bordini, and Michael Fisher. A flexible framework for verifying agent programs. In Lin Padgham, David C. Parkes, Jörg P. Müller, and Simon Parsons, editors, *Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems*, pages 1303–1306. IFAAMAS, 2008. ISBN 978-0-9817381-2-3.

Louise A. Dennis, Michael Fisher, Matthew P. Webster, and Rafael H. Bordini. Model checking agent programming languages. *Automated Software Engineering*, 19(1):5–63, 2012.

Jim des Rivières and Brian Cantwell Smith. The Implementation of Procedurally Reflective Languages. In *LISP and Functional Programming*, pages 331–347, 1984.

Michael Fisher. MetateM: The Story so Far. In Rafael H. Bordini, Mehdi Dastani, Jürgen Dix, and Amal El Fallah-Seghrouchni, editors, *PROMAS*, volume 3862 of *Lecture Notes in Computer Science*, pages 3–22. Springer, 2005. ISBN 3-540-32616-2.

Dov M. Gabbay. The Declarative Past and Imperative Future: Executable Temporal Logic for Interactive Systems. In Behnam Banieqbal, Howard Barringer, and Amir Pnueli,

editors, *Temporal Logic in Specification*, volume 398 of *Lecture Notes in Computer Science*, pages 409–448. Springer, 1987. ISBN 3-540-51803-7.

Michael P. Georgeff and Amy L. Lansky. Reactive Reasoning and Planning. In Kenneth D. Forbus and Howard E. Shrobe, editors, *AAAI*, pages 677–682. Morgan Kaufmann, 1987.

Patrick J. Hayes. In Defense of Logic. In R. Reddy, editor, *IJCAI*, pages 559–565. William Kaufmann, 1977.

Koen V. Hindriks. *Agent programming languages: programming with mental models*. PhD thesis, Universiteit Utrecht, 2001.

Koen V. Hindriks, Frank S. de Boer, Wiebe van der Hoek, and John-Jules Ch. Meyer. Control Structures of Rule-Based Agent Languages. In Jörg P. Müller, Munindar P. Singh, and Anand S. Rao, editors, *ATAL*, volume 1555 of *Lecture Notes in Computer Science*, pages 381–396. Springer, 1998. ISBN 3-540-65713-4.

Koen V. Hindriks, Frank S. de Boer, Wiebe van der Hoek, and John-Jules Ch. Meyer. Agent Programming in 3APL. *Autonomous Agents and Multi-Agent Systems*, 2(4):357–401, 1999.

Koen V. Hindriks, Frank S. de Boer, Wiebe van der Hoek, and John-Jules Ch. Meyer. Agent Programming with Declarative Goals. In Cristiano Castelfranchi and Yves Lespérance, editors, *Intelligent Agents VII Agent Theories Architectures and Languages*, volume 1986 of *Lecture Notes in Computer Science*, pages 228–243. Springer, 2000. ISBN 3-540-42422-9.

João Alexandre Leite, José Júlio Alferes, and Luís Moniz Pereira. MINERVA - A Dynamic Logic Programming Agent Architecture. In John-Jules Ch. Meyer and Milind Tambe, editors, *ATAL*, volume 2333 of *Lecture Notes in Computer Science*, pages 141–157. Springer, 2001. ISBN 3-540-43858-0.

Hector J. Levesque, Raymond Reiter, Yves Lespérance, Fangzhen Lin, and Richard B. Scherl. GOLOG: A Logic Programming Language for Dynamic Domains. *The Journal of Logic Programming*, 31(1-3):59–83, 1997.

Viviana Mascardi, Maurizio Martelli, and Leon Sterling. Logic-based specification languages for intelligent software agents. *CoRR*, cs.AI/0311024, 2003.

Robin Milner. *Communication and concurrency*. PHI Series in computer science. Prentice Hall, 1989. ISBN 978-0-13-115007-2.

David N. Morley and Karen L. Myers. The SPARK Agent Framework. In *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 714–721. IEEE Computer Society Washington, DC, USA, IEEE Computer Society, 2004. ISBN 1-58113-864-4.

Gordon D Plotkin. A structural approach to operational semantics. Technical report, Computer Science Department, Aarhus University Denmark, 1981.

Amir Pnueli. The Temporal Logic of Programs. In *FOCS*, pages 46–57. IEEE Computer Society, 1977.

Alexander Pokahr, Lars Braubach, and Winfried Lamersdorf. Jadex: A BDI Reasoning Engine. In Bordini et al. [2005], pages 149–174. ISBN 0-387-24568-5.

Anand S. Rao. AgentSpeak(L): BDI Agents Speak Out in a Logical Computable Language. In Walter Van de Velde and John W. Perram, editors, *MAAMAW*, volume 1038 of *Lecture Notes in Computer Science*, pages 42–55. Springer, 1996. ISBN 3-540-60852-4.

Anand S. Rao and Michael P. Georgeff. Modeling Rational Agents within a BDI-Architecture. In James F. Allen, Richard Fikes, and Erik Sandewall, editors, *KR*, pages 473–484. Morgan Kaufmann, 1991. ISBN 1-55860-165-1.

Brian Cantwell Smith. Reflection and Semantics in Lisp. In Ken Kennedy, Mary S. Van Deusen, and Larry Landweber, editors, *POPL*, pages 23–35. ACM Press, 1984. ISBN 0-89791-125-3.

John Thangarajah, Lin Padgham, and Michael Winikoff. Detecting & avoiding interference between goals in intelligent agents. In Georg Gottlob and Toby Walsh, editors, *International Joint Conference on Artificial Intelligence*, pages 721–726. Morgan Kaufmann Publishers, Morgan Kaufmann, 2003.

Michael Thielscher. FLUX: A logic programming method for reasoning agents. *Theory and Practice of Logic Programming*, 5(4-5):533–565, 2005.

Birna van Riemsdijk, Wiebe van der Hoek, and John-Jules Ch. Meyer. Agent programming in Dribble: from beliefs to goals using plans. In *Proceedings of the second international joint conference on Autonomous agents and multiagent systems*, pages 393–400. ACM New York, NY, USA, 2003.

Willem Visser, Klaus Havelund, Guillaume P. Brat, Seungjoon Park, and Flavio Lerda. Model Checking Programs. *Autom. Softw. Eng.*, 10(2):203–232, 2003.

Michael Winikoff. JACK^{TM} Intelligent Agents: An Industrial Strength Platform. In Bordini et al. [2005], pages 175–193. ISBN 0-387-24568-5.

Michael Winikoff. Implementing commitment-based interactions. In Edmund H. Durfee, Makoto Yokoo, Michael N. Huhns, and Onn Shehory, editors, *Proceedings of the 6th international joint conference on Autonomous agents and multiagent systems*, page 128. ACM, IFAAMAS, 2007. ISBN 978-81-904262-7-5.

M. Wooldridge. *An introduction to multiagent systems*. Wiley, 2002.

M. Wooldridge and N.R. Jennings. Intelligent agents: Theory and practice. *Knowledge engineering review*, 10(2):115–152, 1995.

Appendix A

Reference of meta-APL

In the following, *ID* is a non-empty totally ordered set of ids; *PRED* is a non-empty set of predicate symbols; *FUNC* is a non-empty set of function symbols; and *VAR* is a non-empty set of variables.

Terms and atoms

$$t \quad ::= \quad i \mid X \mid t_1; \ldots; t_m \mid f(t_1, \ldots, t_m) \mid p(t_1, \ldots, t_m)$$

$$a \quad ::= \quad p(t_1, \ldots, t_m)$$

where $i \in ID$, $X \in VAR$, $f \in FUNC$ and $p \in PRED$.

Plans

$$\pi ::= \epsilon \mid (ea \mid ?q \mid ma \mid !a); \pi$$

Flags

Flag	Meaning
`intended`	The plan instance is an intention
`scheduled`	The plan instance will be executed in the current cycle
`stepped`	The plan instance has been executed in the last cycle
`failed`	The last execution of the plan instance has failed

Queries

Query	Meaning
atom(i, a)	There is an atom instance with id i and atom a
cycle(n)	The current cycle is n
plan(i, π)	There is a plan instance with id i and its plan remainder is π
init-plan(i, π)	There is a plan instance with id i and its initial plan is π
justification(i, j)	There is a plan instance with id i and an atom instance with j where the atom instance is the justification of the plan instance
substitution(i, θ)	There is a plan instance with id i and its substitution is θ
subgoal(i, j)	There is a plan instance with id i and an atom instance with j where the atom instance is the subgoal of the plan instance
state(i, s)	There is a plan instance with id i and its substitution is θ
cycle(i, n)	There is an atom or plan instance existed since cycle n with id i

Meta actions

Meta action	Meaning
`add-atom`(a)	To add a new atom instance of an atom a
`delete-atom`(i, a)	To delete atom instance(s) of which id is i and atom is a
`set-plan`(i, π)	To replace the plan remainder of a plan instance (with id i) with π
`set-substitution`(i, θ)	To extend the subsitution of a plan instance (with id i) with θ
`set-state`(i, s)	To add the flag s to the state of a plan instance with id i
`unset-state`(i,s)	To delete the flag s from the state of a plan instance with id i
`delete-plan`(i)	To delete a plan instance with id i

Clauses

$$q \leftarrow [not]q_1, \ldots, [not]q_n$$

Macros

$$a = a_1; \ldots; a_n$$

Object level rules

$$reason[: context] \rightarrow \pi$$

$$reason ::= q_1, \ldots, q_n$$

where q_i's are primitive mental states queries (i.e., queries of the from atom(i, a) or cycle(i, a)).

$$context ::= [not]q_1, \ldots, [not]q_n$$

where q_i's are mental states queries (i.e., queries of the from atom(i, a) or cycle(i, a) or user-defined queries by clauses using only primitive queries atom(i, a) or cycle(i, a)).

Meta rules

$$context \rightarrow \pi$$

$$context ::= [not]q_1, \ldots, [not]q_n$$

where q_i's are queries.

Meta-APL agent

A **meta-APL** agent is a tuple:

$$(A, Q, M, R_1, \ldots, R_n)$$

where A is a finite set of atoms, Q is a finite set of clauses, M is a finite set of macros, $n \leq 1$ and R_i's are sets of either object level rules or meta rules.

Appendix B

A computation run of the clean robot

We illustrate the operational semantics of **meta-APL** by revisiting our example in Section 3.5 of the previous chapter. In particular, we present a short computation run of the first two cycles of the agent program for the service robot.

From Example 4.1.5, we know that the initial configuration of our service robot is as follows:

$$C_0 = \langle \{(1, belief(pos(room_2)), nil, 0)\}, \varnothing, 0, 0 \rangle$$

We represent configurations as the following table:

Mental state			C_0
$(1, belief(pos(room_2)), nil, 0)$			
Plan state			
– empty –			
Phase counter:	0	Cycle counter:	0

On the top right corner is the name of the configuration. The second row of the table is the content of the mental state. Then, on the fourth row, we have the content of the

plan state. Finally, the last line is for showing the values of the phase counter and the cycle counter. In the following, we show the run by listing a sequence of configurations as they appear, where some uninteresting ones are omitted. We also highlight differences between consecutive configurations in bold.

B.1 First cycle

From C_0, as the phase counter is 0, SENSE is the only transition from C_0. Its effect is for the robot to perform *sense*() which collects information from the environment. We assume that, at this point, through perceptions, the robot knows that room $room_1$ is dirty and there is a box at room $room_3$ to be delivered to room $room_2$. Hence, this transition leads to the following configuration:

Mental state		C_1
$(1, belief(pos(room_2)), nil, 0)$ $\boldsymbol{(2, belief(dirty(room_1)), nil, 0)}$ $\boldsymbol{(3, belief(box(room_3)), nil, 0)}$ $\boldsymbol{(4, belief(dest(room_2)), nil, 0)}$		
Plan state		
–empty–		
Phase counter: **1**	Cycle counter:	0

The phase counter is now 1 in C_1. The agent starts applying meta rules in R_1. Here, only the meta rule (3.9) is applicable in C_1. The result of the transition META-APPLY-1 is a new atom instance of $goal(box(room_2))$. Thus, the new atom enables the meta rule (3.10) and applying this meta rule corresponds to another transition META-APPLY-1 which deletes the atom instance with id 4. After that, no other meta rules in R_1 are applicable, which makes the transition META-APPLY-2 applicable where the phase counter is increased. We obtain the following configuration:

Mental state		C_2
$(1, belief(pos(room_2)), nil, 0)$ $(2, belief(dirty(room_1)), nil, 0)$ $(3, belief(box(room_3)), nil, 0)$ $\mathbf{(5, goal(box(room_2)), nil, 0)}$		
Plan state		
–empty–		
Phase counter:	**2**	Cycle counter: 0

Note that we ignore intermediate configurations between C_1 and C_2 as it is similar to C_2 except of the value of the phase counter. In C_2, the phase counter now is 2. This means the agent will try to apply as many object-level rules in R_2 as possible. Here, the object level rules (3.13) and (3.14) are applicable because of atom instances with id 2, 3 and 5. They can be applied from C_2 by following two consecutive transitions OBJ-APPLY-1 which generates two following plan instances:

$$(6, !pos(room_1); Vacuum(), !pos(room_1); Vacuum(), \{X/room_1\}, \varnothing, \{2\}, 0)$$

$$(7, !pos(room_3); Pick(); !pos(room_2); Drop(),$$
$$!pos(room_3); Pick(); !pos(room_2); Drop(), \{X/room_2, Y/room_3\}, \varnothing, \{5\}, 0)$$

Then, no more rules from R_2 are applicable which enables OBJ-APPLY-2 as the next transition, and yields the following configuration:

Mental state	C_3
$(1, belief(pos(room_2)), nil, 0)$ $(2, belief(dirty(room_1)), nil, 0)$ $(3, belief(box(room_3)), nil, 0)$ $(5, goal(box(room_2)), nil, 0)$	
Plan state	

$(\mathbf{6}, !pos(room_1); Vacuum(), !pos(room_1); Vacuum(), \{X/room_1\}, \varnothing, \{2\}, 0)$ $(\mathbf{7}, !pos(room_3); Pick(); !pos(room_2); Drop(),$ $!pos(room_3); Pick(); !pos(room_2); Drop(),$ $\{X/room_2, Y/room_3\}, \varnothing, \{5\}, 0)$	
Phase counter: **3**	Cycle counter: 0

Notice that in C_3, the atom instance with id 2 is the justification of the plan instance with id 6, and the atom instance with id 5 is the justification of the plan instance with id 7.

As the phase counter is 3 in C_3, the agent now tries to apply meta rules from R_3. Recall that R_3 is for selecting an intention to be executed in the current cycle. Since there is no intention in the plan state, we have that (3.16) is applicable for both plan instances with ids 6 and 7, which enables two META-APPLY-1 transitions from C_3. After transition, this meta rule is not applicable anymore since we will set either the plan instance with id 6 or 7 to be an intention. In this example, we assume that the run of the agent selects the plan instance with id 7 to apply (3.16). Then, we only have (3.18) applicable because of the intention with id 7. By another META-APPLY-1 transition, we obtain a configuration where no other meta rule is applicable. This enables the META-APPLY-2 transition, and we arrive at the following configuration:

Mental state C_4
$(1, belief(pos(room_2)), nil, 0)$ $(2, belief(dirty(room_1)), nil, 0)$ $(3, belief(box(room_3)), nil, 0)$ $(5, goal(box(room_2)), nil, 0)$
Plan state

$(6, !pos(room_1); Vacuum(), !pos(room_1); Vacuum(), \{X/room_1\}, \varnothing, \{2\}, 0)$ $(7, !pos(room_3); Pick(); !pos(room_2); Drop(),$ $\quad !pos(room_3); Pick(); !pos(room_2); Drop(),$ $\quad \{X/room_2, Y/room_3\}, \{\textbf{intended}, \textbf{scheduled}\}, \{5\}, 0)$
Phase counter: **4** | Cycle counter: 0

At C_4, the agent is in the first stage of the Exec phase where there is only a transition DEL-STEPPED. Since no intention was executed in the previous cycle, i.e., plan instances with the flag `stepped`, we arrive at the same configuration except that the phase counter is increased by 1:

Mental state C_5
$(1, belief(pos(room_2)), nil, 0)$ $(2, belief(dirty(room_1)), nil, 0)$ $(3, belief(box(room_3)), nil, 0)$ $(5, goal(box(room_2)), nil, 0)$
Plan state
$(6, !pos(room_1); Vacuum(), !pos(room_1); Vacuum(), \{X/room_1\}, \varnothing, \{2\}, 0)$ $(7, !pos(room_3); Pick(); !pos(room_2); Drop(),$ $\quad !pos(room_3); Pick(); !pos(room_2); Drop(),$ $\quad \{X/room_2, Y/room_3\}, \{\texttt{intended}, \texttt{scheduled}\}, \{5\}, 0)$
Phase counter: **5** | Cycle counter: 0

As the phase counter is 5, we are at the final phase of the first cycle which is to execute selected intentions. In the plan state, there is only one intention with id 7 is selected, i.e., having the flag `scheduled`. We execute the first action of its plan which is a subgoal action. The effect is a new subgoal linked back to the intention. After that, there are no more intentions to execute, hence, the next transition is labelled NEW-CYCLE which resets the phase counter and increases the cycle counter. We obtain the following configuration:

Mental state	C_6
$(1, belief(pos(room_2)), nil, 0)$ $(2, belief(dirty(room_1)), nil, 0)$ $(3, belief(box(room_3)), nil, 0)$ $(5, goal(box(room_2)), nil, 0)$ $(8, goal(pos(room_3)), 7, 0)$	
Plan state	
$(6, !pos(room_1); Vacuum(), !pos(room_1); Vacuum(), \{X/room_1\}, \varnothing, \{2\}, 0)$ $(7, !pos(room_3); Pick(); !pos(room_2); Drop(),$ $!pos(room_3); Pick(); !pos(room_2); Drop(),$ $\{X/room_2, Y/room_3\}, \{\texttt{intended}, \textbf{stepped}\}, \{5\}, 0)$	
Phase counter: **0**	Cycle counter: **1**

As the cycle counter increases, C_6 is now the beginning of the second deliberation cycle.

B.2 Second cycle

From C_6, there is a transition SENSE to the following cycle:

Mental state	C_7
$(1, belief(pos(room_2)), nil, 0)$ $(2, belief(dirty(room_1)), nil, 0)$ $(3, belief(box(room_3)), nil, 0)$ $(5, goal(box(room_2)), nil, 0)$ $(8, goal(pos(room_3)), 7, 0)$ $\mathbf{(9, belief(dest(room_2)), nil, 0)}$	
Plan state	

$(6, !pos(room_1); Vacuum(), !pos(room_1); Vacuum(), \{X/room_1\}, \varnothing, \{2\}, 0)$ $(7, !pos(room_3); Pick(); !pos(room_2); Drop(),$ $!pos(room_3); Pick(); !pos(room_2); Drop(),$ $\{X/room_2, Y/room_3\}, \{\texttt{intended}, \texttt{stepped}\}, \{5\}, 0)$
Phase counter: **1** | Cycle counter: 1

Note that we still receive a belief from the environment as the box has not arrived at the expected room yet. Therefore, in C_7, we have a new atom instance of $belief(dest(room_2))$. It is the same as the atom instance with id 4 which was deleted in the previous cycle. The agent at C_7 tries to apply rules from R_1 as the phase counter is 1. There is only (3.10) is applicable which effectively deletes the new atom instance as there is already an atom instance with the form $goal(box(room_2))$ in the mental state. By two transitions labelled META-APPLY-1 and META-APPLY-2, we arrive at the following configuration:

Mental state C_8
$(1, belief(pos(room_2)), nil, 0)$ $(2, belief(dirty(room_1)), nil, 0)$ $(3, belief(box(room_3)), nil, 0)$ $(5, goal(box(room_2)), nil, 0)$ $(8, goal(pos(room_3)), 7, 0)$
Plan state
$(6, !pos(room_1); Vacuum(), !pos(room_1); Vacuum(), \{X/room_1\}, \varnothing, \{2\}, 0)$ $(7, !pos(room_3); Pick(); !pos(room_2); Drop(),$ $!pos(room_3); Pick(); !pos(room_2); Drop(),$ $\{X/room_2, Y/room_3\}, \{\texttt{intended}, \texttt{stepped}\}, \{5\}, 0)$
Phase counter: **2** | Cycle counter: 1

Now, the agent tries to apply rules in R_2. The object-level rule (3.15) is applicable because of the new subgoal of $goal(pos(room_3))$. Therefore, we have a transition labelled OBJ-APPLY-1 from C_8. Then, there is no other object-level rule in R_2 is applicable, the

next transition is OBJ-APPLY-2 which leads to the following configuration:

Mental state	C_9
$(1, belief(pos(room_2)), nil, 0)$ $(2, belief(dirty(room_1)), nil, 0)$ $(3, belief(box(room_3)), nil, 0)$ $(5, goal(box(room_2)), nil, 0)$ $(8, goal(pos(room_3)), 7, 0)$	
Plan state	
$(6, !pos(room_1); Vacuum(), !pos(room_1); Vacuum(), \{X/room_1\}, \varnothing, \{2\}, 0)$ $(7, !pos(room_3); Pick(); !pos(room_2); Drop(),$ $!pos(room_3); Pick(); !pos(room_2); Drop(),$ $\{X/room_2, Y/room_3\}, \{\texttt{intended}, \texttt{stepped}\}, \{5\}, 0)$ $\mathbf{(10, Go(); !goal(pos(room_3)), Go(); !goal(pos(room_3)),}$ $\mathbf{\{X/room_3\}, \varnothing, \{9\}, 1)}$	
Phase counter: **3**	Cycle counter: 1

From C_9, the agent is in phase 3 where it tries to apply meta rules from R_3. Since there is already an intention in the plan state, the meta rule (3.16) is not applicable. However, (3.17) is applicable because of the new plan instance with id 10 and the subgoal with id 8. Then, there is a transition labelled META-APPLY-1, which adds the flag `intended` to the plan instance with id 10. After that, there is another transition also labelled META-APPLY-1 corresponding to the application of the meta rule (3.18) which schedules the plan instance with id 10 to be executed in this deliberation cycle. Then, there is no other rules applicable, which enables a transition labelled META-APPL-2 and gives us the following configuration:

Mental state	C_{10}

$(1, belief(pos(room_2)), nil, 0)$ $(2, belief(dirty(room_1)), nil, 0)$ $(3, belief(box(room_3)), nil, 0)$ $(5, goal(box(room_2)), nil, 0)$ $(8, goal(pos(room_3)), 7, 0)$
Plan state
$(6, !pos(room_1); Vacuum(), !pos(room_1); Vacuum(), \{X/room_1\}, \varnothing, \{2\}, 0)$ $(7, !pos(room_3); Pick(); !pos(room_2); Drop(),$ $!pos(room_3); Pick(); !pos(room_2); Drop(),$ $\{X/room_2, Y/room_3\}, \{\texttt{intended}, \texttt{stepped}\}, \{5\}, 0)$ $(10, Go(); !goal(pos(room_3)), Go(); !goal(pos(room_3)),$ $\{X/room_3\}, \{\textbf{intended}, \textbf{scheduled}\}, \{9\}, 1)$
Phase counter: **4** | Cycle counter: 1

From C_{10}, the agent is in the Prepare phase. There is a transition labelled DEL-STEPPED which removes all flags `stepped` from the plan state. We obtain the following configuration:

Mental state C_{11}
$(1, belief(pos(room_2)), nil, 0)$ $(2, belief(dirty(room_1)), nil, 0)$ $(3, belief(box(room_3)), nil, 0)$ $(5, goal(box(room_2)), nil, 0)$ $(8, goal(pos(room_3)), 7, 0)$
Plan state
$(6, !pos(room_1); Vacuum(), !pos(room_1); Vacuum(), \{X/room_1\}, \varnothing, \{2\}, 0)$ $(7, !pos(room_3); Pick(); !pos(room_2); Drop(),$ $!pos(room_3); Pick(); !pos(room_2); Drop(),$ $\{X/room_2, Y/room_3\}, \{\texttt{intended}\}, \{5\}, 0)$ $(10, Go(); !goal(pos(room_3)), Go(); !goal(pos(room_3)),$ $\{X/room_3\}, \{\texttt{intended}, \texttt{scheduled}\}, \{9\}, 1)$
Phase counter: **5** | Cycle counter: 1

Since the phase counter of C_{11} is 5, the agent is now in the Exec phase for the second time. It will execute the only intention with the flag `scheduled` which has id 10. The first action of the plan of the intention with id 10 is the external action $Go()$ which will effectively move the position of the agent in the building from $room_2$, where it is currently, to $room_3$, which is the next one. This corresponds to a transition labelled EXEC-EA. As there are no more intentions to execution, there is the next transition labelled NEW-CYCLE resets the phase counter and increases the cycle counter. We arrive that the following configuration:

Mental state	C_{12}
$(1, belief(pos(room_2)), nil, 0)$ $(2, belief(dirty(room_1)), nil, 0)$ $(3, belief(box(room_3)), nil, 0)$ $(5, goal(box(room_2)), nil, 0)$ $(8, goal(pos(room_3)), 7, 0)$	
Plan state	
$(6, !pos(room_1); Vacuum(), !pos(room_1); Vacuum(), \{X/room_1\}, \varnothing, \{2\}, 0)$ $(7, !pos(room_3); Pick(); !pos(room_2); Drop(),$ $!pos(room_3); Pick(); !pos(room_2); Drop(),$ $\{X/room_2, Y/room_3\}, \{\texttt{intended}\}, \{5\}, 0)$ $(10, Go(); !goal(pos(room_3)), !goal(pos(room_3)),$ $\{X/room_3\}, \{\texttt{intended}, \textbf{stepped}\}, \{9\}, 1)$	
Phase counter: **0**	Cycle counter: 2

This configuration is the end of the second cycle, and also the beginning of the new cycle. Note that the effect of the external action $Go()$ has not been known by the robot yet. The agent will get its new position after the first transition in the third cycle.