Chapter 6

Correctness of Multi-Agent Programs: A Hybrid Approach

M. Dastani and J.-J. Ch. Meyer

Abstract This chapter proposes a twofold approach for ensuring the correctness of BDI-based agent programs. On the one hand, we advocate the alignment of the semantics of agent programming languages with agent specification languages such that for an agent programming language it can be shown that it obeys specific desirable properties expressed in the corresponding agent specification language. In this way, one can guarantee that specific properties expressed in the specification language are satisfied by any program implemented in the programming language. On the other hand, we introduce a debugging framework to find and resolve possible defects in such agent programs. The debugging approach consists of a specification language and a set of debugging tools. The specification language allows a developer to express cognitive and temporal properties of multi-agent program executions. The debugging tools allow a developer to verify if a specific multi-agent program execution satisfies a desirable property.

M. Dastani, J.-J. Ch. Meyer
Utrecht University, The Netherlands e-mail: {mehdi,jj}@cs.uu.nl

M. Dastani et al. (eds.), *Specification and Verification of Multi-agent Systems*,
DOI 10.1007/978-1-4419-6984-2_6,

6.1 Introduction

A promising approach to develop computer programs for complex and concurrent applications are multi-agent systems. In order to implement multi-agent systems, various agent-oriented programming languages and development tools have been proposed [64, 65]. These agent-oriented programming languages facilitate the implementation of individual agents and their interactions. A special class of these programming languages aims at programming *BDI-based* multi-agent systems, i.e., multi-agent systems in which individual agents are programmed in terms of cognitive concepts such as beliefs, events, goals, plans, and reasoning rules [63, 122, 224, 343, 433].

There is a whole range of approaches to the correctness of multi-agent programs. While some of them are abstract verification frameworks for agent programs [60, 372], a majority of these approaches fall under the heading of either model checking [21, 61, 371] or theorem proving [6–8] methods. This chapter concerns the correctness of *BDI-based* multi-agent programs by proposing two techniques that are complementary to those of model-checking and theorem proving. These two approaches are somewhat dual to each other, and are the following: (1) we show how to prove properties that are *general* in the sense that they hold for *any* execution of *any* program written in the agent programming language at hand, and (2) we show how to verify the correctness of *specific* execution of *specific* program written in the agent programming language at hand by proposing a *debugging* approach with which it is possible to check whether run-time executions of programs are still in line with the specification, so still according to certain desirable properties (what we may call a 'so far so good?' approach of verification). Moreover, as we will see, both approaches are particularly targeted at programs written in a BDI-style programming language.

So let us first turn to the issue of proving certain properties that do not depend on the particular program but rather on the agent programming language and its interpretation. To this end we define the semantics of the programming language as well as an agent specification language such that we can express these general properties and prove them with respect to the semantics of the programming language. We will establish this such that the semantics of the specification language is aligned with that of the programming language in a systematic and natural way! We show that an agent programming language obeys some desirable properties expressed in an agent specification language, i.e., that any individual agent implemented by the programming language satisfies the desirable property expressed in the specification language. Note that this is an important issue raised in the literature of agent program correctness. As many other approaches to the correctness of agents are based on rather abstract agent (BDI-like) logics [110, 360, 385], it is not always clear how the abstract BDI notions appearing in these logics, often treated by means of modal operators, relate to notions rooted in actual computation. This is referred to by Wooldridge as the problem of *ungrounded semantics* for agent specification languages [439].

There is a number of proposals in the literature to ground agent logics in the actual computation of agents (e.g., [130, 219]). In these approaches, it is attempted to ground agent logics by rendering the notions that occur in these logics such as beliefs and goals less abstract and more computational, so that reasoning about these notions becomes relevant for reasoning about agent programs. In the current paper, we follow the line of [130] by connecting the semantics of agent logics directly to the operational semantics of agent programming languages. More precisely, the modal accessibility relations in Kripke models of an agent logic are associated with the (operational) semantics of the programming language at hand. Furthermore, it is important to realize that this problem is different from the model checking problem. In model checking, the problem is to verify if a certain property, expressed in a specification language such as the language of a BDI logic, holds for *all* executions of one *specific* agent program (and not for every agent program written in the same programming language). Thus, in contrast to model checking, where one verifies that all executions of a particular agent program satisfy a property expressed in a specification language, we are here interested only in certain general properties of individual agents such as such as commitment strategies, e.g., *an agent will not drop its goals until it believes it has achieved them*.

Secondly in this chapter we propose a debugging approach to check the correctness of a *specific* execution of a *specific* agent program. Debugging is the art of finding and resolving errors or possible defects in a computer program. Here, we will focus on the semantic bugs in BDI-based multi-agent programs and propose a generic approach for debugging such programs [63, 84, 114, 121, 122, 343, 345, 346, 425]. In particular, we propose a specification language to express execution properties of multi-agent programs and a set of debugging actions/tools to verify if a specific program execution satisfies a specific property. The expressions of the specification language are related to the proposed debugging actions/tools by means of special programming constructs. Using these constructs in a multi-agent program ensures that the debugging actions/tools are performed/activated as soon as their associated properties hold for the multi-agent program execution at hand. In order to illustrate how a developer can debug both *cognitive* and *temporal* aspects of the (finite) executions of multi-agent programs we discuss a number of examples. They show how the debugging constructs allow a developer to log specific parts of the cognitive state of individual agent programs (e.g., log the beliefs, events, goals, or plans) from the moment that specific condition holds, stop the execution of multi-agent programs whenever a specific cognitive condition holds, or check whether an execution trace of a multi-agent program (a finite sequence of cognitive states) satisfies a specific (cognitive/temporal) property.

Interestingly, if we compare the two approaches in this chapter we will see that although we use temporal logic for both, there are notable differences. The two most important ones being: (1) In the 'general' program-independent approach we use branching-time temporal logic, since we have to reason about behaviour in general, i.e. all possible executions of all programs, including explicit representations of agents' choices, while in the debugging approach we use linear-time temporal logic, since we are only interested in a particular execution, viz. the execution at hand, of

one particular program. (2) In the 'general' approach we use temporal models with infinite branches/paths, as usual in the temporal reasoning about programs, while in the debugging approach we use temporal models with only finite paths, reflecting the fact that we are looking at run-time (mostly unfinished) executions.

The structure of this chapter is as follows. First we present a simple but extendable BDI-based agent-oriented programming language that provides constructs to implement concepts such as beliefs, goals, plans, and reasoning rules. The syntax and semantics of this programming language is presented in section 6.2. In the rest of the chapter we focus on the correctness of programs of this programming language. We start by treating the 'general' approach of verifying program-independent (but semantics-dependent) properties, i.e., properties of the semantics of the programming language, and not of any specific program. To this end, we present in section 6.3 an agent specification language which is closely related to well-known BDI specification languages. This specification language consists of (modal) operators for beliefs, goals, and time such that properties such as commitment strategies can be expressed. In section 6.4 we show that all agents that are implemented in the proposed agent programming language satisfy the desirable properties that are specified in the specification language. Then we turn to the 'specific' approach of verifying execution-dependent properties by presenting a debugging approach for the same simple programming language. We first define a temporal specification language to express execution properties that we aim to verify. Special attention is given to the (non-standard) semantics of the specification language since its formula are evaluated on (finite) execution paths of multi-agent programs. Next we present our proposal of a tool set for debugging multi-agent programs. Finally, we discuss some related works on debugging multi-agent programs and conclude the chapter.

6.2 An agent-oriented Programming Language *APL*

In this section, we propose the syntax and operational semantics of a simple but prototypical logic-based agent-oriented programming language that provides programming constructs to implement agents in terms of cognitive concepts such as beliefs, goals, and plans. In order to focus on the relation between such a programming language and its related specification language and without loss of generality, we ignore some aspects that may be relevant or even necessary for the practicality and effectiveness of the programming language. The presented programming language is thus not meant to be practically motivated, but rather to illustrate how such a logic- and BDI-based programming language can be connected to logical specification languages and how to examine the correctness of the related programs. This aim is accomplished by defining the syntax and semantics of the programming language similar to those of the existing practical agent-oriented programming languages (e.g., 2APL, 3APL, GOAL, and Jason [64, 122, 129, 224]). This makes our approach applicable to the existing logic- and BDI-based agent programming lan-

guages. A concrete extension of this agent-oriented programming language is studied in [130].

6.2.1 Syntax of *APL*

A multi-agent program comprises a set of individual agent programs, each of which is implemented in terms of concepts such as beliefs, goals, and plans. In this section, we focus on a programming language for implementing individual agents and assume that a multi-agent program is a set of programs, each of which implements an individual agent. The presented agent-oriented programming language provides programming constructs for beliefs, goals, plans, and planning rules. We use a propositional language to represent an agent's beliefs and goals while plans are assumed to consists of actions that can update the agent's beliefs base. It is important to note that these simplifications do not limit the applicability of the proposed model.

The idea of an agent's belief is to represent the agent's information about the current state of affairs. The idea of an (achievement) goal is to reach a state that satisfies it. An agent is then expected to generate and execute plans to achieve its goal. The emphasis here is that the goal will not be dropped until a state is reached that satisfies it. An example of an achievement goal is to have fuel in car (`fuel`) for which the agent can generate either a plan to fuel at gas station 1 (`gs1`) or a plan to fuel at gas station 2 (`gs2`). The achievement goal will be dropped as soon as the agent believes that it has fuel in its car, i.e., as soon as it believes `fuel`. In our running example, a car driving agent believes he is in position 1 (`pos1`) and has the goal to fuel (`fuel`).

Definition 6.1. (belief and goal languages) Let L_p be a propositional language. The belief language is denoted by $L_\sigma \subseteq L_p$ and the goal language is denoted by $L_\gamma \subseteq L_p$.

For the purpose of this chapter, we assume a set of plans Plan each of which is executed atomically, i.e., a plan can be executed in one computation step. Moreover, agents are assumed to generate their plans based on their beliefs and goals. The planning rules indicate which plans are appropriate to be selected when the agent has a certain goal and certain beliefs. A planning rule is of the form $\beta, \kappa \Rightarrow \pi$ and represents the possibility to select plan π for the goal κ, if the agent believes β. In order to be able to check whether an agent has a certain belief or goal, we use propositional formulas from L_p to represent belief and goal query expressions.

Definition 6.2. (planning rule) Let Plan be the set of plans that an agent can use. The set of planning rules $\mathcal{R}_{\mathsf{PL}}$ is defined as:

$$\mathcal{R}_{\mathsf{PL}} = \{\beta, \kappa \Rightarrow \pi \mid \beta \in L_\sigma, \kappa \in L_\gamma, \pi \in \mathsf{Plan}\}$$

In the rest of the paper, **Goal**(r) and **Bel**(r) are used to indicate, respectively, the goal condition κ and the belief condition β of the planning rule $r = (\beta, \kappa \Rightarrow \pi)$. In the running example, the agent has two planning rules `pos1,fuel⇒gs1` and `pos2,fuel⇒gs2`. The first (second) planning rule indicates that the agent should fuel at gas station 1 (2) if he wants to fuel and believes that he is in position `pos1` (`pos2`). Given these languages, an agent can be implemented by programming two sets of propositional formulas (representing the agent's beliefs and goals), and one set of planning rules. For the purpose of this chapter, we assume that agents cannot have initial plans, but generate them during their execution.

Definition 6.3. (agent program) Let Id be the set of agent names. An individual agent program is a tuple $(\iota, \sigma, \gamma, \mathsf{PL})$, where $\iota \in Id$, $\sigma \subseteq L_\sigma, \gamma \subseteq L_\gamma$ and $\mathsf{PL} \subseteq \mathcal{R}_{\mathsf{PL}}$. A multi-agent program is a set of such tuples, i.e., $\{(\iota, \sigma, \gamma, \mathsf{PL}) \mid \iota \in Id, \sigma \subseteq L_\sigma, \gamma \subseteq L_\gamma, \mathsf{PL} \subseteq \mathcal{R}_{\mathsf{PL}}\}$.

The individual agent program for our running example is the tuple $(c, \sigma, \gamma, \mathsf{PL})$, where c is the name of the car agent, $\sigma = \{$`pos1`$\}$, $\gamma = \{$`fuel`$\}$, and $\mathsf{PL} = \{$`pos1, fuel⇒gs1, pos2,fuel⇒gs2`$\}$. Note that the agent believes it is in position 1 and has the goal to fuel.

6.2.2 Semantics of APL

The operational semantics of the multi-agent programming language is presented in terms of a transition system. A transition system is a set of derivation rules for deriving transitions. A transition is a transformation of one state into another and it corresponds to a single computation step. For the semantics of the multi-agent programming language a transition is a transformation of one multi-agent configuration (state) into another. The operational semantics of the multi-agent programming language is directly defined in terms of the operational semantics of individual agent programming language.

6.2.2.1 Agent Configuration

An agent's configuration denotes the state of the agent at one moment in time. It is determined by its mental attitudes, i.e., by its beliefs, goals, plans, and reasoning rules. A multi-agent configuration denotes the state of all agents at one moment in time.

Definition 6.4. (configuration) Let $\models_p$ be the classical propositional entailment relation (used also in the rest of the chapter). Let $\Sigma = \{\sigma \mid \sigma \subseteq L_\sigma, \sigma \not\models_p \bot\}$ be the set of possible consistent belief bases and $\Gamma = \{\phi \mid \phi \in L_\gamma, \phi \not\models_p \bot\}$ be the set of goals.

A configuration of an agent is a tuple $\langle \iota, \sigma, \gamma, \Pi, \mathsf{PL} \rangle$, where ι is the agent's identifier, $\sigma \in \Sigma$ is its belief base, $\gamma \subseteq \Gamma$ is its goal base, $\Pi \subseteq (L_\gamma \times L_\gamma \times \mathsf{Plan})$ is its plan base, and $\mathsf{PL} \subseteq \mathcal{R}_{\mathsf{PL}}$ includes its planning rules. The set of all agent configurations is denoted by $\mathcal{A}$. A multi-agent configuration is a subset of $\mathcal{A}$, i.e., a multi-agent configuration is a set of individual agent configurations.

In the sequel, we use $\langle \sigma_\iota, \gamma_\iota, \Pi_\iota, \mathsf{PL}_\iota \rangle$ instead of $\langle \iota, \sigma, \gamma, \Pi, \mathsf{PL} \rangle$ to keep the representation of agent configurations simple. In the above definition, it is assumed that the belief base of an agent is consistent since otherwise the agent can believe everything which is not a desirable property. Also, each goal is assumed to be consistent since otherwise an agent should achieve an impossible state. Finally, the elements of the plan base are defined as 3-tuples $(L_p \times L_p \times \mathsf{Plan})$ consisting of a plan and two goals (propositional formulas) that indicate the reasons for generating the plan. More specifically, (ϕ, κ, π) is added to an agent's plan base if the planning rule $\beta, \kappa \Rightarrow \pi$ is applied because κ is a subgoal of the agent's goal ϕ. Note that the planning rule can be applied only if κ is a subgoal of an agent's goal ϕ. This means that we have $\forall (\phi, \kappa, \pi) \in \Pi : \ \phi \models_p \kappa$. This information will be used to avoid applying a planning rule if it is already applied and the generated plan is not fully executed. In our running example, the agent can apply both planning rules (depending on the agent's beliefs) since the goals of these rules (i.e., `fuel`) is a subgoal of the agent's goal `fuel`, i.e., since `fuel`$\models$`fuel`. Note that these rules could also be applied if the agent had more complex goals such as `fuel`$\wedge$`cw`; `cw` stands for car wash.

The initial configuration of an individual agent is based on the individual agent program (definition 6.3) that specifies the initial beliefs, goals, and planning rules. As noted, for the purpose of this chapter, we assume that an agent does not have initial plans, i.e., the initial plan base is empty. The initial multi-agent configuration is the set of initial configurations of individual agents.

Definition 6.5. (initial configuration) Let $\{(1, \sigma_1, \gamma_1, \mathsf{PL}_1), \ldots, (n, \sigma_n, \gamma_n, \mathsf{PL}_n)\}$ be a multi-agent program. Then, the initial configuration of the multi-agent program is $\{\langle \sigma_1, \gamma_1, \Pi_1, \mathsf{PL}_1 \rangle, \ldots, \langle \sigma_n, \gamma_n, \Pi_n, \mathsf{PL}_n \rangle\}$, where $\Pi_i = \emptyset$ for $i = 1, \ldots, n$.

In the following, we assume that all agents use one and the same set of planning rules PL (i.e., all agents use the same planning rule library) and that the set of planning rules does not change during the agent executions. For this reason, we do not include the set PL in the individual agent configurations and use $\langle \sigma, \gamma, \Pi \rangle$ instead of $\langle \sigma, \gamma, \Pi, \mathsf{PL} \rangle$. This means that an *APL* program specifies only the initial beliefs and goals of an agent.

6.2.2.2 Transition System $\mathcal{T}$

This subsection presents the transition system $\mathcal{T}$ which consists of transition rules (also called derivation rules) for deriving transitions between configurations. Each

transition rule has the following form indicating that the configuration C can be transformed to configuration C' if the condition of the rule holds.

$$\frac{condition}{C \rightarrow C'}$$

We first present three transition rules that transform the configurations of individual agent programs, followed by a transition rule that transform multi-agent configurations. The first three transition rules capture the successful execution of plans, the failed execution of plans, and the application of planning rules, respectively.

In order to define the transition rule for the application of planning rules, we define the notions of *relevant* and *applicable* planning rules w.r.t. an agent's goal and its configuration. Intuitively, a planning rule is relevant for an agent's goal if it can contribute to the agent's goal, i.e., if the goal that occurs in the head of the planning rule is a subgoal of the agent's goal. A planning rule is applicable to an agent's goal if it is relevant for that goal and its belief condition is entailed by the agent's configuration.

Definition 6.6. (relevant, applicable) Let $C = \langle \sigma, \gamma, \Pi, \mathsf{PL} \rangle$ be an agent configuration. Given configuration C containing goal $\phi \in \gamma$, the set of relevant and applicable planning rules are defined as follows:
- $rel(\phi, C) = \{r \in \mathsf{PL} \mid \phi \models_p \mathbf{Goal}(r)\}$
- $app(\phi, C) = \{r \in rel(\phi, C) \mid \sigma \models_p \mathbf{Bel}(r)\}$

In the following transition rules we write $\frac{app(\phi)}{C \rightarrow C'}$ instead of $\frac{app(\phi, C)}{C \rightarrow C'}$.

When executing an agent, planning rules will be selected and applied based on its beliefs, goals and plans. The application of planning rules generates plans which can subsequently be selected and executed. Before introducing the transition rules to specify possible agent execution steps, we need to define what it means to execute a plan. The execution of a plan affects the belief and goal bases. The effect of plan execution on the belief base is captured by an update operator `update`, which takes the belief base and a plan and generates the updated belief base. This update operator can be as simple as adding/deleting atoms to/from the belief base. We assume a partial function $\texttt{update} : (\mathsf{Plan} \times \Sigma) \rightarrow \Sigma$ that takes a plan and a belief base, and yields the belief base resulting from the execution of the plan on the input belief base (if the update is not successful, the update operation is undefined).

The first transition rule (R_1) captures the case where the plan π is successfully executed. The resulting configuration contains a belief base that is updated based on the executed plan, a goal base from which achieved goals are removed, and a plan base from which plans with associated achieved goal are removed.

Rule R_1 (Plan execution 1)

$$\frac{(\phi, \kappa, \pi) \in \Pi \ \&\ \texttt{update}(\sigma, \pi) = \sigma'}{\langle \sigma, \gamma, \Pi \rangle \rightarrow \langle \sigma', \gamma', \Pi' \rangle}$$

where
$\gamma' = \gamma \setminus \{\psi \mid \sigma' \models_p \psi\}$ and $\Pi' = \Pi \setminus (\{(\phi,\kappa,\pi)\} \cup \{(\phi',\kappa',\pi') \in \Pi \mid \sigma' \models_p \phi'\})$.

The second transition rule (R_2) captures the case that the performance of the plan has failed, i.e., the update operation is undefined. In this case, the failed plan (ϕ,ψ,π) will be removed from the plan base.

Rule R_2 (Plan execution 2)

$$\frac{(\phi,\kappa,\pi) \in \Pi \ \&\ \texttt{update}(\sigma,\pi) = undefined}{\langle \sigma,\gamma,\Pi\rangle \rightarrow \langle \sigma,\gamma,\Pi \setminus \{(\phi,\kappa,\pi)\}\rangle}$$

Plans should be generated to reach the state denoted by goals. If the generated and performed plans do not achieve the desired state, then the corresponding goal remains in the goal base. The first transition rule below (called R_3) is designed to apply planning rules in order to generate plans the execution of which may achieve the subgoals of the goals. A planning rule can be applied if the goal in the head of the rule is not achieved yet, if there is no plan for the same subgoal in the plan base (in order to avoid applying rules if it is already applied), and if the subgoal is not achieved yet. The application of a planning rule will add the plan of the planning rule to the plan base.

Rule R_3 (apply planning rules)

$$\frac{\phi \in \gamma \ \&\ (\beta,\kappa \Rightarrow \pi) \in app(\phi) \ \&\ \nexists \pi' \in \mathsf{Plan} : (\phi,\kappa,\pi') \in \Pi \ \&\ \sigma \not\models_p \kappa}{\langle \sigma,\gamma,\Pi\rangle \rightarrow \langle \sigma,\gamma,\Pi \cup \{(\phi,\kappa,\pi)\}\rangle}$$

We consider an execution of a multi-agent program as an interleaved execution of the involved individual agent programs. The following transition rule captures the parallel execution

Rule R_4 (multi-agent execution)

$$\frac{A_i \rightarrow A_i'}{\{A_1,\ldots,A_i,\ldots,A_n\} \rightarrow \{A_1,\ldots,A_i',\ldots,A_n\}}$$

In this and next sections, we focus on the semantics of individual agent programs as the semantics of multi-agent programs is a composition of the semantics of individual agent programs.

6.2.2.3 Agent Execution

In order to define all possible behaviours of an agent program and compare them with each other, we need to define what it means to execute an agent program. Given

a transition system consisting of a set of transition rules, the execution of an agent program is a set of transitions generated by applying the transition rules to the initial configuration of the program (i.e., initial beliefs and goals). Thus, the execution of an agent program starts with its initial configuration and generates subsequent configurations that can be reached from the initial configuration by applying transition rules. The execution of an agent program forms a graph in which the nodes are the configurations and the edges indicate the application of a transition rule (i.e., execution of a plan, or the application of a planning rule). In the following, we define the execution of an agent program A by first defining the set of all possible transitions $\mathcal{R}_\mathcal{T}$ for all possible agents given a transition system $\mathcal{T}$, and then take the subset of those transitions that can be reached from the initial configuration of agent A.

Definition 6.7. (agent execution) Recall that $\mathcal{A}$ be the set of all agent configurations. Then, the set of transitions that are derivable from a transition system $\mathcal{T}$, denoted as $\mathcal{R}_\mathcal{T}$, is defined as follows:

$$\mathcal{R}_\mathcal{T} = \{(c_i, c_j) \mid c_i \rightarrow c_j \text{ is a transition derivable from } \mathcal{T} \ \& \ c_i, c_j \in \mathcal{A}\}$$

Given an agent program A with corresponding initial configuration c_0, the execution of A is the smallest set $\mathcal{E}_\mathcal{T}(A)$ of transitions derivable from $\mathcal{T}$ starting from c_0, i.e., it is the smallest subset $\mathcal{E}_\mathcal{T}(A) \subseteq \mathcal{R}_\mathcal{T}$ such that:

- if $(c_0, c_1) \in \mathcal{R}_\mathcal{T}$, then $(c_0, c_1) \in \mathcal{E}_\mathcal{T}(A)$, for $c_1 \in \mathcal{A}$
- if $(c_i, c_j) \in \mathcal{E}_\mathcal{T}(A)$ and $(c_j, c_k) \in \mathcal{R}_\mathcal{T}$, then $(c_j, c_k) \in \mathcal{E}_\mathcal{T}(A)$, for $c_i, c_j, c_k \in \mathcal{A}$

6.3 CTL_{apl}: A Specification Language for Agent Programs

In the area of agent theory and agent-oriented software systems, various logics have been proposed to characterize the behavior of rational agents. The most cited logics to specify agents' behavior are the BDI logics [110, 315, 360, 385]. These logics are multi-modal logics consisting of temporal and epistemic modal operators. In the BDI logics, the behaviour of an agent is specified in terms of the temporal evolution of its mental attitudes (i.e., beliefs, desires, and intentions) and their interactions. These logics are characterized by means of axioms and inference rules to capture the desired static and dynamic properties of agents' behaviour. In particular, the axioms establish the desired properties of the epistemic and temporal operators as well as the rational balance between them. For example, the axioms to capture the desired static properties of beliefs are $KD45$ (the standard weak $S5$ system), for desires and intentions are KD, and for the rational balance between beliefs and desires are various versions of *realism*. Moreover, some desired dynamic properties of agents' behaviour are captured through axioms that implement various versions of the *commitment strategies*. These axioms are defined using temporal operators expressing when and under which conditions the goals and intentions of agents can

be dropped. For example, an agent can be specified to either hold its goals until it has achieved it (blindly-committed agent type), or drop the goal if it believes that it can *never* achieve it (single-minded agent type) [110, 360].

A main concern in designing and developing agent-oriented programming languages is to provide programming constructs in such a way that their executions generate the agent behaviours having the same desirable properties as in their specifications. This implies that the semantics of the programming languages should be defined in such a way to satisfy the desirable properties captured by means of the axioms in the BDI logics. The main issue addressed in this part of the chapter is how agent specification logics, which are used to specify the agents' behaviour, can be related to agent-oriented programming languages, which are used to implement agents. We study this relation by proposing an instantiation of the BDI_{CTL} logic [360, 385] with a declarative semantics that is aligned with the operational semantics of the programming language *APL* as proposed in section 6.2. We then show that this alignment enables us to prove that certain properties expressed in the specification language are satisfied by the programming language. The specification language is a multi-modal logic consisting of temporal modal operators to specify the evolution of agents' configurations (the agents' execution) through time and epistemic modal operators to specify agents' mental state (beliefs and goals) in each configuration. In order to relate the specification and programming languages, we do not allow the nesting of epistemic operators. This is because the beliefs and goals in the agent programming language *APL*, presented in section 6.2, are propositional rather than modal formulas. This is, however, not a principle limitation as the representation of beliefs and goals in agent programming languages can be extended to modal formulas [437].

In the rest of this section and in section 6.4, we only consider the specification and properties of single agent programs. The proposed specification language can be extended for multi-agent programs in an obvious way because individual agent programs do not interact. The individual agent programs do not communicate or share an environment as their actions are limited to local belief and goal changes.

6.3.1 CTL_{apl} Syntax

The behaviour of an agent, generated by the execution of the agent, is a temporal structure over its mental states. In order to specify the mental state of agents, we will define the language L consisting of non-nested belief and goal formulas.

Definition 6.8 (specification language L). The language L for the specification of agents' mental attitudes consists of non-nested belief and goal formulas, defined as follows: if $\phi \in L_p$, then $B(\phi)$, $G(\phi) \in L$.

We then use the standard CTL^* logic [158] in which the primitive propositions are formulas from the language L. The resulting language will be called CTL_{apl} defined as follows.

Definition 6.9 (specification language CTL_{apl}). The state and path formulas are defined by the following S and P clauses, respectively.

- (S1) Each formula from L is a state formula.
- (S2) If ϕ and ψ are state formulas, then $\phi \wedge \psi$ and $\neg\phi$ are also state formulas.
- (S3) If ϕ is a path formula, then $E\phi$ and $A\phi$ are state formulas.
- (P1) Any state formula is a path formula.
- (P2) If ϕ and ψ are path formulas, then $\neg\phi$, $\phi \wedge \psi$, $X\phi$, $\diamond\phi$, and $\phi\ U\ \psi$ are path formulas.

Using the CTL_{apl} language, one can for example express that if an agent has a goal, then it will not drop the goal until it believes the goal is achieved, i.e., $G(\phi) \rightarrow A(G(\phi)\ U\ B(\phi))$.

6.3.2 CTL_{apl} Semantics

The semantics of the specification language CTL_{apl} is defined on a Kripke structure $M_{\mathcal{T}} = \langle C, R, V \rangle$, where the set of states C is the set of configurations of agents implemented in the agent programming language APL (definition 6.4), and the temporal relation R is specified by the transition system $\mathcal{T}$ of the agent programming language APL (definition 6.7). In particular, there exists a temporal relation between two configurations in the Kripke structure if and only if a transition between these two agent configurations is derivable from the transition system $\mathcal{T}$. Finally, the valuation function $V = (V_b, V_g)$ of the Kripke structure is defined on agent configurations and consists of different valuation functions each with respect to a specific mental attitude of agents' configurations. More specifically, we define a valuation function V_b that valuates the belief formulas in terms of agents' beliefs and a valuation function V_g that valuates the goal formulas in terms of the agents' goals. The belief valuation function V_b maps an agent configuration c to a set of propositions $V_b(c)$ that are derivable from the agent's belief base. An agent believes a proposition if and only if the proposition is included in $V_b(c)$. The valuation function V_g for goals is defined in such a way that all subgoals of an agent's goal are also considered as a goal. The valuation function V_g maps an agent's configuration to a set of sets of propositions. Each set contains all subgoals of a goal. An agent wants to achieve a proposition if and only if the proposition is included in a set. The semantics of the CTL_{apl} expressions are defined as follows.

Definition 6.10. Let $M_{\mathcal{T}} = \langle C, R, V \rangle$ be a Kripke structure specified by the execution of the transition system $\mathcal{T}$, where:

- C is a set of configurations (states) of the form $\langle \sigma, \gamma, \Pi \rangle$.

- $R \subseteq C \times C$ is a serial binary relation such that for each $(c, c') \in R$ we have $(c, c') \in \mathcal{R}_{\mathcal{T}}$ or $c = c'$.
- $V = (V_b, V_g)$ are the belief and goal evaluation functions, i.e.,
  - $V_b : C \rightarrow 2^{L_p}$ s.t. $V_b(\langle \sigma, \gamma, \Pi \rangle) = \{\phi \mid \sigma \models_p \phi\}$,
  - $V_g : C \rightarrow 2^{2^{L_p}}$ s.t. $V_g(\langle \sigma, \gamma, \Pi \rangle) = \{\{\phi' \mid \phi \models_p \phi'\} \mid \phi \in \gamma\}$.

A fullpath is an infinite sequence $x = c_0, c_1, c_2, \ldots$ of configurations such that $\forall i : (c_i, c_{i+1}) \in R$. We use x^i to indicate the i-th state of the path x.

- (S1) $M_{\mathcal{T}}, c \models B(\phi) \Leftrightarrow \phi \in V_b(c)$
- (S1) $M_{\mathcal{T}}, c \models G(\phi) \Leftrightarrow \exists s \in V_g(c) : \phi \in s$
- (S2) $M_{\mathcal{T}}, c \models \phi \wedge \psi \Leftrightarrow M_{\mathcal{T}}, c \models \phi$ and $M_{\mathcal{T}}, c \models \psi$
- (S2) $M_{\mathcal{T}}, c \models \neg\phi \Leftrightarrow M_{\mathcal{T}}, c \not\models \phi$
- (S3) $M_{\mathcal{T}}, c \models E\phi \Leftrightarrow \exists$ fullpath $x = c, c_1, c_2, \ldots \in M_{\mathcal{T}} : M_{\mathcal{T}}, x \models \phi$
- (S3) $M_{\mathcal{T}}, c \models A\phi \Leftrightarrow \forall$ fullpath $x = c, c_1, c_2, \ldots \in M_{\mathcal{T}} : M_{\mathcal{T}}, x \models \phi$
- (P1) $M_{\mathcal{T}}, x \models \phi \Leftrightarrow M_{\mathcal{T}}, x^0 \models \phi$ for ϕ is a state formula
- (P2) $M_{\mathcal{T}}, x \models X\phi \Leftrightarrow M_{\mathcal{T}}, x^1 \models \phi$
- (P2) $M_{\mathcal{T}}, x \models \diamond\phi \Leftrightarrow M_{\mathcal{T}}, x^n \models \phi$ for some $n \geq 0$
- (P2) $M_{\mathcal{T}}, x \models \phi \; U \; \psi \Leftrightarrow$
  a) $\exists k \geq 0$ such that $M_{\mathcal{T}}, x^k \models \psi$ and for all $0 \leq j < k : \; M_{\mathcal{T}}, x^j \models \phi$, or,
  b) $\forall j \geq 0 : \; M_{\mathcal{T}}, x^j \models \phi$

Note that the two options in the last clause capture two interpretations of the until operator. The first (strong) interpretation is captured by the option a and requires that the condition ψ of the until expression should hold at once. The second (weak) interpretation is captured by the option b and requires the formula ϕ can hold forever.

In the above definition, the CTL_{apl} state formulas are evaluated in the Kripke model $M_{\mathcal{T}}$ with respect to an arbitrary configuration c consisting of beliefs, goals, and plans. In the following, we model the execution of a particular agent program A (i.e., the execution of an agent with the initial configuration A[1]) based on the transition system $\mathcal{T}$ as the Kripke model $M^A_{\mathcal{T}} = \langle C^A_{\mathcal{T}}, R^A_{\mathcal{T}}, V \rangle$ on which the CTL_{apl} expressions (i.e., properties to be checked) can be evaluated. The accessibility relation $R^A_{\mathcal{T}}$ is defined as the set of executions (based on transition system $\mathcal{T}$) of the agent program A (i.e., traces that can be generated by applying planning rules and executing plans starting at the configuration specified by A) extended with a reflexive accessibility for all end configurations. This is to guarantee the seriality property of the accessibility relation $R^A_{\mathcal{T}}$. Moreover, the set $C^A_{\mathcal{T}}$ of configurations will be defined in terms of configurations that occur in the execution of the agent A.

[1] An agent's initial configuration is determined by the corresponding agent program which specifies the initial beliefs and goals. It is assumed that there are no initial plans.

Definition 6.11. (agent model) Let A be an agent program and let $\mathcal{E}_{\mathcal{T}}(A)$ be the execution of A. Then the model corresponding with agent program A, which we call an *agent model*, is defined as $M^A_{\mathcal{T}} = \langle C^A_{\mathcal{T}}, R^A_{\mathcal{T}}, V\rangle$, where the accessibility relation $R^A_{\mathcal{T}}$ and the set of configurations $C^A_{\mathcal{T}}$ are defined as follows:

$$R^A_{\mathcal{T}} = \mathcal{E}_{\mathcal{T}}(A) \cup \{(c_n, c_n) \mid \exists (c_{n-1}, c_n) \in \mathcal{E}_{\mathcal{T}}(A) \& \neg \exists (c_n, c_{n+1}) \in \mathcal{E}_{\mathcal{T}}(A)\}$$
$$C^A_{\mathcal{T}} = \{c \mid (c, c') \in R^A_{\mathcal{T}}\}$$

Note that agent models are Kripke structures in the sense of Definition 6.10.

As we are interested in expressing that a certain property holds for all executions of a particular agent program A, we will define the notion of satisfaction in an agent model.

Definition 6.12. (satisfaction in model) A formula ϕ is satisfied in the model $M^A_{\mathcal{T}} = \langle C^A_{\mathcal{T}}, R^A_{\mathcal{T}}, V\rangle$ if and only if ϕ holds in $M^A_{\mathcal{T}}$ with respect to all configurations $c \in C^A_{\mathcal{T}}$, i.e.,

$$M^A_{\mathcal{T}} \models \phi \quad \Leftrightarrow_{def} \quad \forall c \in C^A_{\mathcal{T}} \ : \ M^A_{\mathcal{T}}, c \models \phi$$

In section 6.4, we prove that certain properties hold for any agent program that is implemented in the *APL* programming language. As the above definition of model $M^A_{\mathcal{T}}$ is based on one specific agent program A, we need to quantify over all agent programs. Since the binary relation $R^A_{\mathcal{T}}$ (derived from the transition system $\mathcal{T}$, which is the semantics of the agent programs) has to be the same in all Kripke models, a quantification over agent programs means a quantification over models $M^A_{\mathcal{T}}$. This implies that we need to define the notion of validity of a property as being true for all agent programs and thus for all models $M^A_{\mathcal{T}}$.

Definition 6.13. (validity) A property $\phi \in CTL_{apl}$ holds for the execution of an arbitrary agent A based on the transition system $\mathcal{T}$, expressed as $\models_{\mathcal{T}}$, if and only if ϕ holds in all agent models $M^A_{\mathcal{T}}$, i.e.,

$$\models_{\mathcal{T}} \phi \quad \Leftrightarrow_{def} \quad \forall A \ : \ M^A_{\mathcal{T}} \models \phi$$

Note that this notion of validity is the same as the notion of validity in modal logic since it is defined at the level of frames, i.e., at the level of states and relation and not valuations, which is in our case defined in terms of specific agents.

Finally, we would like to explain our motivation for choosing a variant of *CTL* instead of other formalisms such as for example the linear time temporal logic *LTL*. Such a choice may not be trivial as one might argue that linear time temporal logic would be enough to specify and verify an agent's behavior. The idea would be to consider the execution behavior of the corresponding agent program as a set of linear traces. However, our consideration to use a variant of *CTL* is based on the fact that agents have choices (e.g., to select and execute plans from their plan library) and that these choices are essential characteristic of their autonomy. The computational

tree logic CTL with its branching time structure enables the specification and verification of such choices. In order to illustrate the characterising difference between CTL and LTL that is essential for capturing an agent's choices, consider the two execution models illustrated in Figure 6.1. While these two execution models include the same set of linear traces, the execution models A and B differ in the choices available to the agent. This is reflected by the fact that the CTL formula $AXE\Box p$ is true in state $s0$ of model B, while this is not the case for state $s0$ of model A. In other words, we have $A, s0 \not\models AXE\Box p$ and $B, s0 \models AXE\Box p$. In the next section, we will present an agent property which is related to the agent's choices. This property justifies the choice for using a variant of CTL for our specification and verification of agent programs.

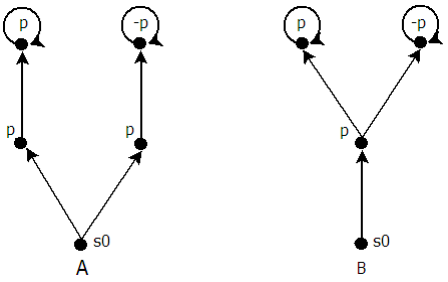

Fig. 6.1 Two execution models with two different choice moments.

6.4 Properties

Given the semantics of the programming language APL and the specification language CTL_{apl}, we can prove that certain properties expressed in CTL_{apl} hold for agents programmed in APL. Other properties for an extension of the APL language are provided in [130].

6.4.1 Proving the Properties

In this section, we present a number of desired properties and prove that they hold for arbitrary agent programs implemented in the APL language.

First, since the accessibility relation $R_A^{\mathcal{T}}$ of $M_{\mathcal{T}}^A$ is based on the transition system $\mathcal{T}$, we present some properties of the accessibility relation with respects to specific subsets of the transition system. In particular, the following proposition shows

persistence of unachieved goals through transitions that are derived based on transition rule R_1. Note that this transition rule modifies an agent's beliefs and remove achieved goals.

Proposition 6.1. *If $c_{i-1} \rightarrow c_i$ is a transition derived based on transition rule $R_1 \in \mathcal{T}$, $M^A_{\mathcal{T}}, c_{i-1} \models G(\phi)$, and $M^A_{\mathcal{T}}, c_i \not\models B(\phi)$, then $M^A_{\mathcal{T}}, c_i \models G(\phi)$.*

Proof. Following Definition 6.10 and using notation $sub(\psi) = \{\psi' \mid \psi \models_p \psi'\}$, we have $V_g(c_i) = \{sub(\psi) \mid \psi \in \gamma_i\}$ where γ_i is the goal based of configuration c_i. Following the definition of transition rule $R_1 \in \mathcal{T}$ for determining γ_i, we have $\{sub(\psi) \mid \psi \in \gamma_i\} = \{sub(\psi) \mid \psi \in \gamma_{i-1} \setminus \{\psi' \mid \sigma_i \models_p \psi'\}\}$, where σ_i is the belief base of configuration c_i. Using Definition 6.10 again, we have $\{sub(\psi) \mid \psi \in \gamma_{i-1} \setminus \{\psi' \mid \sigma_i \models_p \psi'\}\} = \{sub(\psi) \mid \psi \in \gamma_{i-1} \setminus V_b(c_i)\} = \{sub(\psi) \mid \psi \in \gamma_{i-1}\} \setminus \{sub(\psi) \mid \psi \in V_b(c_i)\} = V_g(c_{i-1}) \setminus \{sub(\psi) \mid \psi \in V_b(c_i)\}$. Suppose now that $sub(\phi) \in V_g(c_{i-1})$ and $\phi \notin V_b(c_i)$. Then, from the above equations we have $sub(\phi) \in V_g(c_i)$, which proves the proposition.

We can now generalize this proposition by showing the persistence of unachieved goals through all transitions.

Proposition 6.2. *If $c_{i-1} \rightarrow c_i$ is a transition, $M^A_{\mathcal{T}}, c_{i-1} \models G(\phi)$, and $M^A_{\mathcal{T}}, c_i \not\models B(\phi)$, then $M^A_{\mathcal{T}}, c_i \models G(\phi)$.*

Proof. This is the direct consequence of the following facts: 1) an agent's goals persist through transitions that are derived based on transition rules R_2 and R_3 as these transition rules do not modify the agent's goals, 2) an agent's goals persists through reflexive transitions, and 3) unachieved goals of an agent persist through transitions derived based on transition R_1 (Proposition 6.1).

The next property satisfied by the programs implemented in *APL* language is a variant of what has been termed "blind commitment" in [360], and what are called "persistent goals" in [110]. This property expresses that the execution of an *APL* agent program should not drop a goal before it is believed to be achieved.

Proposition 6.3. *(blind commitment)* $\models_{\mathcal{T}} G(\phi) \rightarrow A(G(\phi)\ U\ B(\phi))$

Proof. Using Definitions 6.13 and 6.12, we have to to prove that for any agent models $M^A_{\mathcal{T}}$ and all its configurations c:
if $M^A_{\mathcal{T}}, c \models G(\phi)$ then $M^A_{\mathcal{T}}, c \models A(G(\phi)\ U\ B(\phi))$.
Using Definition 6.10, we have to prove:
if $M^A_{\mathcal{T}}, c \models G(\phi)$ then $\forall$ fullpath $x = c, c', c'', \ldots \in M^A_{\mathcal{T}} : M^A_{\mathcal{T}}, x \models G(\phi)\ U\ B(\phi)$.
We prove that for arbitrary $M^A_{\mathcal{T}}$ and configuration c_0, it holds:
if $M^A_{\mathcal{T}}, c_0 \models G(\phi)$ then $\forall$ fullpath $x = c_0, c', c'', \ldots \in M^A_{\mathcal{T}} : M^A_{\mathcal{T}}, x \models G(\phi)\ U\ B(\phi)$.
Assume $M^A_{\mathcal{T}}, c_0 \models G(\phi)$ and take an arbitrary path $c_0, c_1, c_2, \ldots$ starting with c_0. We have to prove that $M^A_{\mathcal{T}}, c_0, c_1, c_2, \ldots \models G(\phi)\ U\ B(\phi)$. Following definition 6.10, we

have to prove the following:
a) $\exists k \geq 0$ such that $M, c_k \models B(\phi)$ and for all $0 \leq j < k : \ M, c_j \models G(\phi)$, or
b) $\forall j \geq 0 : \ M, c_j \models G(\phi)$
For the path $c_0, c_1, c_2, \ldots$, we distinguish two cases. Either for all consecutive states c_{i-1} and c_i in the path it holds that the transition $c_{i-1} \rightarrow c_i$ is such that $M^A_{\mathcal{T}}, c_i \not\models B(\phi)$, or there exists consecutive states c_{k-1} and c_k such that transition $c_{k-1} \rightarrow c_k$ is the first with $M^A_{\mathcal{T}}, c_k \models B(\phi)$. The first case (formulation of clause b above) is proven by induction as follows:
(Basic case) Given $M^A_{\mathcal{T}}, c_0 \models G(\phi)$ (assumption), $M^A_{\mathcal{T}}, c_1 \not\models B(\phi)$, Proposition 6.2 guarantees that $M^A_{\mathcal{T}}, c_1 \models G(\phi)$.
(Inductive case) Let $M^A_{\mathcal{T}}, c_{i-1} \models G(\phi)$. Using Proposition 6.2 together with the fact $M^A_{\mathcal{T}}, c_i \not\models B(\phi)$, we have $M^A_{\mathcal{T}}, c_i \models G(\phi)$.
The second case (formulation of clause a above) is proven as follows. As $c_{k-1} \rightarrow c_{k-1}$ is the first transition such that $M^A_{\mathcal{T}}, c_k \models B(\phi)$, we have $\forall 0 < i < k \ : \ M^A_{\mathcal{T}}, c_i \not\models B(\phi)$. Since $M^A_{\mathcal{T}}, c_0 \models G(\phi)$, we can apply Proposition 6.2 to all transitions $c_0 \rightarrow c_1 \ , \ldots, \ c_{k-2} \rightarrow c_{k-1}$ to show that $M^A_{\mathcal{T}}, c_{k-1} \models G(\phi)$. This is exactly the formulation of clause a above.

It should be noted that blind commitment in [360] is defined for intentions, rather than goals. Goals are also present in their framework, but are contrasted with intentions in that the agent is not necessarily committed to achieving its goals (but is committed in some way to achieving its intentions).

We now proceed to give a definition of intention, and show how intentions defined in this way are related to an agent's goals. We define that an agent intends κ, if κ follows from the second component of one of the plans in an agent's plan base. The second component of a plan specifies the subgoal for which the plan was selected, and it is these subgoals for which the agent is executing the plans, that we define to form the agent's intentions. This is analogous to the way the semantics of intention is defined in [77].

Definition 6.14. (intention) Let $V_i : C \rightarrow 2^{2^{L_p}}$ be defined as $V_i(\langle \sigma, \gamma, \Pi \rangle) = \{\{\kappa' \mid \kappa \models_p \kappa'\} \mid (\phi, \kappa, \pi) \in \Pi\}$. Then $M, c \models I(\kappa)$ is defined as $\exists s \in V_i(c) : \kappa \in s$.

Given this definition, we prove that an agent's intentions are a "subset" of the agent's goals, i.e., we prove the following proposition.

Proposition 6.4. *(intentions)*

$$\models_{\mathcal{T}} I(\kappa) \rightarrow G(\kappa)$$

Proof. The proof is based on induction by showing that for arbitrary agent model $M^A_{\mathcal{T}}$ and initial state c_0 the proposition holds for the initial state (basic case), and if it holds for a state of the model, then it holds for the next state of the model as well.

(Basic case) $M^A_{\mathcal{T}}, c_0 \models I(\kappa) \rightarrow G(\kappa)$ for the initial state c_0. Since agents are assumed to have no plans initially, we have $\nexists s \in V_i(c_0) : k \in s$. Using the definition of $I(\kappa)$ we conclude that $M^A_{\mathcal{T}}, c_0 \not\models I(\kappa)$ and thus $M^A_{\mathcal{T}}, c_0 \models I(\kappa) \rightarrow G(\kappa)$.

(Inductive case) Suppose $M^A_{\mathcal{T}}, c \models I(\kappa) \rightarrow G(\kappa)$ and there is a transition $c \rightarrow c'$. We show that $M^A_{\mathcal{T}}, c' \models I(\kappa) \rightarrow G(\kappa)$. We distinguish two cases for the assumption $M^A_{\mathcal{T}}, c \models I(\kappa) \rightarrow G(\kappa)$: 1) $M^A_{\mathcal{T}}, c \models I(\kappa)$, and 2) $M^A_{\mathcal{T}}, c \not\models I(\kappa)$.
(Case 1) Suppose $M^A_{\mathcal{T}}, c \models I(\kappa) \rightarrow G(\kappa)$, $M^A_{\mathcal{T}}, c \models I(\kappa)$, and thus $M^A_{\mathcal{T}}, c \models G(\kappa)$. We show $M^A_{\mathcal{T}}, c' \models \neg G(\kappa) \rightarrow \neg I(\kappa)$ (contraposition), i.e., we show if $M^A_{\mathcal{T}}, c' \not\models G(\kappa)$ then $M^A_{\mathcal{T}}, c' \not\models I(\kappa)$. By Definition 6.10, $M^A_{\mathcal{T}}, c \models G(\kappa) \Leftrightarrow \exists s \in V_g(c) : k \in s$. Given $M^A_{\mathcal{T}}, c \models G(\kappa)$ and $M^A_{\mathcal{T}}, c' \not\models G(\kappa) \Leftrightarrow \nexists s \in V_g(c') : \kappa \in s$, we conclude that the transition $c \rightarrow c'$ is derived based on transition rule R_1 (as this is the only transition rule that modifies the agent's goals) and therefore $\forall \psi, \pi : (\psi, \kappa, \pi) \notin \Pi_{c'}$. Note that if $\exists s \in V_g(c') : \psi \in s$ would be the case, then we should also have $k \in s$ (because $\psi \models \kappa$), which contradict the assumption $\nexists s \in V_g(c') : \kappa \in s$. Since $\forall \psi, \pi : (\psi, \kappa, \pi) \notin \Pi_{c'}$ we conclude $\nexists s \in V_i(c') : \kappa \in s$ and thus $M^A_{\mathcal{T}}, c' \not\models I(\kappa)$.
(case 2) Suppose $M^A_{\mathcal{T}}, c \models I(\kappa) \rightarrow G(\kappa)$, $M^A_{\mathcal{T}}, c \not\models I(\kappa)$, and thus $M^A_{\mathcal{T}}, c \not\models G(\kappa)$. We show then $M^A_{\mathcal{T}}, c' \models I(\kappa) \rightarrow G(\kappa)$, i.e., if $M^A_{\mathcal{T}}, c' \models I(\kappa)$, then $M^A_{\mathcal{T}}, c' \models G(\kappa)$. Assume $M^A_{\mathcal{T}}, c' \models I(\kappa)$. Given $M^A_{\mathcal{T}}, c \not\models I(\kappa)$ we can conclude that the transition $c \rightarrow c'$ is derived based on transition rule R_3 by applying a planning rule. $M^A_{\mathcal{T}}, c' \models I(\kappa) \Leftrightarrow \exists s \in V_i(c') : \kappa \in s \Leftrightarrow \exists (\phi, \kappa', \pi) \in \Pi_{c'} : \kappa' \models \kappa$ and $\phi \models \kappa'$. The fact that a planning rule is applied means that $M^A_{\mathcal{T}}, s \models G(\phi)$ and therefore $M^A_{\mathcal{T}}, s \models G(\kappa)$.

Intuitively, this property holds since "intentions" or plans are generated on the basis of goals such that a plan cannot be created without a corresponding goal. Moreover, if a goal is removed, its corresponding plans are also removed. Note that while the commitment strategies were defined for intentions in [360] and hold for goals in our framework, the property of the BDI logic that relates goals and intentions *does* map directly to goals and (what we have defined as) intentions in our framework. Note also that the opposite of Proposition 6.4 does not hold, as it can be the case that an agent has a goal for which it has not yet selected a plan.

Finally, we present a property related to the choices of an agent, implemented by an *APL* program. This property shows our motivation for choosing a variant of *CTL* as the specification language. The property states that an agent can choose to commit to one of its goals and generate an intention, if the agent has appropriate means.

Proposition 6.5. *(intention = choice + commitment) Assume an agent program A with a planning rule $\beta, \kappa \Rightarrow \pi$. Let $\mathcal{T}$ be the transition system generated by the transition rules $R_1, \ldots, R_3$ based on this agent program.*

$$\models_{\mathcal{T}} (B(\beta) \wedge G(\kappa) \wedge \neg I(\kappa)) \rightarrow EX\, I(\kappa)$$

Proof. We prove that for arbitrary $M^A_{\mathcal{T}}$ and configuration c_0, it holds: if $M^A_{\mathcal{T}}, c_0 \models B(\beta) \wedge G(\kappa) \wedge \neg I(\kappa)$, then $M^A_{\mathcal{T}}, c_0 \models EX\ I(\kappa)$. Following definition 6.10, we have to prove that if $M^A_{\mathcal{T}}, c_0 \models B(\beta) \wedge G(\kappa) \wedge \neg I(\kappa)$, then $\exists$ fullpath $x = c_0, c_1, c_2, \ldots \in M^A_{\mathcal{T}} : M^A_{\mathcal{T}}, x \models X\ I(\kappa)$. Assume $M^A_{\mathcal{T}}, c_0 \models B(\beta) \wedge G(\kappa) \wedge \neg I(\kappa)$. Then, following definitions 6.10 and 6.14, we have $\beta \in V_b(c_0)$, $\exists s \in V_g(c_0) : k \in s$, and $\forall s \in V_i(c_0) : \kappa \notin s$. Note that definition 6.14 ensures that $\forall \phi\ \nexists \pi' \in \mathsf{Plan} : (\phi, \kappa, \pi') \in \Pi_{c_0}$ (where Π_{c_0} is the plan base that corresponds to configuration c_0). This means that the transition rule R_3 is applicable in c_0, which in turn means that a transition $c_0 \rightarrow c_1$ is derivable in the transition system $\mathcal{T}$ such that $(\phi, \kappa, \pi) \in \Pi_{c_1}$ for some $\phi \in V_g(c_1)$. By definition 6.14, we conclude that $M, c_1 \models I(\kappa)$. This ensures the existing of a path $x = c_0, c_1, \ldots : M, x \models X\ I(\kappa)$.

6.5 Debugging Multi-Agent Programs

In previous sections, we showed how a BDI-based agent-oriented programming language can be related to a BDI specification language. The relation allows us to prove that certain *generic* properties expressed in the specification language hold for the agent programming language, and thus for all executions of all agent programs that are implemented using this agent programming language.

However, one may want to verify properties for a specific execution of a specific multi-agent program. Of course, model-checking and theorem proving are two verification approaches that can be used to check properties of specific programs. The problem with these verification approaches is that they are often less effective for complex and real application programs. In order to check properties of such complex programs, one may consider a debugging approach and check a specific execution of a specific program. Thus, in contrast to model checking and theorem proving that analyze all possible full execution traces of a program at once, the debugging approach analyzes one specific execution trace of a specific program. It is important to emphasize that model-checking and theorem proving can therefore be used to prove the correctness of programs, while debugging can only be used to find possible defects of programs (as displayed in particular runs).

In the following sections, we propose a debugging approach that can be used to check temporal and cognitive properties of specific BDI-based multi-agent programs, e.g., if two or more implemented agents[2] can have the same beliefs, whether the number of agents is suited for the environment (e.g. it is useless to have a dozen explorers on a small area, or many explorers when there is only one cleaner that cannot keep up with them.), whether the protocol is suited for the given task (e.g. there might be a lot of overhead because facts are not shared, and therefore, needlessly rediscovered), whether important beliefs are shared and adopted, or rejected, once

[2] In the following, we write 'agents' and 'implemented agents' interchangeably since we focus on programs that implement agents.

they are received. We may also want to check if unreliable sources of information are ignored, or whether the actions of one agent are rational to take based on the knowledge of other agents.

6.5.1 Debugging Modes

Ideally one would specify a *cognitive* and *temporal* property and use it in two different debugging modes. In one debugging mode, called *continuous mode*, one may want to execute a multi-agent program and get notified when the specified property evaluates to true *during its execution*. In the second debugging mode, called *post mortem*, one may want to execute a multi-agent program, stop it after some execution steps, and check if the specified property evaluates to true for the performed execution. For both debugging modes, the specified properties are evaluated in the *initial state* of the multi-agent program *execution trace generated thusfar*. For the post mortem debugging mode, the generated execution trace thusfar is the trace generated from the start of the program execution until the execution is stopped. However, for the continuous debugging mode, the specified properties are evaluated after each execution step and with respect to the execution trace generated thusfar, i.e., the execution trace generated from the start of the program execution until the last execution step. This is because during a program execution a trace is modified and extended after each execution step. It should be noted that subsequent execution steps generate new program states and therefore new traces.

In the continuous debugging mode, the evaluation of a specified property *during* the execution of a multi-agent program means a continuous evaluation of the property on its evolving execution trace as it develops by consecutive execution steps. This continuous evaluation of the property can be used to halt the program execution as soon as a trace is generated which satisfies the property. It is important to know that properties are evaluated in the initial state of the execution trace so that the trace properties should be specified as temporal properties. A developer of multi-agent programs is assumed to know these aspects of our debugging framework in order to debug such programs effectively. Similar ideas are proposed in Jadex [343].

In the following, we introduce a specification language, called MDL (multi-agent description language), to specify the *cognitive* and *temporal* behavior (i.e., execution traces) of the BDI-based multi-agent programs. The MDL description language is taken to be a variant of LTL (Linear Temporal Logic) because execution traces of multi-agent programs, which are used to debug[3] such programs, are assumed to be linear traces. Note that this assumption is realistic as the interpreter of most (multi-agent) programs performs one execution step at a time and thereby generates a linear trace. An MDL expression is evaluated on the (finite) execution trace of a

[3] In contrast to debugging that analyzes one linear execution trace of a program, other verification techniques such as model checking and theorem proving analyze all possible execution traces of a program at once.

multi-agent program and can activate a debugging tool when it is evaluated to true. The debugging tools are inspired by traditional debugging tools, extended with the functionality to verify a multi-agent program execution trace. One example of such a debugging tool is a multi-agent version of the breakpoint. The breakpoint can halt the execution of a single agent program, a group of agent programs or the complete multi-agent program. This multi-agent version of the breakpoint can also have an MDL expression as a condition, making it a conditional breakpoint.

6.5.2 Specification Language for Debugging: Syntax

In this section, we present the syntax of the MDL written in EBNF notation. An expression of this language describes a property of an execution of a multi-agent program in *APL* and can be used to perform/activate debugging actions/tools. In the following, ⟨*group_id*⟩ is a group identifier (uncapitalized string), ⟨*agent_id*⟩ an agent identifier (uncapitalized string), ⟨*query_name*⟩ a property description name (a reference to a specified property used in the definition of macros; see later on for a discussion on macros), ⟨*Var*⟩ a variable (Variables are capitalized strings), [*all*] indicates the group of all agents, and ⟨*agent_var*⟩ an agent identifier, a group identifier, or a variable. Finally, we use *Bquery*, *Gquery*, and *Pquery* to denote an agent's Beliefs, Goals, and Plans, respectively.

⟨*group_def*⟩ ::= "[" ⟨*group_id*⟩ "]" " = " ⟨*agent_list*⟩
⟨*agent_list*⟩ ::= "[" ⟨*agent_id*⟩ ("," ⟨*agent_id*⟩) ∗ "]"
⟨*mdl_pd*⟩ ::= ⟨*query_name*⟩ "{" ⟨*mdl_query*⟩ "}"
⟨*mdl_query*⟩ ::= "{" ⟨*mdl_query*⟩ "}"
| ⟨*agent_var*⟩ "@Beliefs(" ⟨*Bquery*⟩ ")"
| ⟨*agent_var*⟩ "@Goals(" ⟨*Gquery*⟩ ")"
| ⟨*agent_var*⟩ "@Plans(" ⟨*Pquery*⟩ ")"
| ⟨*UnOp*⟩ ⟨*mdl_query*⟩
| ⟨*mdl_query*⟩ ⟨*BinOp*⟩ ⟨*mdl_query*⟩
| "?" ⟨*query_name*⟩
⟨*BinOp*⟩ ::= "and" | "or" | "implies" | "until"
⟨*UnOp*⟩ ::= "not" | "next" | "eventually" | "always"
⟨*agent_var*⟩ ::= ⟨*Var*⟩ | ⟨*agent_id*⟩ | ⟨*group_id*⟩ | "[all]"

Note that ⟨*mdl_pd*⟩ is a specified property that describes the (temporal and cognitive) behavior of a multi-agent program execution.

In order to specify that either `all` agents believe that there is a bomb at position `2,3` (i.e., `bomb(2,3)`) or `all` agents believe that there is no bomb at that position (i.e. `not bomb(2,3)`), we can use the following MDL expression.

```
[all]@Beliefs(bomb(2,3))  or
[all]@Beliefs(not bomb(2,3))
```

Since specified properties in our framework are always evaluated in the initial state of the program execution trace (and thus specified by the multi-agent program), the above property will evaluate to true if it holds in the initial state. Therefore, if this property is evaluated to true in a program execution trace, then it will evaluate to true for the rest of the program execution. Note that if this property should hold in all states of the program execution, then it should be put in the scope of the 'always' operator. Moreover, if the property should hold in the last state of the program execution, then it should be put in the scope of the 'eventually' operator.

We can generalize the above property by assigning a name to it and parameterizing the specific beliefs (in this case `bomb(X,Y)`). This generalization allows us to specify a property as a macro that can be used to define more complex properties. For example, consider the following generalization (macro) that holds in a state of a multi-agent program if and only if either all agents believe the given belief ϕ or all agents do not believe ϕ.

```
isSharedBelief(φ){
    [all]@Beliefs(φ) or
    [all]@Beliefs(not φ)
}
```

Note that `isSharedBelief(`ϕ`)` can now be used (e.g., in other property specifications) to check whether or not ϕ is a shared belief. In general, one can use the following abstract scheme to name an MDL expression. Parameters `Var1, Var2,` and `Var3` are assumed to be used in the MDL expression.

```
name( Var1, Var2, Var3, ...) { MDL expression }
```

The following example demonstrates the use of macros. To use an MDL expression inside another one, the macro's names should be preceded by a "?" mark. We now define a cell as detected when agents agree on the content of that cell. We define `detectedArea(R)` as follows.

```
detectedArea(X, Y) { ?isSharedBelief( bomb(X,Y) ) }
```

The next example shows an MDL expression that can be used to verify whether the `gridworld` will eventually be clean if an agent has the goal to clean it. In particular, the expression states that if an agent `A` has the goal to clean the `gridworld` then eventually that agent `A` will believe that the `gridworld` is clean.

```
cleanEnvironment(A) {
    A@Goals(clean(gridworld))
    implies
    eventually A@Beliefs(clean(gridworld))
}
```

It is important to note that if this property evaluates to false for an execution thusfar, it may *not* continue to be false for the rest of the execution (cf. Definition 6.15). This is due to the evaluation of the `eventually` operator in the context of finite traces. In particular, if the above property evaluates to false for a finite program execution trace, then it may not evaluate to false for a suffix of that trace. One particular use of the eventually operator is therefore to check and stop the execution of a multi-agent program when it reaches a state with a specific property.

The following MDL expression states that an agent `A` will not unintentionally drop the bomb that it carries. More specifically, the expression states that if an agent believes to carry a bomb, then the agent will believe to carry the bomb until it has a plan to drop the bomb. It is implicitly assumed that all plans will be successfully executed.

```
doesNotLoseBomb(A) {
  always (  A@Beliefs(carry(bomb))
            implies
            (  A@Beliefs(carry(bomb))
               until
               A@Plans(dropped_bomb)
            )
         )
}
```

6.5.3 Specification Language for Debugging: Semantics

The semantics of the MDL language describes how a property is evaluated against a trace of a BDI-based multi-agent program. In the context of debugging, we consider *finite traces* generated by partial execution of multi-agent programs (a partial execution of a program starts in the initial state of the program and stops after a finite number of deliberation steps). A finite trace is a finite sequence of multi-agent program states in which the state of each agent is a tuple consisting of beliefs, goals, and plans.

An MDL expression is evaluated with respect to a finite multi-agent program trace that results from a partial execution of a multi-agent program. In the following, we use t to denote a finite trace, $|t|$ to indicate the length of the trace t (a natural number; a trace consists of 1 or more states), st to indicate a trace starting with state

s followed by the trace t, $|st| = 1 + |t|$, and functions *head* and *tail*, defined as follows: $head(st) = s$, $head(t) = t$ if $|t| = 1$, $head(t,i) = A_i$ if $head(t) = \{A_1, \ldots, A_i, \ldots, A_n\}$, $tail(st) = t$ and $tail(t)$ is undefined if $|t| \leq 1$ (*tail* is a partial function). Moreover, given a finite trace $t = s_1 s_2 \ldots s_n$, we write t_i to indicate the suffix trace $s_i \ldots s_n$.

Definition 6.15. Let $s_i = \{A_1, \ldots, A_n\}$ be a multi-agent program configuration and let $t = s_1 s_2 \ldots s_n$ be a finite trace of a multi-agent program such that $|t| \geq 1$. Let also the evaluation functions V_b, V_g, and V_i be as defined in Definitions 6.10 and 6.14, respectively. The satisfaction of MDL expressions by the trace t is defined as follows:

$t \models \texttt{i@Beliefs}(\phi) \Leftrightarrow \phi \in V_b(head(t,i))$

$t \models \texttt{i@Goals}(\phi) \Leftrightarrow \exists s \in V_g(head(t,i)) : \phi \in s$

$t \models \texttt{i@Plans}(\phi) \Leftrightarrow \exists s \in V_i(head(t,i)) : \phi \in s$

$t \models \phi \texttt{ and } \psi \Leftrightarrow t \models \phi \ \textit{and} \ t \models \psi$

$t \models \phi \texttt{ or } \psi \Leftrightarrow t \models \phi \ \textit{or} \ t \models \psi$

$t \models \phi \texttt{ implies } \psi \Leftrightarrow t \models \phi \ \textit{implies} \ t \models \psi$

$t \models \texttt{not } \phi \Leftrightarrow t \not\models \phi$

$t \models \texttt{next } \phi \Leftrightarrow tail(t) \models \phi \ \textit{and} \ |t| > 1$

$t \models \texttt{eventually } \phi \Leftrightarrow \exists i \leq |t| \ \ (t_i \models \phi)$

$t \models \texttt{always } \phi \Leftrightarrow \forall i \leq |t| \ \ (t_i \models \phi)$

$t \models \phi \texttt{ until } \psi \Leftrightarrow \exists i \leq |t| \ \ (t_i \models \psi \ \textit{and} \ \forall j < i \ \ (t_j \models \phi))$

Based on this definition of MDL expressions, we have implemented some debugging tools that are activated and updated when their corresponding MDL expression holds in a partial execution of a multi-agent program. These debugging tools are described in the next section. This definition of the satisfaction relation can behave different than the standards definition of satisfaction relation of LTL which is defined on infinite traces. For example, some LTL properties such as $\neg\texttt{next}\phi = \texttt{next}\neg\phi$ are valid only for infinite traces. However, the validity of such properties is not relevant for our debugging framework as debugging is only concerned with the execution thusfar and therefore with finite traces. We would like to emphasize that different LTL semantics for finite traces of program executions have been proposed. See [28] for a comparison between different proposals.

6.6 Multi-Agent Debugging Tools

A well-known technique often used for debugging single sequential and concurrent programs is a *breakpoint*. A breakpoint is a marker that can be placed in the program's code. Breakpoints can be used to control the program's execution. When

the marker is reached program execution is halted. Breakpoints can be either conditional or unconditional. Unconditional breakpoints halt the program execution when the breakpoint marker is reached. Conditional breakpoints only halt the program execution when the marker is reached and some extra condition is fulfilled. Another (similar) functionality, that can be used to re-synchronize program executions, is called a process barrier breakpoint. Process barrier breakpoints are much like normal breakpoints. The difference is they halt the processes that reached the barrier point until the last process reaches the barrier point. A different debugging technique used for traditional programming practices is called the *watch*. The watch is a window used to monitor variables' values. Most watch windows also allow the developer to type in a variable name and if the variable exists the watch will show the variable's value. In the IDEs of most high-level programming languages the watch is only available when the program's execution is halted. Other traditional debugging techniques are logging and visualization. Logging allows a developer to write some particular variable's value or some statement to a logging window or a file. Visualization is particularly helpful in the analysis and fine tuning of concurrent systems. Most relevant in light of our research is the ability to visualize the message queue.

Despite numerous proposals for BDI-based multi-agent programming languages, there has been little attention on building effective debugging tools for *BDI-based* agent-oriented programs. The existing debugging tools for BDI-based programs enable the observation of program execution traces (the sequence of program states generated by the program's execution) [63,114,121,122,343] and browsing through these execution traces, allowing to run multi-agent programs in different execution modes by for example using breakpoints and assertions [63, 114, 122, 343], observing the message exchange between agents and checking the conformance of agents' interactions with a specific communication protocol [84,114,343,345,346,425]. Although most proposals are claimed to be applicable to other BDI-based multi-agent programming languages, they are presented for a specific multi-agent platform and the corresponding multi-agent programming language. In these proposals, debugging multi-agent aspects of such programs are mainly concerned with the interaction between individual agents and the exchanged messages. Finally, the temporal aspects of multi-agent program execution traces are only considered in a limited way and not fully exploited for debugging purposes.

This section presents a set of Multi-Agent Debugging Tools (MADTs) to illustrate how the MDL language can be used to debug multi-agent programs. In order to use the debugging tools, markers are placed in the multi-agent programs to denote under which conditions which debugging tool should be activated. A marker consists of an (optional) MDL expression and a debugging tool. The MDL expression of a marker specifies the condition under which the debugging tool of the marker should be activated. In particular, if the MDL expression of a marker evaluates to true for a given finite trace/partial execution of a multi-agent program, then the debugging tool of the marker will be activated. When the MDL expression of a marker is not given (i.e., not specified), then the associated debugging tool will be activated as soon as the multi-agent program is executed. Besides an MDL expression, a marker can also have a group parameter. This group parameter specifies which

agents the debugging tool operates on. The general syntax of a marker is defined as follows:

⟨*marker*⟩ ::= "`MADT`("⟨*madt*⟩[","⟨*mdl_query*⟩][",@"⟨*group*⟩]")"
⟨*group*⟩ ::= "["⟨*group_id*⟩"]"|⟨*agent_list*⟩

The markers that are included in a multi-agent program are assumed to be processed by the interpreter of the corresponding multi-agent programming language. In particular, the execution of a multi-agent program by the interpreter will generate consecutive states of a multi-agent program and, thereby, generating a trace. At each step of the trace generation (i.e., at each step where a new state is generated) the interpreter evaluates the MDL expression of the specified markers in the initial state of the finite trace (according to the definition of the satisfaction relation; see definition 6.15) and activates the corresponding debugging tools if the MDL expressions are evaluated to true. This means that the trace of a multi-agent program is verified after every change in the trace. This mode of processing markers corresponds to the *continuous debugging mode* and does not stop the execution of the multi-agent program; markers are processed *during* the execution of the program. In the *post mortem debugging mode*, where a multi-agent program is executed and stopped after some deliberation steps, the markers are processed based on the finite trace generated by the partial execution of the program. It is important to note again that MDL expressions are always evaluated in the initial state of traces as we aim at debugging the (temporal) behavior of multi-agent programs and thus their execution traces from the initial state. The following example illustrates the use of a marker in a multi-agent program:

```
MADT( breakpoint_madt ,
      eventually cleaner@Beliefs(bomb(X,Y))
    )
```

This marker, which can be placed in the multi-agent program, activates a breakpoint as soon as the cleaner agent believes that there is a bomb in a cell of the `gridworld`. It is important to note that if no MDL expression is given in a specified marker, then the associated debugging tool will be activated after each update of the trace. Removing the specified MDL expression from the abovementioned marker means that the execution of the multi-agent program will be stopped after each trace update. This results in a kind of stepping execution mode. Furthermore, if no group parameter is given in the marker, the "[all]" group is used by default.

In the rest of this section, we illustrate the use of a set of debugging tools that have shown to be effective in debugging software systems. Examples of debugging tools are breakpoint, logging, state overview, or message list. The behavior of these debugging tools in the context of markers are explained in the rest of this section. The proposed set of debugging tools is by no means exhaustive and can be extended with other debugging tools. We thus do neither propose new debugging tools nor

evaluate their effectiveness. The focus of this chapter is a framework for using (existing) debugging tools to check cognitive and temporal behavior of multi-agent program. Our approach is generic in the sense that a debugging tool can be associated with an MDL expression by means of a marker and that markers can be used in two debugging modes.

6.6.1 Breakpoint

The breakpoints for multi-agent programs are similar to breakpoints used in concurrent programs. They can be used to pause the execution of a single agent program, a specific group of agent programs, or the execution of the entire multi-agent program. Once the execution of a program is paused, a developer can inspect and browse through the program execution trace generated so far (including the program state in which the program execution is paused). The developer can then continue the program execution in a stepping mode to generate consecutive program states. An attempt to further execute the program continuously (not in stepping mode) pauses immediately since the MDL expression associated to the breakpoint will be evaluated in the initial state of an extension of the same trace. In general, if an MDL expression evaluates to true in a state of a trace, then it will evaluate to true in the same state of any extension of that trace.

The example below demonstrates the use of a conditional breakpoint on the agents `explorer1` and `explorer2`. The developer wants to pause both agents as soon as agent `cleaner` has the plan to go to cell `(5, 5)`.

```
MADT( breakpoint_madt,
      eventually cleaner@Plans(goto(5, 5)) ,
      @[explorer1, explorer2]
    )
```

Note that it is possible to use the cognitive state of more than one agent as the break condition. The next example demonstrates how a developer can get an indication about whether the number of explorer and cleaner agents are suitable for a certain scenario. In fact, if there are not enough cleaners to remove bombs, or when all explorers are located at the same area, then all explorers will find the same bomb.

```
MADT( breakpoint_madt,
      eventually [explorers]@Beliefs(bomb(X,Y))
    )
```

The breakpoint tool is set to pause the execution of all agents, once all agents that are part of the "explorers" group have the belief that a bomb is located at the same cell `(X,Y)`. Note that it need not be explicitly defined to pause the execution of *all*

agents. The breakpoint is useful in conjunction with the watch tool to investigate the mental state of the agent. Other agent debugging approaches, e.g., [114], propose a similar concept for breakpoints, but for a single BDI-based agent program. Also, Jason [63] allows annotations in plan labels to associate extra information to a plan. One standard plan annotation is called a breakpoint. If the debug mode is used and the agent executes a plan that has a breakpoint annotation, execution pauses and the control is given to the developer, who can then use the step and run buttons to carry on the execution. Note that in contrast with other approaches, the condition in our approach may contain logic and temporal aspects.

6.6.2 Watch

The watch can display the current mental state of one or more agents. Furthermore, the watch allows the developer to query any of the agents' bases. The developer can, for example, use the watch to check if a belief follows from the belief base. It is also possible to use an MDL expression in the watch; if the expression evaluates to *true*, the watch will show the substitution found. The watch tool can also be used to visualize which agents have shared or conflicting beliefs. The watch tool is regularly used in conjunction with a conditional breakpoint. Once the breakpoint is hit, the watch tool can be used to observe the mental state of one or more agents. In general, the watch tool should be updated unconditionally and for all agents in the system. Adding `MADT(watch_madt)` to a multi-agent program will activate the watch on every update of its execution trace. In Jason and Goal [63, 224], a similar tool is introduced which is called the mind inspector. This mind inspector, however, can only be used to observe the mental state of individual agents. Jadex [343] offers a similar tool called the BDI-inspector which allows visualization and modification of internal BDI-concepts of individual agents.

6.6.3 Logging

Logging is done by the usage of probes which, unlike breakpoints, do not halt the multi-agent program execution. When a probe is activated it writes the current state of a multi-agent program, or a part of it, to a log screen or a file (depending on the type of probe). Using a probe without an MDL expression and without a group specification is very common and can be done by adding `MADT(probe_madt)` in multi-agent programs. The probe will be activated on every update of the program trace such that it keeps a log of all multi-agent program states. The next example saves the state of the multi-agent program when the cleaner agent drops a bomb in a depot, but there is still some agent who believes the bomb is still at its original place.

```
MADT( probe_madt,
      eventually( cleaner@Plans(dropBomb(X,Y))
                  and
                  A@Beliefs(bomb(X,Y))
                )
)
```

This means that the `probe_madt` will be activated directly after an execution step that generates a trace on which the MDL expression evaluates to true. A developer can thus use such expressions (of the form `eventually`ϕ) in order to be notified at once and as soon as the program execution satisfies it. Once this expression evaluates to true, the developer should know that any continuation of the program execution will evaluates it to true. Thus, from a developer's perspective, properties specified by expressions of the form `eventually`ϕ can be used to get notified (or stop the execution) only once and as soon as it is satisfied. Similar work is done in Jadex [410] where a logging agent is introduced to allow collection and viewing of logged messages from Jadex agents. It should be noted that the probes in our approach offer the added functionality of filtering on a cognitive condition of *one ore more agents*.

6.6.4 Message-list

Another visualization tool is the message-list, which is one of the simplest forms of visualization. The message-list keeps track of the messages sent between agents, by placing them in a list. This list can be sorted on each of the elements of the messages. For example, sorting the messages on the "sender" element can help finding a specific message send by a known agent. Besides ordering, the list can also be filtered. For example, we could filter on "Senders" and only show the message from the sender with the name "cleaner". To update the message-list on every update of the trace, we can place the marker `MADT(message_list_madt)` in the multi-agent program. Another use of the message-list could be to show only the messages from within a certain group, e.g., `MADT(message_list_madt, @[explorers])` can be used to view the messages exchanged between the members of the explorers group. Finally, in our proposal one can also filter exchanged messages based on conditions on the mental states of individual agents. For example, in the context of our gridworld example, one can filter useless messages, i.e., messages whose content are known facts. Exchanging too many useless messages is a sign of non-effective communication. The example below triggers the message list when an agent `A`, who believes there is a bomb at coordinates `X,Y`, receives a message about this fact from another agent `S`.

```
MADT( message_list_madt,
      eventually( A@Beliefs(bomb(X,Y))
                  and
```

```
                A@Beliefs(message(S,P,bombAt(X,Y)))
            )
    )
```

In this example, it is assumed that a received message is automatically added to the belief base of the receiving agent, and that the added message has the form $message(Sender, Performative, Content)$. All existing agent programming platforms offer a similar tool to visualize exchanged messages. The main difference with our approach is the ability to log when certain cognitive conditions hold.

6.6.5 Causal tree

The causal tree tool shows each message and how it relates to other messages in a tree form. The hierarchy of the tree is based on the relation between messages (replies become branches of the message they reply to). Messages on the same hierarchical level, of the same branch, are ordered chronologically. The advantage of the causal tree (over the message-list) is that it is easier to spot communication errors. When, for example, a reply is placed out of context (not in relation with its cause) this implies there are communication errors. The causal tree also provides an easy overview to see if replies are sent when required. The causal tree tool can be used by adding the marker `MADT(causal_tree_madt)` to multi-agent programs. Another example could be to set the group parameter and only display message from a certain group, e.g., `MADT(causal_tree_madt, @[explorers])`. It should be noted that for the causal tree to work, the messages need to use performatives such as inform and reply.

6.6.6 Sequence diagram

The sequence diagram is a commonly used diagram in the Unified Modeling Language (UML) or its corresponding agent version (AUML). An instantiation of a sequence diagram can be used to give a clear overview of (a specific part of) the communication in a multi-agent program. They can help to find irregularities in the communication between agents. The sequence diagram tool can be used by adding the marker `MADT(sequence_diagram_madt)` to multi-agent programs. This example updates the sequence diagram on every update of the trace. Another example could be to use the group parameter and only update the sequence diagram for the agents in a certain group, e.g., `MADT(sequence_diagram_madt, @[cleaner, explorer2])`. Adding this marker to our multi-agent program will show the communication between the agents “cleaner” and “explorer2”. The sequence diagram tool is useful in conjunction with a conditional breakpoint and the stepwise execution mode where the diagram can be constructed step by step. The sequence diagram

is also useful in conjunction with the probe. The probe can be used to display detailed information about the messages. Similar tools are proposed in some other approaches, e.g., the sniffer agent in [32]. However, we believe that the sequence diagram tool in our approach is more effective since it can be used for specific parts of agent communication.

6.6.7 Visualization

Sometimes the fact that a message is sent is more important than the actual contents of the message. This is, for example, the case when a strict hierarchy forbids certain agents to communicate. In other cases it can be important to know how much communication takes place between agents. For such situations the *dynamic agent communication* tool is a valuable add-on. This tool shows all the agents and represents the communication between the agents by lines. When agents have more communication overhead the line width increase in size and the agents are clustered closer together. This visualization tool, which can be triggered by adding the marker `MADT(dynamic_agent_madt)` to multi-agent program, is shown in figure 6.2.

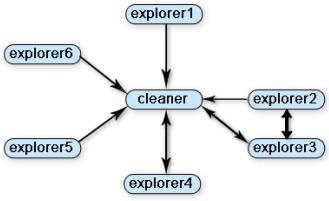

Fig. 6.2 The dynamic agent communication tool.

Another visualization tool is the static group tool. This debugging tool, which shows specific agent groups, is illustrated in figure 6.3. The line between the groups indicates the (amount) of communication overhead between the groups. In addition the developer can "jump into" a group and graphically view the agents and the communication between them.

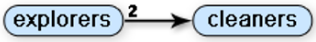

Fig. 6.3 The static group tool.

The static group tool can be helpful to quickly check if the correct agents are in the correct group. It can also be used to check communication between different groups. If two groups show an unusual amount of communication overhead the developer can jump into the group and locate the source of the problem. The marker to activate the static group tool can be specified as follow:

```
MADT(static_group_madt, @[explorers])
MADT(static_group_madt, @[cleaners])
```

The above markers update the tool on every change of the multi-agent program trace. According to these markers, the groups "explorers" and "cleaners" will be visualized. Generally it is most valuable to have a visualization of all communication between agents. However, to pinpoint the exact problem in a communication protocol it can be an invaluable addition to use a condition, which filters the messages that are shown. These same principles apply to the filtered view. As discussed in the related works section, other approaches (e.g., [84]) offers similar tools.

6.7 Conclusion and Future Work

In this chapter we have shown two methods for showing the correctness of BDI-based multi-agent programs that are complementary to the well-known methods of model checking and theorem proving that are used in the realm of multi-agent verification. Various chapters in this volume present different model-checking and theorem proving approaches for multi-agent programs.

The first one is a general one allowing us to prove that certain properties expressed in a specification language hold for the agent programming language, and thus for all individual agents that are implemented in this agent programming language. To this end we showed how a BDI-based agent-oriented programming language can be related to a BDI specification language in a systematic and natural manner. We used here a very simple agent programming language which can be extended in many different ways. In [77], a comparable effort is undertaken for the agent programming language AgentSpeak. The specification language in that work is, however, not a temporal logic, and the properties proven are different (not related to dynamics of goals).

The multi-agent programming language and its corresponding logic, presented in this chapter, are designed to focus on the cognitive aspects and related properties of individual agent programs. The multi-agent programming language can be extended with communication actions and shared environments to allow the implementation of multi-agent systems in which individual agents interact by either sending and receiving messages or performing actions in their shared environment. Most existing agent programming languages have already proposed specialized constructs to implement communication and shared environments. Future research is needed to

propose logical frameworks to specify and verify the interaction properties of multi-agent programs.

The properties that we have studied in this chapter are general in the sense that we consider the set of all possible execution traces of multi-agent programs. In practice, the execution of a multi-agent program is often based on an interpreter that uses a specific execution strategy. For example, an interpreter may apply one/all planning rules before executing one/all plans, or executing one/all plans before applying one/all planning rules. In principle, there are many different strategies that can be used to execute a multi-agent program. In future work we will study properties that are related to a specific execution strategy. This enables the verification of properties of a multi-agent program for a given interpreter.

The second approach to the correctness of BDI-based agent programs we showed is based on debugging. Our proposal extends previous approaches by debugging the *interaction* between implemented agents, not only in terms of the exchanged messages, but also in terms of the relations between their internal states. A developer/debugger of a multi-agent program is assumed to have access to the multi-agent program code, which is a realistic assumption, and therefore to the internal state of those programs. The proposed approach is based on a specification language to express cognitive and temporal properties of the executions of multi-agent programs. The expressions of the specification language can be used to trigger debugging tools such as breakpoints, watches, probes, and different visualization tools to examine and debug communication between individual agents.

Since the specification language is abstract, our debugging approach is generic and can be modified and applied to other BDI-based agent programming languages. The only modification is to align the evaluation function of the specification language with the programming language at hand. We have already applied this debugging approach to 2APL [122] platform by modifying its corresponding interpreter to process debugging markers in both debugging modes. The 2APL interpreter evaluates the expressions of the specification language based on the partial execution trace of the multi-agent programs. We have also implemented the proposed debugging tools that are discussed in this paper for the 2APL platform.

We plan to extend the MDL language by including constructs related to the external environments of a multi-agent program. In this way, one can specify properties that relates agent states to the state of the external environments. Moreover, we plan to extend our debugging framework with the society aspects that may be involved in multi-agent programming languages [325]. Recent developments in multi-agent programming languages [131, 166, 188, 243] have proposed specific programming constructs enabling the implementation of social concepts such as norms, roles, obligations, and sanctions. Debugging such multi-agent programs requires therefore specific debugging constructs to specify properties related to the social aspects and facilitate finding and resolving defects involved in such programs.

The presented debugging framework assumes all agents are developed on one single platform such that their executions for debugging purposes are not distributed on different platforms. One important challenge and a future work on debugging

multi-agent systems remains the debugging of multi-agent programs that run simultaneously on different platforms. The existing debugging techniques are helpful when errors manifest themselves directly to the system developers. However, errors in a program do not always manifest themselves directly. For mission and industrial critical systems it is therefore necessary to extensively test the program before deploying it. This testing should remove as many bugs (and possible defects) as possible. However, it is infeasible to test every single situation the program could be in. We believe that a systematic integration of debugging and testing approaches can be effective in verifying the correctness of multi-agent programs and therefore essential for their developments. A testing approach proposed for multi-agent programs is proposed by Poutakidis and his colleagues [345, 346].

Acknowledgements

We would like to thank Birna van Riemsdijk for her contribution to work on which this chapter is partly based.