Proceedings

Ninth International Workshop on
Programming Multi-Agent Systems

ProMAS 2011

Taipei, Taiwan
May 3rd, 2011

`http://www.inf.ufrgs.br/promas2011/`

Held with the Tenth International Joint Conference on
Autonomous Agents and Multi-Agent Systems (AAMAS 2011)

Edited by

Louise A. Dennis, Olivier Boissier, Rafael H. Bordini

Preface

The ProMAS workshop series has produced, throughout this decade, a number of solid contributions towards programming languages and development tools that are appropriate for the development of complex autonomous systems that operate in dynamic environments. With applications of autonomous software (e.g., UAVs, companion robots, ambient intelligence, and semantic applications, to name just a few) becoming required with wide commercial interest, it is imperative to support the ever more complex task of professional programmers of multi-agent systems. Importantly, such languages and tools must be developed in a principled but practical way. ProMAS aims to address both theoretical and practical issues related to developing and deploying multi-agent systems.

Now in its 9th edition, ProMAS has been an invaluable venue bringing together leading researchers from both academia and industry to discuss issues on the design of programming languages and tools for multi-agent systems. In particular, the workshop promotes the discussion and exchange of ideas concerning the techniques, concepts, requirements, and principles that are important for multi-agent programming technology. These include the theory and applications of agent programming languages, how to effectively implement a multi-agent system design or specification, the verification and analysis of agent systems, as well as the implementation of social structures in agent-based systems (e.g., organisations, coordination, and communication in multi-agent systems).

May 2011

Louise A. Dennis
Olivier Boissier
Rafael H. Bordini

Programme Committee

Matteo Baldoni (University of Torino, Italy)
Juan Botia (Universidad de Murcia, Spain)
Lars Braubach (University of Hamburg, Germany)
Rem Collier (University College Dublin, Ireland)
Ian Dickinson (Epimorphics Ltd., UK)
Marc Esteva (IIIA-CSIC, Spain)
Michael Fisher (University of Liverpool, UK)
Jorge Gomez-Sanz (Computense University of Madrid, Spain)
Vladimir Gorodetsky (IIAS, Russia)
Dominic Greenwood (Whitestein Technologies AG, Switzerland)
James Harland (RMIT University, Australia)
Koen Hindriks (TU Delft, Netherlands)
Benjamin Hirsch (Technical University of Berlin, Germany)
Jomi Hübner (Federal University of Santa Catarina, Brazil)
João Leite (New University of Liston, Portugal)
Brian Logan (University of Nottingham, UK)
Viviana Mascardi (University of Genova, Italy)
Philippe Mathieu (University of Lille, France)
John-Jules Meyer (Utrecht University, Netherlands)
Jörg Müller (TU Clausthal, Germany)
Andrea Omicini (University of Bologna, Italy)
Agostino Poggi (University of Parma, Italy)
Alexander Pokahr (University of Hamburg, Germany)
Alessandro Ricci (University of Bologna, Italy)
Birna van Riemsdijk (TU Deflt, Netherlands)
Ralph Ronnquist (Intendico Pty Ltd, Australia)
Ichiro Satoh (National Institute of Informatics, Japan)
Michael Ignaz Schumacher (HES-SO, Switzerland)
Munindar Singh (NCSU, USA)
Tran Cao Son (New Mexico State University, USA)
Patrick Taillibert (Thales Aerospace Division, France)
Paolo Torroni (University of Bologna, Italy)
Jørgen Villadsen (DTU Informatics, Denmark)
Gerhard Weiss (University of Maastricht, Netherlands)
Michael Winikoff (University of Otago, New Zealand)
Neil Yorke-Smith (American University of Beirut and SRI)

Committees

Organising Committee

Olivier Boissier, Ecole des Mines de St Etienne, France
Rafael H. Bordini, Federal University of Rio Grande do Sul, Brazil
Louise A. Dennis, University of Liverpool, UK

Steering Committee

Rafael H. Bordini, Federal University of Rio Grande do Sul, Brazil
Mehdi Dastani, Utrecht University, The Netherlands
Jürgen Dix, Clausthal University of Technology, Germany
Amal El Fallah Seghrouchni, Univesity of Paris VI, France

Workshop Schedule

09:00–09:30	Workshop Opening
	Session 1
09:30–10:30	**Invited Talk** TBA
10:30–11:00	**Coffee Break**
11:00–12:00	**Foundations of Agent Programming Languages**
	Logical Foundations for a Rational BDI Agent Programming Language (Extended Version) (Shakil M. Khan and Yves Lespérance)
	A Coupled Operational Semantics for Goals and Commitments (Pankaj R. Telang, Neil Yorke-Smith, and Munindar P. Singh)
12:00–13:00	**Applying (Multi-)Agent Oriented Programming**
	Developing a Knoweldge Management Multi-Agent System Using JaCoMo (Carlos M. Toledo, Rafael H. Bordini, Omar Chiotti, and Maria R. Galli)
	Notes on pragmatic agent-programming with Jason (Radek Pibil, Peter Novák, Cyril Brom, and Jakub Gemrot)
13:00–14:00	**Lunch**
14:00–15:30	**Programming Languages and Platforms**
	The agent programming language Meta-APL (Thu Trang Doan, Natasha Alechina, and Brian Logan)
	BDI4JADE: a BDI layer on top of JADE (Ingrid Nunes, Carlos J. P. de Lucena, and Michael Luck)
	Integrating Expectation Handling into Jason (Surangika Ranathunga, Stephen Cranefield, and Martin Purvis)
15:30–16:00	**Coffee Break**
16:00–17:30	**Model Checking**
	Abstraction for Model Checking Modular Interpreted Systems over ATL (Michael Köster and Peter Lohmann)
	MAS: Qualitative and Quantitative Reasoning (Ammar Mohammed and Ulrich Furbach)
	State Space Reduction for Model Checking Agent Programs (Sung-Shik T. Q. Jongmans, Koen V. Hindriks, and M. Birna van Riemsdijk)
17:30–18:00	**Closing Session**
	Multi-Agent Programming Contest (Announcement) (Tristan Behrens, Jürgen Dix, Jomi Hübner, Michael Köster, and Federico Schlesinger)
	Workshop Close

Contents

Chapter 1

Foundations of (Multi-)Agent Programming

Logical Foundations for a Rational BDI Agent Programming Language (Extended Version)*

Shakil M. Khan and Yves Lespérance

Department of Computer Science and Engineering
York University, Toronto, ON, Canada
{skhan, lesperan}@cse.yorku.ca

Abstract. To provide efficiency, current BDI agent programming languages with declarative goals only support a limited form of rationality – they ignore other concurrent intentions of the agent when selecting plans, and as a consequence, the selected plans may be inconsistent with these intentions. In this paper, we develop logical foundations for a rational BDI agent programming framework with prioritized declarative goals that addresses this deficiency. We ensure that the agent's chosen declarative goals and adopted plans are consistent with each other and with the agent's knowledge. We show how agents specified in our language satisfy some key rationality requirements.

1 Introduction

This paper contributes to the foundations of Belief-Desire-Intention agent programming languages/frameworks (BDI APLs), such as PRS [10], AgentSpeak [19], etc. Recently, there has been much work on incorporating *declarative goals* in these APLs [7, 28, 21, 5, 27, 22]. In addition to defining a set of plans that can be executed to try to achieve a goal, these programming languages also incorporate goals as declarative descriptions of the states of the world which are sought. A typical BDI APL with declarative goals (APLwDG) uses a user-specified hierarchical plan library Π containing abstract plans, a procedural goal-base Γ containing a set of plans that the agent is committed to execute, and a declarative goal-base Δ that has goals that the agent is committed to achieve. In response to events in the environment and to goals in Δ, in each cycle the agent interleaves selecting plans from Π, adopting them to Γ, and executing actions in Γ. The execution of some of these actions can in turn trigger the adoption of other declarative goals. This process is repeated until all the goals in Δ are successfully achieved. The role of these declarative goals in an APLwDG is essentially for monitoring goal achievement and performing recovery when a plan has failed by decoupling plan failure/success from that of goal. Since these declarative goals capture the reason for executing plans, they are necessary to perform rational deliberation, and react in a rational way to changes in goals that result from communication, e.g. requests.

While current APLwDGs have evolved over the past few years — e.g. some of them handle restricted forms of temporally extended goals [8] — to keep them tractable

* This paper is an extended version of [16] and is also a revised version of [14].

and practical, they sacrifice some principles of rationality. In particular, while selecting plans to achieve a declarative goal, they ignore other concurrent intentions of the agent. As a consequence, the selected plan may be inconsistent with the agent's other intentions. Thus the execution of such an intended plan can render other contemporary intentions impossible to bring about. Also, these APLwDGs typically rely on syntactic formalizations of declarative goals, subgoals, and their dynamics, whose properties are often not well understood.

Apart from this, there has been work that focuses on maintaining consistency of a set of concurrent intentions. For example, Clement et al. [3,4] argue that agents should be able to reason about abstract HTN plans and their interactions before they are fully refined. They propose a method for deriving summary information (i.e. external preconditions and effects) of abstract plans and discuss how this information can be used to coordinate the interactions of plans at different levels of abstractions. Thangarajah et al. [26] use such summary information to detect and resolve conflicts between goals at run time. Horty and Pollack [9] propose a decision theoretic approach to compute the utility of adopting new (non-hierarchical) plans, given a set of already adopted plans. While some of these approaches can be integrated in APLs (e.g. [26]), they leave out many aspects of rationality (e.g. they do not say what the agent should do if external interference makes two of her intentions permanently incompatible), and do not deal with declarative goals.

In this paper, we develop a logical framework for a rational BDI APL with prioritized declarative goals called Simple Rational APL (SR-APL, henceforth), that addresses these deficiencies of previous APLwDGs. Our framework combines ideas from the situation calculus-based Golog family of APLs (e.g. [6]), our expressive semantic formalization of prioritized goals, subgoals, and their dynamics [13, 15], and work on BDI APLs. We ensure that the agent's chosen declarative goals and adopted plans are consistent with each other and with the agent's knowledge. In doing this, we must address two fundamental questions about rational agency: (1) *What does it mean for a BDI agent to be committed to concurrently execute a set of plans next while keeping the option of further commitments to other plans open, in a way that does not allow procrastination?* (2) *How to ensure consistency between an agent's adopted declarative goals and adopted plans, given that some of the latter might be abstract, i.e. might be only partially instantiated in the sense that they include subgoals for which the agent has not yet adopted a (concrete) plan?* We show how agents specified in our framework satisfy some key rationality requirements. We discuss how new practical programming languages can be developed by restricting the proposed representation and reasoning. Our framework tries to bridge the gap between agent theories and practical APLs by providing a model and specification of an idealized BDI agent whose behavior is closer to what a rational agent does. As such, it allows one to understand how compromises made during the development of a practical APLwDG affect the agent's rationality.

The paper is organized as follows: in the next section, we discuss a motivating example. In Sections 3 and 4, we outline our formal BDI framework. In Section 5, we specify the semantics of SR-APL. In Section 6, we show that in the absence of external interference, our agent behaves in ways that satisfy some key rationality principles. Then in Section 7, we summarize our results and discuss possible future work.

2 A Motivating Example

Consider a blocks world domain, where each block is one of four possible colors: blue, yellow, green, and red. There is only a stacking action $stack(b, b')$: b can be stacked on b' in state s if $b \neq b'$, both b and b' are *clear* in s, and b is *on the table* in s. There are no unstacking actions, so the agent cannot use a block to build two different towers at different times. Assume that there are four blocks, B_B, B_Y, B_G, and B_R, one of each color. the agent knows the *color of* these blocks, and knows that initially all the blocks are on the table and are clear. Now assume that the agent has the following two goals: (1) to eventually have a 2 blocks tower that has a green block on top and a non-yellow block underneath, and (2) to have a 2 blocks tower with a blue block on top and a non-red block underneath; thus $\Delta = \{\Diamond \text{Twr}^G_Y, \Diamond \text{Twr}^B_R\}$, where $\text{Twr}^{C_1}_{C_2} \doteq \exists b, b'.\ \text{OnTbl}(b') \wedge \text{On}(b, b') \wedge \neg \text{C}_2(b') \wedge \text{C}_1(b)$. Suppose our agent's plan library Π has two rules:

$$\begin{aligned}
\Diamond \text{Twr}^G_Y : [&\text{OnTbl}(b) \wedge \text{OnTbl}(b') \wedge b \neq b' \wedge \text{Clear}(b) \\
&\wedge \text{Clear}(b') \wedge \neg \text{Y}(b) \wedge \text{G}(b')] \leftarrow stack(b', b), \\
\Diamond \text{Twr}^B_R : [&\text{OnTbl}(b) \wedge \text{OnTbl}(b') \wedge b \neq b' \wedge \text{Clear}(b) \\
&\wedge \text{Clear}(b') \wedge \neg \text{R}(b) \wedge \text{B}(b')] \leftarrow stack(b', b).
\end{aligned}$$

That is, if the agent has the goal to have a green and non-yellow tower and knows about a green block b' and a distinct non-yellow block b that are both clear and are on the table, then she should adopt the plan of stacking b' on b, and similarly for the goal of having a blue and non-red tower.

Now, consider a typical APLwDG, that (without considering the overall consistency of the agent's intentions) simply select plans from Π for the agent's goals in Δ and eventually executes them in an attempt to achieve her goals. We claim that such an APL is not always sound and rational. For instance, according to this plan library, one way of building a green non-yellow (and a blue non-red) tower is to construct a green-blue (a blue-green, respectively) tower. While these two plans are individually consistent, they are inconsistent with each other, since the agent has only one block of each color. Thus a rational agent should not adopt these two plans. However, it can be shown that the following would be a legal trace for our blocks world domain in such an APL:

$$\langle \{\}, \Delta \rangle \Rightarrow \langle \{\sigma_1\}, \Delta \rangle \Rightarrow \langle \{\sigma_1, \sigma_2\}, \Delta \rangle \Rightarrow \langle \{\sigma_2\}, \{\Diamond \text{Twr}^G_Y\} \rangle.$$

The agent first moves to configuration $\langle \{\sigma_1\}, \Delta \rangle$ by adopting the plan $\sigma_1 = stack(B_B, B_G)$ in response to $\Diamond \text{Twr}^B_R$, then to $\langle \{\sigma_1, \sigma_2\}, \Delta \rangle$ by adopting $\sigma_2 = stack(B_G, B_B)$ to handle $\Diamond \text{Twr}^G_Y$, and then to $\langle \{\sigma_2\}, \{\Diamond \text{Twr}^G_Y\} \rangle$ by executing the intended action σ_1. At this point, the agent is stuck and cannot complete successfully. Thus, in such an APL, not only is the agent allowed to adopt two inconsistent plans, but the execution of one of these plans makes other concurrent goals impossible (e.g. the execution of $stack(B_B, B_G)$ makes $\Diamond \text{Twr}^G_Y$ impossible to achieve).

The problem arises in part because actions are not reversible in this domain; there is no action for moving a block back to the table or for unstacking it. This is common in real world domains, for instance, most tasks with deadlines or resources, e.g. doing some errands before noon, a robot delivering mail without running out of battery power,

etc. While such irrational behavior could in principle be avoided by using appropriate conditions in the antecedent of the plan-selection rules (e.g. by stating that the agent should only adopt a given plan if she does not have certain other goals), this puts an excessive burden on the agent programmer. Ideally, such reasoning about goals should be delegated to the agent.

3 Preliminaries

Our base framework for modeling goal change is the situation calculus as formalized in [17, 20]. In this framework, a possible state of the domain is represented by a situation. There is a set of initial situations corresponding to the ways the agent believes the domain might be initially, i.e. situations in which no actions have yet occurred. Init(s) means that s is an initial situation. The actual initial state is represented by a special constant S_0. There is a distinguished binary function symbol do where $do(a, s)$ denotes the successor situation to s resulting from performing the action a. Thus the situations can be viewed as a set of trees, where the root of each tree is an initial situation and the arcs represent actions. Relations (and functions) whose truth values vary from situation to situation, are called relational (functional, respectively) fluents, and are denoted by predicate (function, respectively) symbols taking a situation term as their last argument. There is a special predicate Poss(a, s) used to state that action a is executable in situation s. Finally, the function symbol Agent(a) denotes the agent of action a.

We use a theory $\mathcal{D}$ that includes the following set of axioms:[1] (1) action precondition axioms, one per action a characterizing Poss(a, s), (2) successor state axioms (SSA), one per fluent, that succinctly encode both effect and frame axioms and specify exactly when the fluent changes [20], (3) initial state axioms describing what is true initially including the mental states of the agents, (4) axioms identifying the agent of actions, one per action a characterizing Agent(a), (5) unique name axioms for actions, and (6) domain-independent foundational axioms describing the structure of situations [17].

Following [23], we model knowledge using a possible worlds account adapted to the situation calculus. $K(s', s)$ is used to denote that in situation s, the agent thinks that she could be in situation s'. Using K, the knowledge of an agent is defined as: Know$(\Phi, s) \doteq \forall s'.\ K(s', s) \supset \Phi(s')$, i.e. the agent knows Φ in s if Φ holds in all of her K-accessible situations in s. K is constrained to be reflexive, transitive, and Euclidean in the initial situation to capture the fact that agents' knowledge is true, and that agents have positive and negative introspection. The dynamics of knowledge is specified by providing a SSA for K that supports knowledge expansion as a result of sensing actions [23] and some *informing* communicative actions [12]. As shown in [23], the constraints on K continue to hold after any sequence of actions since they are preserved by the SSA for K. We also assume that the agent is aware of all actions.

To support modeling temporally extended goals, we introduced a new sort of *paths* along with an axiomatization for paths in [13]. A path is essentially an infinite sequence of situations, where each situation along the path can be reached by performing some *executable* action in the preceding situation. We use (possibly sub/super-scripted) variables p to denote paths. There is a predicate OnPath(p, s), meaning that the situation s

[1] We will be quantifying over formulae, and thus assume $\mathcal{D}$ includes axioms for encoding of formulae as first order terms, as in [25]. We will also be using lists of programs, and assume that $\mathcal{D}$ includes an axiomatization of lists.

is on path p. Also, Starts(p, s) means that s is the starting situation of path p. A path p starts with s iff s is the earliest situation on p.

We use $\Phi(s), \Psi(s), \cdots$, etc. to denote *state formulae* in the context of knowledge (and $\phi(p), \psi(p), \cdots$, etc. for *path formulae* in that of goals), each of which has a free situation variable s (path variable p, respectively). s (and p) will be bound by the context where the formula $\Phi(s)$ (and $\phi(p)$, respectively) appears. Where the intended meaning is clear, we sometimes suppress the situation variable (path variable) from $\Phi, \Psi, \cdots$, etc. ($\phi, \psi, \cdots$, etc. respectively). Also, we often use now to refer to a placeholder constant that stands for the current situation.

We will use some useful constructs that are defined in [13]. A state formula Φ *eventually holds* over the path p if Φ holds in some situation that is on p, i.e.: $\Diamond\Phi(p) \doteq \exists s'.\ \text{OnPath}(p, s') \wedge \Phi(s')$. Secondly, Suffix$(p', p, s)$ means that path p' is a suffix of another path p w.r.t. a situation s; Suffix(p', p, s) holds iff s is on p, and p' is the subpath of p that starts with s. Finally, SameHist(s_1, s_2) means that the situations s_1 and s_2 share the same history of actions, but perhaps starting from different initial situations.

4 Formalization of Prioritized Goals

In [13], we proposed a logical framework for modeling *prioritized goals* and their dynamics. In that framework, an agent can have multiple *goals* or *desires* at different priority levels, possibly inconsistent with each other. We specify how these goals evolve when actions/events occur and the agent's knowledge changes. We define the agent's *chosen goals* or *intentions*, i.e. the goals that the agent is actively pursuing, in terms of this goal hierarchy. In that framework, agents constantly optimize their chosen goals. To this end, we keep all prioritized goals in the goal-base unless they are explicitly dropped. At every step, we compute an optimal set of chosen goals given the hierarchy of prioritized goals, preferring higher priority goals, such that chosen goals are consistent with each other and with the agent's knowledge. Thus at any given time, some goals in the hierarchy are *active*, i.e. chosen, while others are *inactive*. Some of these inactive goals may later become active (e.g. if a higher priority active goal that is currently blocking an inactive goal becomes impossible or is dropped) and trigger the inactivation of other currently active (lower priority) goals.

Goal Semantics As in [13], we specify the agent's prioritized goals or *p-goals* using accessibility relation/fluent G. A path p is G-accessible at priority level n in situation s if all the goals of the agent at level n are satisfied over this path and if it starts with a situation that has the same action history as s. The latter requirement ensures that the agent's G-accessible paths are compatible with the actions that have been performed so far. We say that an agent has the p-goal that ϕ at level n in situation s (i.e. PGoal(ϕ, n, s)) iff ϕ holds over all paths that are G-accessible at n in s. A smaller n represents higher priority, and the highest priority level is 0. Thus as in [13], we assume that the set of p-goals are totally ordered according to priority. Note that, in this framework one can evaluate goals over infinite paths and thus can handle arbitrary temporally extended goals; hence, unlike some other situation calculus based accounts where goal formulae are evaluated w.r.t. finite paths (e.g. [24]), in this framework one can handle, for example, unbounded maintenance goals.

As in [13], we allow the agent to have infinitely many p-goals. However in many cases, the modeler will want to specify a finite set of initial p-goals. When a finite num-

ber of p-goals is assumed, we can use the functional fluent $NPGoals(s)$ to represent the number of prioritized goals that the agent has in situation s. The modeler/programmer will usually provide some specification of the agent's initial p-goals at the various priority levels, using some *initial goal axioms*. For instance, the initial prioritized goals for our blocks world example with domain theory $\mathcal{D}_{BW}$ can be specified as follows:

$$\begin{aligned}
&\text{(a) } \text{Init}(s) \supset ((G(p,0,s) \equiv \exists s'.\ \text{Starts}(p,s') \wedge \text{Init}(s') \wedge \Diamond \text{Twr}^G_Y) \\
&\qquad\qquad \wedge (G(p,1,s) \equiv \exists s'.\ \text{Starts}(p,s') \wedge \text{Init}(s') \wedge \Diamond \text{Twr}^B_R)), \\
&\text{(b) } \forall n,p,s.\ \text{Init}(s) \wedge n \geq 2 \supset (G(p,n,s) \equiv \exists s'.\ \text{Starts}(p,s') \wedge \text{Init}(s')).
\end{aligned}$$

(a) specifies the p-goals of the agent in the initial situations (we assume that the goal $\Diamond \text{Twr}^G_Y$ has higher priority than $\Diamond \text{Twr}^B_R$); (b) makes $G(p,n,s)$ true for every path p that starts with an initial situation for $n \geq 2$. Thus at these levels, the agent has the trivial p-goal that she be in an initial situation.

An agent's chosen goals must be realistic. To filter out the paths that are known to be impossible from G, we define *realistic* p-goal accessible paths: p is G_R-accessible at level n in s if it is G-accessible at n in s and if it starts with a situation that is K-accessible in s. In our framework, an agent has the *realistic p-goal* that ϕ at level n in situation s (i.e. $\text{RPGoal}(\phi, n, s)$) iff ϕ holds over all G_R-accessible paths at n in s.

We define chosen goals or *c-goals* using realistic p-goals. Note that an agent's realistic p-goals at various priority levels can be viewed as candidates for her c-goals. Given the set of realistic p-goals, in each situation the agent's c-goals are specified to be those that are in the maximal consistent set of higher priority realistic p-goals. We define this iteratively starting with a set that contains the highest priority realistic p-goal accessible paths, i.e. G_R-accessible paths at level 0. At each iteration we obtain the intersection of this set with the set of next highest priority G_R-accessible paths. If the intersection is not empty, a new chosen set of p-goal accessible paths (and p-goals defined by these paths) at level i is obtained. We call a p-goal chosen by this process an *active* p-goal. If on the other hand the intersection is empty, then it must be the case that the p-goal represented by this level is either in conflict with another active higher priority p-goal/a combination of two or more active higher priority p-goals, or is known to be impossible. In that case, that p-goal is ignored (i.e. marked as *inactive*), and the chosen set of p-goal accessible paths at level i is the same as at level $i-1$. To get the prioritized intersection of the set of G_R-accessible paths up to level n, the process is repeated until $i = n$ is reached. $G_\cap(p,n,s)$ is used to denote that in situation s, path p is in the prioritized intersection of G_R-accessible paths up to level n. We say that a path p is $G_\cap$-accessible in situation s, i.e. $G_\cap(p,s)$, if $G_\cap(p,n,s)$ holds for all levels n. Finally, we say that an agent has the c-goal that ϕ in situation s (i.e. $\text{CGoal}(\phi, s)$) if ϕ holds over all $G_\cap$-accessible paths in s. We can show that initially our blocks world agent has the p-goals/c-goals that $\Diamond \text{Twr}^G_Y$ and $\Diamond \text{Twr}^B_R$, i.e.: $\mathcal{D}_{BW} \models \forall s.\ \text{Init}(s) \supset \text{CGoal}(\Diamond \text{Twr}^G_Y \wedge \Diamond \text{Twr}^B_R, s)$.

To get positive and negative introspection of goals, we impose two inter-attitudinal constraints on the K and G-accessibility relations in the initial situations. We have shown that these constraints then continue to hold after any sequence of actions since they are preserved by the SSAs for K and G. See [11] for details.

Goal Dynamics An agent's goals change when her knowledge changes as a result

of the occurrence of an action (including exogenous events), or when she adopts or drops a goal. There are two special actions, for *adopting a p-goal ϕ at some level n* and *dropping a p-goal ϕ*, $adopt(\phi, n)$ and $drop(\phi)$, and a third action for *adopting a subgoal ψ relative to a supergoal ϕ*, $adoptRT(\psi, \phi)$.

The dynamics of p-goals are specified using a SSA for G as follows (the agent's c-goals are automatically updated when her p-goals change). Firstly, to handle the occurrence of a non-adopt/drop action a, all p-goals are progressed to reflect the fact that this action has occurred. Secondly, to handle adoption of a p-goal ϕ at level m, a new formula containing the p-goal is added to the agent's goal hierarchy at m. To be precise, in addition to progressing all p-goals at all levels, a new level containing the p-goal that ϕ is inserted at m and all current levels with priority greater or equal to m are pushed one level down the hierarchy. Finally, to handle the dropping of a p-goal ϕ, the levels that imply the dropped goal in the agent's goal hierarchy are replaced by the trivial formula that the history of actions in the current situation has occurred, and thus the agent no longer has the p-goal that ϕ. See [13] for details.

Handling Subgoals We also handle subgoal adoption and model the dependencies between goals and the subgoals and plans adopted to achieve them. The latter is important since subgoals and plans adopted to bring about a goal should be dropped when the parent goal becomes impossible, or is dropped. We handle this as follows: adopting a subgoal ψ relative to a parent goal ϕ adds a new p-goal that contains *both this subgoal and this parent goal*, i.e. $\psi \wedge \phi$. This ensures that when the parent goal is dropped, the subgoal is also dropped, since when we drop the parent goal ϕ, all the p-goals at all G-accessibility levels that imply ϕ including $\psi \wedge \phi$ are also dropped. Note that the parent goal ϕ could be a p-goal at multiple levels. We assume that the subgoal ψ is always adopted w.r.t. the *highest priority supergoal level*, i.e. the highest priority level where ϕ holds. Also, the subgoal ψ is always adopted at the level immediately below the supergoal ϕ's level. The reason for doing this is that since ψ is a means to the end ϕ, they should have similar priorities. ψ is said to be a subgoal of ϕ in situation s (i.e. SubGoal(ψ, ϕ, s)) iff there is a G-accessibility level n in s such that ϕ is a p-goal at n while ψ is not, and for all G-accessibility levels in s where ψ is a p-goal, ϕ is also a p-goal. See [15, 11] for details of our formalization of subgoals.

Prioritized Goals for Committed Agents The formalization of prioritized goal dynamics in [13] ensures that the agent always tries to optimize her chosen goals. She will abandon a c-goal ϕ if an opportunity to commit to a higher priority but inconsistent with ϕ goal arises. As such, our account in [13] displays an idealized form of rationality. This is in contrast to Bratman's [1] practical rationality that takes into consideration the resource-boundedness of real world agents. According to Bratman, intentions limit the agent's reasoning as they serve as a *filter for adopting new intentions*. However, the agent is allowed to override this filter in some cases, e.g. when adopting ϕ increases her utility considerably. The framework in [13] can be viewed as a theory of intention where the filter override mechanism is always triggered.

Note that, in that framework, the agent's c-goals are very dynamic. For instance, as mentioned earlier, a currently inactive p-goal ϕ may become active at some later time, e.g. if a higher priority active c-goal that is currently blocking ϕ (as it is inconsistent with ϕ) becomes impossible. This also means that another currently active c-goal ψ

may as a result become inactive, not because ψ has become impossible, was achieved, or was dropped, but due to the fact that ψ has lower priority than and is inconsistent with the newly activated goal ϕ (see [13] for a concrete example).

Such very dynamic c-goals/intentions are problematic as a foundation for an APL, as the agent spends a lot of effort in "recomputing" her intentions and plans to achieve them, and her behavior becomes hard to predict for the programmer. To avoid this, here we use a modified version of our formalization in [13] that eliminates the filter override mechanism altogether so that agents' p-goals/desires are dropped as soon as they become inactive. We can do this with the following simple changes: (1) we require that initially the agent knows that her p-goals are all possible and consistent with each other, (2) we don't allow the agent to adopt p-goals that are inconsistent with her current c-goals/intentions, and (3) we modify the SSA for G so that the agent's p-goals are dropped when they become impossible or inconsistent with other higher priority c-goals. In the resulting "committed agent" framework, an agent's p-goals are much more dynamic than in the original framework. On the other hand, her c-goals are now much more persistent, and are simply the consequential closure of her desires, as these must now all be consistent with each other and with the agent's knowledge. The resulting model of goals is somewhat simplistic, but is sufficient in an APL context.

5 Agent Programming with Prioritized Goals

Our proposed framework SR-APL combines elements from BDI APLs such as AgentSpeak [19] and from the ConGolog APL [6], which is defined on top of the situation calculus. In addition, to facilitate monitoring of goal achievement and performing plan failure recovery, we incorporate declarative goals in SR-APL. To specify the operational semantics of plans in SR-APL, we will use a subset of the ConGolog APL. This subset includes programming constructs such as primitive actions a, wait/test actions $\Phi?$, sequence of actions $\delta_1; \delta_2$, nondeterministic choice of arguments $\pi v.\ \delta$, nondeterministic iteration δ^*, and concurrent execution of programs $\delta_1 \| \delta_2$, to mention a few. Also, as in ConGolog, we will use $\text{Trans}(\sigma, s, \sigma', s')$ to say that program σ in situation s can make a single step to reach situation s' with the program σ' remaining, and $\text{Final}(\sigma, s)$ to mean that the program σ may legally terminate in situation s. Finally, $\text{Do}(\sigma, s, s')$ means that there is a terminating execution of program σ that starts in s and ends in s'.

Components of SR-APL First of all, we have a *set of axioms/theory* $\mathcal{D}$ specifying actions that can be done, the initial knowledge and (both declarative and procedural) goals of the agent, and their dynamics, as discussed in Section 3 and 4. Moreover, we also have a *plan library* Π with rules of the form $\phi : \Psi \leftarrow \sigma$, where ϕ is a goal formula, Ψ is a knowledge formula, and σ is a plan; a rule $\phi : \Psi \leftarrow \sigma$ means that if the agent has the c-goal that ϕ and knows that Ψ, then she should consider adopting the plan that σ. The *plan language* for σ is a simplified version of ConGolog and includes the empty program nil, primitive actions a, waiting for a condition $\Phi?$, sequence $(\sigma_1; \sigma_2)$, and the special action for subgoal adoption, $adoptRT(\Diamond\Phi, \sigma)$; here $\Diamond\Phi$ is a subgoal to be adopted and σ is the plan relative to which it is adopted.[2] While our account of

[2] We use the ConGolog APL here because it has a situation calculus-based semantics that is well specified and compatible with our agent theory. We could have used any APL with these characteristics.

goal change is expressive enough to handle arbitrary temporally extended goals, here we focus on achievement goals and procedural goals exclusively. We believe that extending our framework to support maintenance goals should be straightforward, since maintenance goals behave like additional constraints on the agent behavior in contrast to achievement goals for which the agent needs to plan for.

Semantics of SR-APL An SR-APL agent can work on multiple goals at the same time. Thus at any time, an agent might be committed to several plans that she will execute in an interleaved fashion. We use our situation calculus domain theory $\mathcal{D}$ to model *both adopted declarative goals and plans*. Initially $\mathcal{D}$ only contains declarative goals. As specified by the SSA for G, $\mathcal{D}$ is updated by adding plans or other declarative goals to the agent's goal hierarchy when a transition rule (see below) makes the agent perform an $adopt$ or $adoptRT$ action. We ensure that an agent's declarative goals and adopted plans are consistent with each other and with the agent's knowledge. In our semantics, we specify this by ensuring that there is at least one possible course of actions (i.e. a path) known to the agent, and if she were to follow this path, she would end up realizing all of her declarative goals and executing all of her procedural goals.

One way of specifying an agent's commitment to execute a plan σ next in $\mathcal{D}$ is to say that she has the intention that $\text{Starts}(s) \wedge \exists s'.\ \text{OnPath}(s') \wedge \text{Do}(\sigma, s, s')$, i.e. that each of her intention-accessible paths p is such that it starts with some situation s, it has the situation s' on it, and s' can be reached from s by executing σ. However, this does not allow for the interleaved execution of several plans, since Do requires that σ be executed before any other actions/plans.

A better alternative is to represent the procedural goal as $\text{Starts}(s) \wedge \exists s'.\ \text{OnPath}(s') \wedge \text{DoAL}(\sigma, s, s')$, which says that the agent has the intention to execute *at least* the program σ next, and possibly more. $\text{DoAL}(\sigma, s, s')$ holds if there is an execution of program σ, possibly interleaved with other actions by the agent herself, that starts in situation s and ends in s', which we define as:[3]

$$\text{DoAL}(\sigma, s, s') \doteq \text{Do}(\sigma \| (\pi a.\ \text{Agent}(a) = agt?; a)^*, s, s').$$

However, a new problem with this approach is that it allows the agent to procrastinate in the execution of the intended plans in $\mathcal{D}$. For instance, suppose that the agent has the p-goal at priority level n_1 to execute the program σ_1 and at level n_2 to execute σ_2 next. Then, according to our definition of DoAL, the agent has the intention at level n_1 to execute σ_1 and at level n_2 to execute σ_2, possibly concurrently with other actions next, since we use DoAL to specify those goals. The "other actions" at level n_1 (n_2, respectively) are meant to be actions from the plan σ_2 (σ_1, respectively). However, nothing requires that the additional actions that the agent might execute are indeed from σ_2(σ_1, respectively), and thus this allows her to perform actions that are unnecessary as long as they do not perturb the execution of σ_1 and σ_2.

To deal with this, we include an additional component, a *procedural intention-base* Γ, to an SR-APL agent. Γ is a list of plans that the agent is currently actively pursuing. To avoid procrastination, we will require that any action that the agent actually performs

[3] We will use this construct to specify the procedural goals of an agent agt. Note that, while our theory supports exogenous actions performed by other agents, we assume that all actions in the plans of agt that specify her behavior must be performed by agt herself.

comes from Γ (as specified in the transition rule A_{step} below). In the following, we will use $\Gamma^{\|}$ to denote the concurrent composition of the programs in Γ:[4]

$$\Gamma^{\|} \doteq \textbf{if } (\Gamma = [\text{nil}]) \textbf{ then } \text{nil } \textbf{else } \text{First}(\Gamma) \| (\text{Rest}(\Gamma))^{\|}.$$

In SR-APL, a *program configuration* $\langle \sigma, s \rangle$ is a tuple consisting of a program σ and a ground situation s. An *agent configuration* on the other hand is a tuple $\langle \Gamma, s \rangle$ that consists of a list of plans Γ and a ground situation s. The initial agent configuration is $\langle [\text{nil}], S_0 \rangle$. Although strictly speaking an agent configuration includes the knowledge and the goals of the agent, these can be obtained from the (fixed) theory $\mathcal{D}$ and the situation in the configuration.

The semantics of SR-APL are defined by a two-tier transition system. *Program-level transition rules* specify how a program written in our plan language may evolve. On top of this, we use *agent-level transition rules* to specify how an SR-APL agent may evolve. Our program-level transition rules are simply a subset of the ConGolog transition rules. We use $\langle \sigma, s \rangle \rightarrow \langle \sigma', s' \rangle$ as an abbreviation for $\text{Trans}(\sigma, s, \sigma', s')$.

Agent-Level Transition Rules These transition rules are given in Table 1 and are similar to those of a typical BDI APL.[5] First of all, we have a rule A_{sel} for *selecting and adopting a plan* using the plan library Π for some realistic p-goal $\Diamond\Phi$. It states that if: (a) there is a rule in the plan library Π which says that the agent should adopt an instance of the plan σ if she has $\Diamond\Phi$ as her p-goal and knows that some instance of Ψ, (b) $\Diamond\Phi$ is a realistic p-goal with priority n in s for which the agent hasn't yet adopted any subgoal, (c) the agent knows in s that Ψ', (d) θ unifies Ψ and Ψ', and (e) the agent does not intend not to adopt $\text{DoAL}(\sigma\theta)$ w.r.t. $\Diamond\Phi$ next, then she can adopt the plan $\sigma\theta$, adding $\text{DoAL}(\sigma\theta)$ as a subgoal of $\Diamond\Phi$ to her goals in the theory $\mathcal{D}$, and adding $\sigma\theta$ to Γ (here $\text{Handled}(\phi, s)$ is defined as $\exists\psi.\ \text{SubGoal}(\psi, \phi, s)$).

We can show that if an agent does not have the c-goal in s not to adopt a subgoal ψ w.r.t. a supergoal ϕ, then she does not have the c-goal that $\neg\psi$ next in s, i.e.:

Theorem 1.

$$\mathcal{D} \models \neg\text{CGoal}(\neg\exists s'.\ \text{Do}(adoptRT(\psi, \phi), now, s'), s) \supset \\ \neg\text{CGoal}(\neg\exists s', p'.\ \text{Starts}(s') \wedge \text{Suffix}(p', do(adoptRT(\psi, \phi), s')) \wedge \psi(p'), s).$$

Theorem 1 and condition (e) above imply that the agent does not have the c-goal not to execute $\sigma\theta$ concurrently with $\Gamma^{\|}$ and possibly other actions next, i.e.:

$$(i).\ \neg\text{CGoal}(\neg\exists s', s''.\ \text{Do}(adoptRT(\text{DoAL}(\sigma\theta), \Diamond\Phi), now, s') \\ \wedge\ \text{DoAL}(\sigma\theta \parallel \Gamma^{\|}, s', s''), s).$$

[4] We will use various standard list operations, e.g. First (representing the first item of a list), Rest (representing the sublist that contains all but the first item of a list), Cons (for constructing a new list from an item and a list), Member (for checking membership of an item within a list), Remove (for removing a given item from a list), Replace (for replacing a given item with another item in a list), etc.

[5] We use $\text{CGoal}(\exists s'.\ \text{DoAL}(\sigma, now, s'), s)$ or simply $\text{CGoal}(\text{DoAL}(\sigma), s)$ as a shorthand for $\text{CGoal}(\exists s'.\ \text{Starts}(now) \wedge \text{OnPath}(s') \wedge \text{DoAL}(\sigma, now, s'), s)$.

Table 1. Agent Transition Rules

(A_{sel})	$\dfrac{\begin{array}{c}\text{Member}(\Diamond\Phi : \Psi \leftarrow \sigma, \Pi),\ \ \mathcal{D} \models \text{RPGoal}(\Diamond\Phi, n, s),\\ \mathcal{D} \models \neg\text{Handled}(\Diamond\Phi, s) \wedge \text{Know}(\Psi', s),\ \ \text{mgu}(\Psi, \Psi') = \theta,\\ \mathcal{D} \models \neg\text{CGoal}(\neg\exists s'.\ \text{Do}(adoptRT(\text{DoAL}(\sigma\theta), \Diamond\Phi), now, s'), s)\end{array}}{\langle \Gamma, s\rangle \Rightarrow \langle \text{Cons}(\sigma\theta, \Gamma), do(adoptRT(\text{DoAL}(\sigma\theta), \Diamond\Phi), s)\rangle}$
(A_{step})	$\dfrac{\begin{array}{c}\text{Member}(\sigma, \Gamma),\ \ \mathcal{D} \models \text{RPGoal}(\text{DoAL}(\sigma), n, s),\\ \mathcal{D} \models \langle \sigma, s\rangle \rightarrow \langle \sigma', do(a, s)\rangle \wedge \neg\text{CGoal}(\neg\exists s'.\ \text{Do}(a, now, s'), s)\end{array}}{\langle \Gamma, s\rangle \Rightarrow \langle \text{Replace}(\sigma, \sigma', \Gamma), do(a, s)\rangle}$
(A_{exo})	$\dfrac{\mathcal{D} \models \text{Exo}(a) \wedge \text{Poss}(a, s)}{\langle \Gamma, s\rangle \Rightarrow \langle \Gamma, do(a, s)\rangle}$
(A_{clean})	$\dfrac{\text{Member}(\sigma, \Gamma),\ \ \mathcal{D} \models \neg\exists n.\ \text{RPGoal}(\text{DoAL}(\sigma), n, s)}{\langle \Gamma, s\rangle \Rightarrow \langle \text{Remove}(\sigma, \Gamma), s\rangle}$
(A_{rep})	$\dfrac{\begin{array}{c}\mathcal{D} \models \neg\exists s'.\ \langle \Gamma^{\|}, s\rangle \rightarrow \langle \Gamma', s'\rangle,\ \ \mathcal{D} \models \neg\text{Final}(\Gamma^{\|}, s),\\ \text{For all } \sigma \text{ s.t. Member}(\sigma, \Gamma) \text{ we have:}\\ \mathcal{D} \models \exists n.\ \text{RPGoal}(\text{DoAL}(\sigma), n, s) \wedge \text{Handled}(\text{DoAL}(\sigma), s),\\ \mathcal{D} \models \neg\text{CGoal}(\neg\exists s'.\ \text{Do}(adopt(\text{Do}(\vec{a}), NPGoals(s)), now, s'), s),\\ \mathcal{D} \models \text{Agent}(\vec{a}) = agt \wedge \text{Do}(\vec{a}, s, s') \wedge \langle \Gamma^{\|}, s'\rangle \rightarrow \langle \Gamma', s''\rangle\end{array}}{\langle \Gamma, s\rangle \Rightarrow \langle \text{Cons}(\vec{a}, \Gamma), do(adopt(\text{Do}(\vec{a}), NPGoals(s)), s)\rangle}$

Moreover, it can be shown that in our framework, an agent acquires the c-goal that ψ after she adopts it as a subgoal of ϕ in s, provided that she has the realistic goal at some level n in s that ϕ, and that she does not have the c-goal in s that $\neg\psi$ next, i.e.:

Theorem 2.

$$\begin{array}{l}\mathcal{D} \models \exists n.\ \text{RPGoal}(\phi, n, s) \wedge\\ \quad \neg\text{CGoal}(\neg\exists s', p'.\ \text{Starts}(s') \wedge \text{Suffix}(p', do(adoptRT(\psi, \phi), s')) \wedge \psi(p'), s)\\ \quad \supset \text{CGoal}(\psi, do(adoptRT(\psi, \phi), s)).\end{array}$$

From (b), (i), and Theorem 2, we have that:

(ii). $\text{CGoal}(\exists s'.\ \text{DoAL}(\sigma\theta \parallel \Gamma^{\|}, now, s'), do(adoptRT(\text{DoAL}(\sigma\theta), \Diamond\Phi), s))$.

(i) ensures that the adopted subgoal $\sigma\theta$ is consistent with $\Gamma^{\|}$ in the sense that they can be executed concurrently, possibly along with other actions in s. (ii) confirms that $\sigma\theta$ is indeed intended after the $adoptRT$ action has happened. Note that this notion of consistency is a weak one, since it does not guarantee that there is an execution of the program $(\sigma\theta \parallel \Gamma^{\|})$ after the $adoptRT$ action happens, but rather ensures that the program $\text{DoAL}(\sigma\theta \parallel \Gamma^{\|})$ is executable. In other words, $\sigma\theta$ and the programs in Γ *alone* might not be concurrently executable, and additional actions might be required. We'll come back to this issue later.

Secondly, we have a transition rule A_{step} for single stepping the agent program by *executing an intended action* from Γ. It says that if: (a) a program σ in Γ can make

a program-level transition in s by performing a primitive action a with program σ' remaining in $do(a, s)$ afterwards, (b) DoAL(σ) is a realistic p-goal with priority n in s, and (c) the transition is consistent with the agent's goals in the sense that she does not have the c-goal not to execute a in s, then the agent can execute a, and Γ and s can be updated accordingly.

Once again we have a weak consistency requirement in condition (c) above. Ideally, we would have added to (c) that the agent can continue from $do(a, s)$ in the sense that she does not have the c-goal not to execute the remaining program σ' concurrently with the other programs in Γ in $do(a, s)$, i.e. that $\mathcal{D} \models \neg$CGoal$(\neg\exists s'.\ \text{Do}(a; (\sigma' \parallel \Gamma^{\parallel}), now, s'), s)$. However, note that Γ may not be complete in the sense that it may include plans that have actions that trigger the adoption of subgoals, for which the execution of $\Gamma^{\parallel}$ waits; but Γ does not have any adopted plans yet that can achieve these subgoals. Thus $\Gamma^{\parallel}$ by itself might currently have no complete execution, and will only become completely executable when all such subgoals have been fully expanded.

For example, consider a new agent for our blocks world domain who has a goal to eventually build a 3 blocks tower, i.e. $\Diamond$3Tower, where 3Tower $\doteq \exists b, b', b''.\ \text{OnTbl}(b) \wedge \text{On}(b', b) \wedge \text{On}(b'', b')$. Also, in addition to the above rules, her plan library Π includes the following rule:

$$\begin{aligned}&\Diamond\text{3Tower} : [\text{OnTbl}(b) \wedge \text{OnTbl}(b') \wedge \text{OnTbl}(b'') \wedge b \neq b' \wedge \\ &\qquad\qquad \text{Clear}(b) \wedge \text{Clear}(b') \wedge \text{Clear}(b'') \wedge \neg\text{Y}(b) \wedge \text{G}(b') \wedge \text{Y}(b'')] \leftarrow \sigma_1, \\ &\text{where}\ \ \sigma_1 = adoptRT(\Diamond\text{Twr}^G_{\bar{Y}}, \text{DoAL}(\sigma_2)); \sigma_2,\ \text{and}\ \ \sigma_2 = \text{Twr}^G_{\bar{Y}}?; stack(b'', b').\end{aligned}$$

This says that, if the agent knows about a non-yellow block b, a distinct green block b', and a yellow block b'' that are all clear and on the table, then her goal of building a 3 blocks tower can be fulfilled by adopting the plan that involves adopting the subgoal to eventually build a green non-yellow tower, waiting for the achievement of this subgoal, and then stacking b'' on b'. Suppose that in response to $\Diamond$3Tower, the agent adopted σ_1 as above as a subgoal of this goal using the A_{sel} rule, and thus σ_1 is added to Γ. In the next few steps, she will step through the adopted plan σ_1, executing one action at a time in an attempt to achieve her goal that $\Diamond$3Tower.

Note that, in SR-APL, the hierarchical decomposition of a subgoal, e.g. σ_1 above, is a two step process. In the first step, in response to the execution (via A_{step}) of the $adoptRT(\Diamond\text{Twr}^G_{\bar{Y}}, \text{DoAL}(\sigma_2))$ action in her plan σ_1 in Γ, the agent adopts $\Diamond\text{Twr}^G_{\bar{Y}}$ as a subgoal of executing the remaining program σ_2, possibly along with other actions, i.e. w.r.t. DoAL(σ_2). Then in the second step, she uses the A_{sel} rule to select and adopt a plan for the subgoal $\Diamond\text{Twr}^G_{\bar{Y}}$. We assume that the subgoal $\Diamond\text{Twr}^G_{\bar{Y}}$ must always be achieved before the supergoal. To do this, we suspend the execution of the supergoal by waiting for the achievement of the subgoal. This can be specified by the programmer by having the supergoal σ_2 start with the wait action $\text{Twr}^G_{\bar{Y}}?$ that waits for the subgoal to complete. But this means that σ_2 (and thus σ_1) by itself, i.e. without the DoAL construct, might not have a complete execution as it might get blocked when it reaches $\text{Twr}^G_{\bar{Y}}?$. Moreover, since σ_2 is a member of Γ, $\Gamma^{\parallel}$ will have a complete execution only when all the subgoals in Γ have been fully expanded. To deal with this, we use a weak consistency check that does not perform full lookahead over $\Gamma^{\parallel}$. However, our semantics ensures that any action a performed by the agent must not make the concurrent

execution of all the adopted plans of the agent possibly with other actions impossible, i.e. it must be consistent with DoAL($\Gamma^{\|}$), since A_{step} requires that doing a must be consistent with all her DoAL procedural goals (and other concurrent declarative goals) in her goal hierarchy, i.e. that $\mathcal{D} \models \neg\mathrm{CGoal}(\neg\exists s'.\ \mathrm{Do}(a, now, s'), s)$.

Thirdly, we have a rule A_{exo} for *accommodating exogenous actions*, i.e. actions occurring in the agent's environment that are not under her control. When such an action a occurs in s, the agent must update her p-goals by progressing the situation component of her configuration to $do(a, s)$.

Fourthly, we use the A_{clean} rule for *dropping adopted plans from the procedural goal-base Γ that are no longer intended in the theory $\mathcal{D}$*. It says that if there is a program σ in Γ, and executing σ possibly along with other actions is no longer a realistic p-goal, then σ should be dropped from Γ. This might be required when the occurrence of an exogenous action forces the agent to drop a plan by making it impossible to execute or inconsistent with her higher priority realistic p-goals. Recall that our theory automatically drops such plans from the agent's goal-hierarchy specified by $\mathcal{D}$.

Finally, we have a rule A_{rep} for *repairing an agent's plans in case she gets stuck*, i.e. when for all programs σ in Γ, the agent has the realistic p-goal that DoAL(σ) at some level n (and thus all of these DoAL(σ) are still individually executable and collectively consistent), but together they are not concurrently executable without some non-σ actions in the sense that $\Gamma^{\|}$ has no program-level transition in s. This could happen as a result of an exogenous action or as a side effect of our weak consistency check, as discussed below. The A_{rep} rule says that if: (a) $\Gamma^{\|}$ does not have a program level transition in s (which ensures that A_{step} can't be applied), (b) $\Gamma^{\|}$ is not considered to be completed in s, (c) every program in Γ is currently a realistic p-goal that has been handled (which ensures that A_{clean} and A_{sel} can't be applied), (d) there is a sequence of actions $\overrightarrow{a}$ that the agent does not intend not to execute next, and (e) $\overrightarrow{a}$ repairs Γ in the sense that there is a program level transition of $\Gamma^{\|}$ after $\overrightarrow{a}$ has been executed in s, then in an attempt to repair Γ, the agent should adopt $\overrightarrow{a}$ at the lowest priority level (i.e. at $NPGoals(s)$).

Why do we need this rule? One reason is because the agent could get stuck due to the occurrence of an exogenous action e, e.g. when e makes the preconditions of a plan σ in Γ false; note that, DoAL(σ) might still be executable after the occurrence of e, e.g. if there is an action r (encoded by the DoAL construct) that can be used to restore the preconditions of σ.

Another reason repair may be needed is that we use partial lookahead when executing actions via A_{step}. For example, assume a domain with actions a, b, and r, all of which are initially possible. The execution of b makes the preconditions of a false, while that of r restores them. Our agent has two adopted plans, DoAL(a) and DoAL(b) in the theory $\mathcal{D}$, and $\Gamma = [a, b]$. Note that $b; a$ is not a valid execution of $\Gamma^{\|}$, since the execution of b breaks the preconditions of a. But $b; r; a$ is indeed a valid execution of (DoAL(a) $\wedge$ DoAL(b)). Since we only do partial consistency checking, our semantics allows the agent to perform b as the first action.[6] That is, to execute b using the A_{step} transition rule, we only need to ensure that b has a program-level transition in s

[6] Note that this does not mean that A_{step} allows the agent to perform an action that makes one of her goals impossible, e.g. to execute b when such a repair action r is not available.

and that this transition is consistent with the agent's goals in $\mathcal{D}$, i.e. with DoAL(a) and DoAL(b), both of which hold. After the execution of b, the agent will get stuck, as there is no action in the progression of Γ that she can perform. To deal with this, we include the repair rule that makes the agent plan for and commit to a sequence of actions that can be used to repair Γ, which for our example is r. Note that, we could have avoided the need for repairing plans in this case by strengthening the conditions of the A$_{step}$ rule to do full lookahead by expanding all subgoals in Γ. However, this requires modeling the plan selection/goal decomposition process as part of the consistency check, which we leave for future work. We could have also relied on plan failure recovery techniques [28]. Finally, our repair rule does a form of conformant planning; more sophisticated forms of planning such as synthesizing conditional plans that include sensing actions could also be performed.

When the agent has complete information, there must be a repair plan available to the agent (whose actions can be performed by the agent herself) if her goals are consistent. In our framework, since the SSA for G drops all inconsistent goals/plans, the agent's p-goals are always consistent, and thus if complete information is assumed, it is always possible to repair the remaining plans. Consider our previous example: if the agent has DoAL(a) and DoAL(b) as her realistic p-goals, $\Gamma = [a, b]$, and if she has the c-goal not to execute an action from $\Gamma^{\|}$ (i.e. CGoal$(\neg\exists s'.\ \langle \Gamma^{\|}, now\rangle \rightarrow \langle \Gamma', s'\rangle, s)$), then it must be the case that she does not have the c-goal not to execute $\Gamma^{\|}$ along with other actions (e.g. r), i.e. $\neg$CGoal$(\neg\exists s'.$ DoAL$(a\|b, now, s'), s)$. Otherwise, one of DoAL(a) or DoAL(b) would have been dropped by the SSA for G as an agent's p-goals are always consistent with each other. Thus there must be a plan $\vec{a}$ that can repair Γ. Since the agent has complete information, this plan must work in all her epistemic alternatives (our repair rule does a form of conformant planning). Also, since by definition, the agent of the "other actions" in the DoAL construct is the agent herself, this means that she is also the agent of $\vec{a}$. If on the other hand the agent has only incomplete information, then a repair plan may need to perform sensing actions and branch on the results. We leave this kind of conditional planning for future work.

Also, note that this rule allows the agent to procrastinate in the sense that in addition to the plan that actually repairs Γ, she is allowed to adopt and execute actions that are unnecessary. This could be avoided by constraining the repair plan $\vec{a}$, e.g. by requiring it to be the shortest or the least costly plan etc. We leave this for future work.

In our operational semantics, we want to ensure that the procedural goals in Γ are consistent with those in the theory $\mathcal{D}$ before expansion of a subgoal/execution of an action occurs; so we assume that the A$_{clean}$ rule has higher priority than A$_{sel}$ and A$_{step}$. We can do this by adding appropriate preconditions to the antecedent of the latter, which we leave out for brevity.

To summarize, in SR-APL we formalize both declarative goals and plans uniformly in the same goal hierarchy specified by $\mathcal{D}$. We maintain the consistency of adopted declarative and procedural goals by ensuring that there is at least one path known to the agent over which all of her adopted declarative goals hold and that encodes the concurrent execution of all of her adopted plans, possibly along with other actions. Whenever the agent's goals/plans become inconsistent due to some external interference, the successor-state axiom in $\mathcal{D}$ will drop some of the adopted goals/plans, respecting their

priority, and consistency of the goal-base is automatically restored. We also have a procedural goal-base Γ containing the adopted plans in $\mathcal{D}$, whose sole purpose is to ensure that the agent does not procrastinate w.r.t. her adopted plans. The set of transition rules of SR-APL allows an SR-APL agent to select, adopt, and execute plans from the plan library and thus serves as SR-APL's practical reasoning component. While adopting plans and executing actions, we use a weak consistency check, and thus avoid searching over the entire plan-space while ensuring consistency. SR-APL also includes a repair rule that can be used to repair plans if the agent gets stuck (a) as a result of our weak consistency check (and lack of lookahead in plan selection), (b) due to external interferences, or (c) due to the existence of an adopted declarative goal for which there is no plan specified in the plan library.

Let us now define some useful notions of program execution in SR-APL. A *labeled execution trace $\mathcal{T}$ relative to a theory $\mathcal{D}$* is a (possibly infinite) sequence of configurations $\langle \Gamma_0, s_0\rangle \overset{l_0}{\Rightarrow} \langle \Gamma_1, s_1\rangle \overset{l_1}{\Rightarrow} \langle \Gamma_2, s_2\rangle \overset{l_2}{\Rightarrow} \langle \Gamma_3, s_3\rangle \overset{l_3}{\Rightarrow} \cdots$, s.t. $\Gamma_0 = [\text{nil}]$, $s_0 = S_0$ is the actual initial configuration, and for all $\langle \Gamma_i, s_i\rangle$, the agent level transition rule l_i can be used to obtain $\langle \Gamma_{i+1}, s_{i+1}\rangle$. Here l_i is one of A_{sel}, A_{step}, A_{exo}, A_{clean}, and A_{rep}, and in the absence of exogenous actions, l_i can be one of A_{sel}, A_{step}, A_{clean}, and A_{rep}. We sometimes suppress these labels. A *complete trace $\mathcal{T}$ relative to a theory $\mathcal{D}$* is a finite labeled execution trace relative to $\mathcal{D}$, $\langle \Gamma_0, s_0\rangle \overset{l_0}{\Rightarrow} \cdots \overset{l_{n-1}}{\Rightarrow} \langle \Gamma_n, s_n\rangle$, s.t. $\langle \Gamma_n, s_n\rangle$ does not have an agent level transition, i.e. $\langle \Gamma_n, s_n\rangle \not\Rightarrow$.

For our blocks world example, we can show that our SR-APL agent for this domain will not adopt inconsistent plans as seen in Section 2 and will in fact achieve all her goals. Note that, when arbitrary exogenous actions can occur, even the best laid plans can fail. Here we only consider the case of where exogenous actions are absent. We model this using the following axiom, which we call $NoExo$: $\forall a.\ \neg exo(a)$. Given this, we can show that:

Proposition 1 (*a*). *There exists a complete trace $\mathcal{T}$ relative to $\mathcal{D}_{BW} \cup \{NoExo\}$ for our blocks world program.* (*b*). *For all such complete traces $\mathcal{T} = \langle \Gamma_0, s_0\rangle \Rightarrow \langle \Gamma_1, s_1\rangle \Rightarrow \cdots \Rightarrow \langle \Gamma_n, s_n\rangle$, we have: $\mathcal{D}_{BW} \cup \{NoExo\} \models \text{Final}(\Gamma_n^{\|}, s_n) \wedge \text{Twr}_Y^G(s_n) \wedge \text{Twr}_R^B(s_n)$.* (*c*). *There are no infinite traces relative to $\mathcal{D}_{BW} \cup \{NoExo\}$.*

Thus when exogenous actions cannot occur, any execution of our SR-APL blocks world agent achieves all her goals.

6 Rationality of SR-APL Agents

We next prove some rationality properties that are satisfied by SR-APL agents. We only consider the case when exogenous actions do not occur. We could have considered exogenous actions, but in that case we would have to complicate the framework further, e.g. by assuming a fair environment that gives a chance to the agent to perform actions. Moreover, it is not obvious what rational behavior means in such contexts.

First of all, in each situation, for all domains $\mathcal{D}$ that are part of an SR-APL agent, the knowledge and c-goals/intentions as specified by $\mathcal{D}$ must be consistent:[7]

[7] This follows independently from the underlying agent theory.

Theorem 3 (Consistency of Knowledge and CGoals).

$$\mathcal{D} \models \forall s.\ \neg\text{Know}(false, s) \wedge \neg\text{CGoal}(false, s).$$

We can also show that the procedural goals in Γ and the declarative and procedural goals in the theory $\mathcal{D} \cup \{NoExo\}$ remain consistent. Let's say that *the procedural goals in Γ are consistent with those in the theory $\mathcal{D}$ in situation s in a configuration* $\langle \Gamma, s\rangle$ iff for all σ s.t. Member(σ, Γ), we have $\mathcal{D} \models$ CGoal(DoAL$(\sigma), s)$. Also, define $\mathcal{D}_{\bar{Exo}} \doteq \mathcal{D} \cup \{NoExo\}$. We have that:

Theorem 4 (Consistency of Γ and $\mathcal{D}_{\bar{Exo}}$). *If $\mathcal{T} = \langle \Gamma_0, s_0\rangle \Rightarrow \langle \Gamma_1, s_1\rangle \Rightarrow \cdots \Rightarrow \langle \Gamma_n, s_n\rangle$ is a complete trace of an SR-APL agent w.r.t. a theory $\mathcal{D}_{\bar{Exo}}$, then for all i s.t. $0 \leq i < n$, we have:*
(a). If $s_{i+1} = do(a, s_i)$ for some a, then the procedural goals in Γ_i are consistent with those in the theory $\mathcal{D}_{\bar{Exo}}$ in s_i,
(b). If $s_i = s_{i+1}$, then there exists j s.t. $0 < i < j \leq n$ and the goals in Γ_j are consistent with those in the theory $\mathcal{D}_{\bar{Exo}}$ in s_j,
(c). The procedural goals in Γ_n are consistent with those in the theory $\mathcal{D}_{\bar{Exo}}$ in s_n.

(a) and (c) are self-explanatory. (b) shows that whenever there is some procedural goal in Γ_i that is not a goal w.r.t. the theory $\mathcal{D}_{\bar{Exo}}$, the A_{clean} rule will remove it from Γ_i, and eventually consistency is restored.[8] It follows from Theorem 4 that in all configurations $\langle \Gamma, s\rangle$ where the plans in Γ are consistent with those in the theory $\mathcal{D}_{\bar{Exo}}$ in s, the agent intends to execute the programs in Γ concurrently starting in s, possibly with other actions, i.e. $\mathcal{D}_{\bar{Exo}} \models$ CGoal$(\exists s'.$ DoAL$(\Gamma^{\|}, now, s'), s)$.

Finally, our agents evolve in a rational way:

Theorem 5 (Rationality of Actions in a Trace). *If $\mathcal{T} = \langle \Gamma_0, s_0\rangle \stackrel{l_0}{\Rightarrow} \langle \Gamma_1, s_1\rangle \stackrel{l_1}{\Rightarrow} \cdots \stackrel{l_{n-1}}{\Rightarrow} \langle \Gamma_n, s_n\rangle$ is a trace of an SR-APL agent relative to a theory $\mathcal{D}_{\bar{Exo}}$, then for all i s.t. $0 < i \leq n$ and $s_i = do(a, s_{i-1})$, we have:*

$(a).\ \mathcal{D}_{\bar{Exo}} \models \neg$CGoal$(\neg\exists s'.$ Do$(a, now, s'), s_{i-1})$.
$(b).$ *If* $l_{i-1} = A_{step}$ *then* $\mathcal{D}_{\bar{Exo}} \models$ CGoal$(\exists s'.$ DoAL$(a, now, s'), s_{i-1})$.
$(c).\ \mathcal{D}_{\bar{Exo}} \models \forall \phi, \psi, n.\ a = adoptRT(\psi, \phi) \vee a = adopt(\psi, n) \supset$
$\neg$CGoal$(\neg\exists s', p'.$ Starts$(s') \wedge$ Suffix$(p', do(a, s')) \wedge \psi(p'), s_{i-1})$.

This states that SR-APL is sound in the sense that any trace produced by the APL semantics is consistent with the agent's chosen goals. To be precise, (a) if an SR-APL agent performs the action a in situation s_{i-1}, then it must be the case that she does not have the intention not to execute a next in s_{i-1}. Moreover, (b) if a is performed via A_{step}, then a is indeed intended in s_{i-1} in the sense that she has the intention to execute a possibly along with some other actions next. Finally, (c) if a is the action of adopting a subgoal ψ w.r.t. a supergoal ϕ or that of adopting a goal ψ at some level n, then the agent does not have the c-goal in s_{i-1} not to bring about ψ next.

[8] Recall that applications of A_{clean} do not change situations.

7 Discussion and Conclusion

Based on a "committed agent" variant of our rich theory of prioritized goal/subgoal dynamics [13], we developed a specification of an APL framework that handles prioritized goals and maintains the consistency of adopted declarative and procedural goals. We also showed that an agent specified in this language satisfies some strong rationality properties. While doing this, we addressed some fundamental questions about rational agency. We model an agent's concurrent commitments by incorporating the DoAL construct in her adopted plans, which allows her to be open towards future commitments to plans, using a procedural goal-base Γ to prevent procrastination. We formalized a weak notion of consistency between goals and plans that does not require the agent to expand all adopted goals while checking for consistency.

While SR-APL agents rely on a user-specified plan library, they can achieve a goal even if such plans are not specified. Indeed the A_{rep} rule can be used as a first principles planner for goals that can be achieved using sequential plans. Thus, given a goal $\Diamond\Phi$, all the programmer needs to do to trigger the planner is to add a plan of the form $(\Diamond\Phi : true \leftarrow \Phi?)$ to the plan library Π. Since the program $\Phi?$ is neither executable nor final, it will eventually trigger the A_{rep} rule, which will make the agent adopt a sequence of actions to achieve Φ.

Here, we focused on developing an expressive agent programming framework that yields a rational/robust agent without worrying about tractability. Thus our framework is a specification and model of an ideal APL rather than a practical APL. In the future, we would like to investigate restricted versions of SR-APL that are practical, with an understanding of how they compromise rationality. We think this can be done. For instance if we assume a finite domain, then reasoning with the underlying theory should be decidable. We could adapt techniques from partial order planning such as summary information/causal links to support consistency maintenance. We could also simply find a global linear plan and cache it, using summary information to revise it when necessary. There are some controller synthesis techniques that can deal with temporally extended goals [18, 2].

Also, it would be desirable to study a version where the agent fully expands an abstract plan and checks its executability before adopting it. Finally, while our underlying agent theory supports arbitrary temporally extended goals, in SR-APL we only consider achievement goals. We would like to relax this in the future.

References

1. M. E. Bratman. *Intentions, Plans, and Practical Reason*. Harvard University Press, Cambridge, MA, USA, 1987.
2. D. Calvanese, G. De Giacomo, and M. Y. Vardi. Reasoning about Actions and Planning in LTL Action Theories. In *Proc. KR'02*, pages 593–602, 2002.
3. B. J. Clement and E. H. Durfee. Theory for Coordinating Concurrent Hierarchical Planning Agents Using Summary Information. In *Proc. AAAI'99*, pages 495–502, 1999.
4. B. J. Clement, E. H. Durfee, and A. C. Barrett. Abstract Reasoning for Planning and Coordination. *J. of Artificial Intelligence Research*, 28:453–515, 2007.
5. M. Dastani. 2APL: A Practical Agent Programming Language. *J. of AAMAS*, 16(3):214–248, 2008.

6. G. De Giacomo, Y. Lespérance, and H. J. Levesque. ConGolog, a Concurrent Programming Language Based on the Situation Calculus. *Artificial Intelligence*, 121:109–169, 2000.
7. K. V. Hindriks, F. S. de Boer, W. van der Hoek, and J.-J. Ch. Meyer. Agent Programming with Declarative Goals. In *Intelligent Agents VII : Agent Theories, Architecture, and Languages*, vol. 1986 of LNAI, pages 228–243. Springer-Verlag, 2000.
8. K. V. Hindriks, W. van der Hoek, and M. B. van Riemsdijk. Agent Programming with Temporally Extended Goals. In *Proc. AAMAS'09*, pages 137–144, 2009.
9. J. F. Horty and M. E. Pollack. Evaluating New Options in the Context of Existing Plans. *Artificial Intelligence*, 127:199–220, 2001.
10. F. F. Ingrand, M. P. Georgeff, and A. S. Rao. An Architecture for Real-Time Reasoning and System Control. *IEEE Expert*, 7(6):34–44, 1992.
11. S. M. Khan. *Rational Agents : Prioritized Goals, Goal Dynamics, and Agent Programming Languages with Declarative Goals (in preparation)*. Ph.D. thesis, York University, Canada, 2011.
12. S. M. Khan and Y. Lespérance. ECASL: A Model of Rational Agency for Communicating Agents. In *Proc. AAMAS'05*, pages 762–769, 2005.
13. S. M. Khan and Y. Lespérance. A Logical Framework for Prioritized Goal Change. In *Proc. AAMAS'10*, pages 283–290, 2010.
14. S. M. Khan and Y. Lespérance. Towards a Rational Agent Programming Language with Prioritized Goals. In *Working Notes of DALT VIII*, pages 18–33, 2010.
15. S. M. Khan and Y. Lespérance. Prioritized Goals and Subgoals in a Logical Account of Goal Change – A Preliminary Report. In *Proc. DALT VII*, vol. 5948 of LNAI, pages 119–136, Springer-Verlag, 2010.
16. S. M. Khan and Y. Lespérance. SR-APL: A Model for a Programming Language for Rational BDI Agents with Prioritized Goals (Extended Abstract). To appear in *Proc. AAMAS'11*, 2011.
17. H. J. Levesque, F. Pirri, and R. Reiter. Foundations for a Calculus of Situations. *Electronic Transactions of AI (ETAI)*, 2(3–4):159–178, 1998.
18. M. Pistore and P. Traverso. Planning as Model Checking for Extended Goals in Non-Deterministic Domains. In *Proc. IJCAI'01*, pages 479–484, 2001.
19. A. S. Rao. AgentSpeak(L): BDI Agents Speak Out in a Logical Computable Language. In *Agents Breaking Away*, vol. 1038 of LNAI, pages 42–55. Springer-Verlag, 1996.
20. R. Reiter. *Knowledge in Action. Logical Foundations for Specifying and Implementing Dynamical Systems*. MIT Press, 2001.
21. S. Sardiña, L. de Silva, and L. Padgham. Hierarchical Planning in BDI Agent Programming Languages: A Formal Approach. In *Proc. AAMAS'06*, pages 1001–1008, 2006.
22. S. Sardiña and L. Padgham. A BDI Agent Programming Language with Failure Recovery, Declarative Goals, and Planning. *J. of AAMAS* (to appear), 2010.
23. R. Scherl and H. J. Levesque. Knowledge, Action, and the Frame Problem. *Artificial Intelligence*, 144(1–2):1–39, 2003.
24. S. Shapiro and G. Brewka. Dynamic Interactions Between Goals and Beliefs. In *Proc. IJCAI'07*, pages 2625–2630, 2007.
25. S. Shapiro, Y. Lespérance, and H. J. Levesque. Goal Change in the Situation Calculus. *J. of Logic and Computation*, 17(5):983–1018, 2007.
26. J. Thangarajah, L. Padgham, and M. Winikoff. Detecting and Avoiding Interference between Goals in Intelligent Agents. In *Proc. IJCAI'03*, pages 721–726, 2003.
27. M. B. van Riemsdijk, M. Dastani, and J.-J. Ch. Meyer. Goals in Conflict: Semantic Foundations of Goals in Agent Programming. *J. of AAMAS*, 18(3):471–500, 2009.
28. M. Winikoff, L. Padgham, J. Harland, and J. Thangarajah. Declarative and Procedural Goals in Intelligent Agent Systems. In *Proc. KR'02*, pages 470–481, 2002.

A Coupled Operational Semantics for Goals and Commitments

Pankaj R. Telang[1,4], Neil Yorke-Smith[2,3], and Munindar P. Singh[4]

[1] Cisco Systems Inc., Research Triangle Park, NC 27709, USA. prtelang@ncsu.edu
[2] Olayan School of Business, American University of Beirut, Lebanon.
[3] SRI International, Menlo Park, CA 94025, USA. nysmith@aub.edu.lb
[4] North Carolina State University, Raleigh, NC 27695-8206, USA. singh@ncsu.edu

Abstract. Commitments capture how an agent relates with another agent, whereas (private) goals describe states of the world that an agent is motivated to bring about. Researchers have observed that goals and commitments are complementary, but have not yet developed a combined operational semantics for them. This paper does just that by relating their respective lifecycles as a basis for (1) how these concepts cohere for an individual agent and (2) engender cooperation among agents. We illustrate on a real-world scenario in the domain of aerospace aftermarket services. Our semantics yields important desirable properties, including convergence of the configurations of cooperating agents, thereby delineating some theoretically well-founded yet practical modes of cooperation in a multiagent system.

1 Introduction and Motivation

Whereas the study of goals is a long-standing theme in AI, the last several years have seen the motivation and elaboration of a theory of (social) commitments. The concepts of goals and commitments are intuitively complementary. A commitment describes how an agent relates with another agent, while a goal describes a state of the world that an agent is motivated to bring about. A commitment carries deontic force in terms of what an agent would bring about for another agent, while a goal describes an agent's proattitude toward some condition.

Researchers have begun tying these two concepts together. We go beyond existing works by developing a formal, modular approach that accomplishes the following. First, it characterizes the lifecycles and more generally the operational semantics of the two concepts. Second, it characterizes the interplay between goals and commitments. Third, this approach distinguishes the purely semantic aspects of their lifecycles from the pragmatic aspects of how a cooperative agent may reason. Fourth, it shows that certain desirable properties can be guaranteed for agents who respect selected rules of cooperation. These properties include *convergence*: the agents achieve a level of consistency internally (between the states of their goals and commitments) and externally (between the states of their commitments relevant to each other).

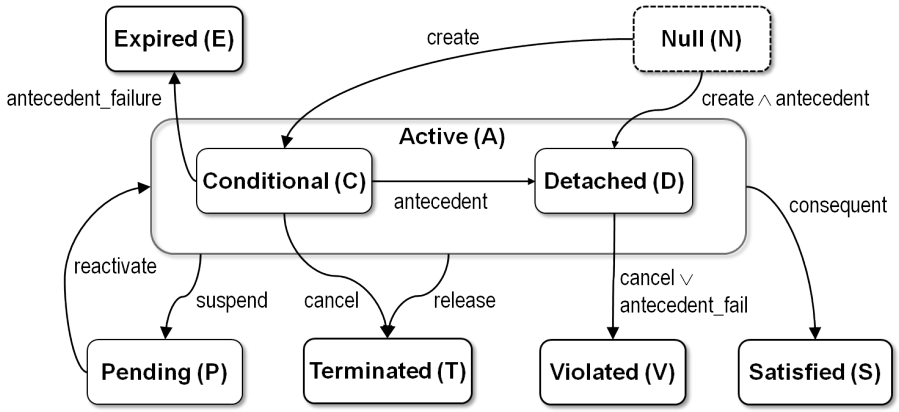


Fig. 1. Commitment lifecycle as a state transition diagram.

We begin in Sect. 2 by introducing the concepts of commitment and goals, and for each presenting their lifecycle as a state transition diagram. Sect. 3 presents our combined operational semantics, which is based on guarded rules. We distinguish between two types of rules: mandatory structural rules which reflect the lifecycle of goals and commitments, and practical rules that an agent may choose to follow in order to achieve certain desirable properties. In Sect. 4 we prove convergence properties for agents that adopt both types of rules. Sect. 5 illustrates on a real-world scenario, and Sect. 6 places our work in context.

2 Background: Commitments and Goals

Commitments. A *commitment* expresses a social relationship between two agents. Specifically, a commitment C(DEBTOR, CREDITOR, antecedent, consequent) denotes that the DEBTOR commits to the CREDITOR to bringing about the consequent if the antecedent begins to hold [10]. Fig. 1 shows the lifecycle of a commitment simplified from Telang and Singh [12] (below, we disregard timeouts, and commitment delegation or assignment). A labeled rectangle represents a commitment state, and a directed edge represents a transition, labeled with the corresponding action or event.

A commitment can be in one of the following states: Null (before it is created), Conditional (when it is initially created), Expired (when its antecedent remains forever false, while it was still Conditional), Satisfied (when its consequent is brought about while it was Active regardless of its antecedent), Violated (when its antecedent has been true but its consequent will forever be false, or it is canceled when Detached), Terminated (when canceled while Conditional or released while Active), or Pending (when suspended while Active). Active has two substates: Conditional (when its antecedent is false) and Detached (when its antecedent has held) A debtor may create, cancel, suspend, or reactivate a commitment; a creditor may release a debtor from a commitment.

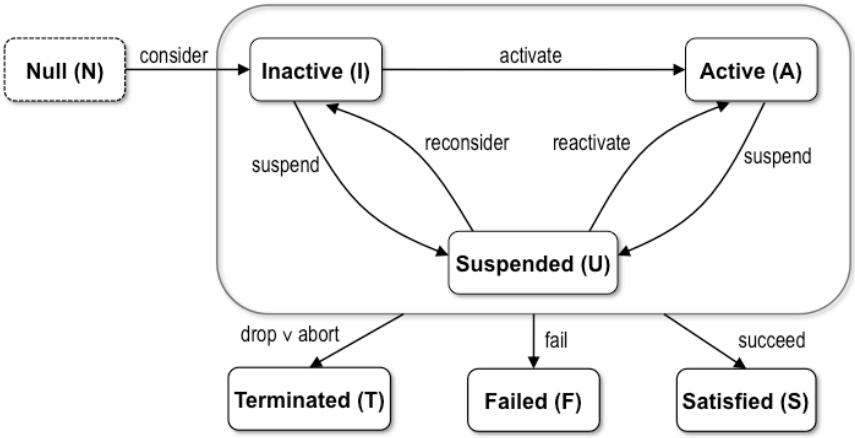


Fig. 2. Simplified lifecycle of an achievement goal as a state transition diagram.

Goals. An agent's desires represent a proattitudes on part of the agent; an agent may concurrently hold mutually inconsistent desires. By contrast, goals are at least consistent desires: we take a rational agent to believe that its goals are mutually consistent. An agent's intentions are adopted or activated goals.

A *goal* $G = \mathsf{G}(x, p, r, q, s, f)$ of an agent x has a *precondition* (or context) p that must be true before G can become Active and an intention can be adopted to achieve it, a *in-condition* r that is true once G is Active until its achievement, and a *post-condition* (or effect) q that becomes true if G is successfully achieved [16]. The *success condition* s defines the success of G, and the *failure condition* f defines its failure. A goal G is successful iff s becomes true prior to f: that is, the truth of s entails the satisfaction of G only if f does not intervene. Often, the post-condition q and the success condition s coincide, but they need not. As for commitments, the success or failure of a goal depends only on the truth or falsity of the various conditions, not on which agent brings them about.

Fig. 2 simplifies Thangarajah et al.'s [13] lifecycle of an achievement goal (we do not consider maintenance goals). A goal can be in one of the following states: Null, Inactive (renamed from Pending to avoid conflict with commitments), Active, Suspended, Satisfied, Terminated, or Failed. The last three collectively are *terminal states*: once a goal enters any of them, it stays there forever. The semantic rules will link the the definition of a goal G and its states.

Before its creation, a candidate goal is in state Null; once considered by an agent (its "goal holder"), it commences as Inactive, in contrast to commitments which are created in state Active. Upon activation, the goal becomes Active; the agent may pursue its satisfaction by attempting to achieve s. If s is achieved, the goal moves to Satisfied. At any point, if the failure condition of the goal becomes true, the goal moves to Failed. At any point, the goal may enter Suspended, from which it may eventually return to an Inactive or Active state. Lastly, at any point the agent may drop or abort the goal, thereby moving it to the Terminated state.

3 Proposed Operational Semantics

Whereas a goal is specific to an agent (but see Sect. 6), a commitment involves a pair of agents. On the one hand, an agent may create commitments towards other agents in order to achieve its goals. On the other hand, an agent may consider goals in order to fulfill its commitments to other agents.

Chopra et al. [3] formalize a semantic relationship between commitments and goals. They write goals in either or both of the antecedent or consequent of a commitment, i.e., $\mathsf{C}(x, y, g_1, g_2)$, where antecedent ($g_1$) and consequent ($g_2$) are objective conditions (success conditions of one or more goals), not goals. For example, a car insurer may commit to a repair garage to paying if the latter performs a repair: C(INSURER, REPAIRER, car_repaired, payment_made). Here, car_repaired is the success condition of the insurer's goal. The insurer may consider a goal with success condition of payment_made to satisfy the commitment.

3.1 Formal Semantics

We consider the *configuration* of an agent x as the tuple $S_x = \langle \mathcal{B}, \mathcal{G}, \mathcal{C} \rangle$ where $\mathcal{B}$ is its set of beliefs, $\mathcal{G}$ its set of goals, and $\mathcal{C}$ its set of commitments. Conceptually, an agent's configuration relates to both its cognitive and its social state: it incorporates its beliefs and goals as well as its commitments. Where necessary, we index sets or states by agent; for brevity, we omit the parts of the configuration that are clear. We adopt a standard propositional logic.

- $\mathcal{B}$ is the set of x's beliefs about the current snapshot of the world, and include beliefs about itself and other agents. Each snapshot is itself atemporal.
- $\mathcal{C}$ is a set of commitments, of the form $\mathsf{C}(x, y, s, u)$, where x and y are agents and s and u are logical conditions. We use a superscript from Fig. 1 to denote the state of a commitment
- $\mathcal{G}$ is a set of goals adopted by x, of the form $\mathsf{G}(x, p, r, q, s, f)$. $\mathcal{G}$ includes goals that are Inactive. Since the goals in G are adopted, we take it that they are mutually consistent [16]. Superscripts from Fig. 2 denote goal states.

We capture the operational semantics of reasoning about goals and commitments via guarded rules in which S_i are configurations:

$$\frac{guard}{S_1 \longrightarrow S_2} \tag{1}$$

$S_i.\mathcal{B}$, $S_i.\mathcal{G}$, and $S_i.\mathcal{C}$ are the appropriate components of S_i. $S_i \longrightarrow S_j$ is a transition. In most settings, we can specify a family of transitions as an action. For example, for a commitment C, $\mathsf{suspend}(C)$ refers to the set of transitions $S_i \longrightarrow S_j$ where $C \in S_i.\mathcal{C}$ and $\mathsf{suspend}(C) \in S_j.\mathcal{C}$. For actions a and b, $a \wedge b$ indicates that both must be performed.

The same guard may enable multiple transitions $S_i \longrightarrow S_j$ with the same S_i, indicating choice (of the agent involved). For example, intuitively, if a commitment corresponding to a goal expires, an agent could either (i) establish an

alternative commitment or (ii) drop the goal. The resulting rules have the same guards, but specify different transitions.

We assume that rational agents seek to achieve their Active goals. That is, an agent at least believes that it has some means to achieve the success condition s of a goal it intends. Either the agent can adopt a plan whose success will achieve s, or it can seek to persuade another agent to bring about the condition s.

3.2 Structural Rules

We distinguish between two types of rules. *Structural* rules specify the progression of a commitment or a goal per their respective lifecycles. Each action that an agent can perform on a goal or a commitment derives a rule of this form. The guard of such a rule is an objective fact. For example, if f holds, a goal whose failure condition is f would be considered as having Failed. Rules such as these capture the hard integrity requirements represented by the lifecycles of goals and commitments. In our particular setting, such rules are both complete and deterministic, in that there is exactly one target state for each potential transition. The state diagrams in Fig. 1 and 2 correspond to the structural rules. The rules are straightforward to derive; we write one rule out in full below, and omit the remainder for reasons of space. We also do not write the standard lifting rule that relates transitions on single commitment/goal to transitions on sets.

A conceptual relationship is established between a goal and a commitment when they reference each other's objective conditions. Even when related in such a manner, however, the goal and the commitment independently progress in accordance with their respective lifecycles. For example, consider a goal $G = \mathsf{G}(x, p, r, q, s, f)$ of agent x. To satisfy this goal, x may create a commitment $\mathsf{C}(x, y, s, u)$. That is, agent x may commit to agent y to bring about u if y brings about s. When y brings about s, C detaches, and G is satisfied. We describe the progression of x's configuration as a structural rule:

$$\frac{\mathcal{B}_x \models s}{\langle G^A, C^A \rangle \longrightarrow \langle G^S, C^D \rangle} \tag{2}$$

where the superscripts denote commitment and goal states from Figs. 1 and 2. Some rules apply in multiple states, indicated via superscripts such as $C^{E \vee T}$.

3.3 Practical Rules

Practical (reasoning) rules capture not necessary integrity requirements, but rather patterns of pragmatic reasoning that agents may or may not adopt under different circumstances. The guard of such a rule is usually the antecedent or consequent of a commitment or the success or failure condition of a goal. The outcome of such a rule can be expressed as an action or an event from the applicable lifecycle diagram, which effectively summarizes a family of transitions from configurations to configurations. For example, an agent having an Active goal may decide to create a commitment as an offer to another agent, in order

to persuade the second agent to help achieve its goal. Or, an agent may decide to create a goal to service a commitment.

Such practical rules may be neither complete nor deterministic, in that an agent may find itself at a loss as to how to proceed or may find itself with multiple options. Such nondeterminism corresponds naturally to a future-branching temporal model: each agent's multiplicity of options leads to many possible progressions of its configuration and of the configurations of its peers. The convergence results we show below indicate that our formulated set of rules are complete (i.e., sufficient) in a useful technical sense.

Note that the practical rules are merely options that an agent has available when it adopts these rules as patterns of reasoning—as illustrated in our earlier example of two possible agent actions when a commitment expires. An agent may refine on these rules to always select from among a narrower set of the available options, for example, through other reasoning about its preferences and utilities. Our approach supports such metareasoning capability in principle, but we defer a careful investigation of it to future research.

It is helpful to group the practical rules into two cases.

Case I: From Goals to Commitments. Here, an agent creates a commitment to satisfy its goal. Consider an agent x having a goal $G = \mathsf{G}(x, p, r, q, s, f)$, and a commitment from x to y: $C = \mathsf{C}(x, y, s, u)$. Notice that s occurs as the success condition of G and the antecedent of C. This case presumes that x lacks (or prefers not to exercise) the capability to bring about s, but can bring about u, and that y can bring about s. Thus x uses C as a means to achieve G (x's *end goal*). Agent x's (goal holder) practical reasoning rules are as follows.

Note that we do not assume that commitments are symmetric. That is, in general, an agent may have a commitment to another agent without the latter having a converse commitment to the former agent.

Recall that superscripts indicate the state of a goal or commitment; for a goal G, the Suspended state is indicated by G^{U}. The guard is a pattern-matching expression. For example, $\langle G^A \rangle$ matches all configurations in which G is Active, regardless of other goals and commitments.

– ENTICE: If G is active and C is null, x creates an offer (C) to another agent.

$$\frac{\langle G^A, C^N \rangle}{\mathsf{create}(C)} \text{ ENTICE} \tag{3}$$

Motivation: (Only) by creating the commitment can the agent satisfy its goal.

– SUSPEND OFFER: If G is suspended, then x suspends C.

$$\frac{\langle G^U, C^A \rangle}{\mathsf{suspend}(C)} \text{ SUSPEND OFFER} \tag{4}$$

Motivation: The agent may employ its resources in other tasks instead of working on the commitment.

– REVIVE: If G is active, and C is pending, then x reactivates C.

$$\frac{\langle G^A, C^P \rangle}{\mathsf{reactivate}(C)} \text{ REVIVE} \tag{5}$$

Motivation: An active commitment is needed by the agent to satisfy its goal.

– WITHDRAW OFFER: If G fails or is terminated, then x cancels C.

$$\frac{\langle G^{T \vee F}, C^A \rangle}{\mathsf{cancel}(C)} \text{ WITHDRAW OFFER} \tag{6}$$

Motivation: The commitment is of no utility once the end goal for which it is created no longer exists.

– REVIVE TO WITHDRAW: If G fails or is terminated and C is pending, then x reactivates C.

$$\frac{\langle G^{T \vee F}, C^P \rangle}{\mathsf{reactivate}(C)} \text{ REVIVE TO WITHDRAW} \tag{7}$$

Motivation: If the goal fails or is terminated, and the commitment is pending, then the agent reactivates the commitment, and later cancels the commitment by the virtue of WITHDRAW OFFER. As per the commitment lifecycle from Fig. 1, an agent needs to reactivate a commitment before cancelling it.

– NEGOTIATE: If C terminates or expires, and G is active or suspended, then x creates another commitment C' to satisfy its goal.

$$\frac{\langle G^{A \vee U}, C^{E \vee T} \rangle}{\mathsf{create}(C')} \text{ NEGOTIATE} \tag{8}$$

Motivation: The agent persists with its goal by trying alternative ways to induce other agents to cooperate.

– ABANDON END GOAL: If C terminates or expires, then x gives up on G.

$$\frac{\langle G^{A \vee U}, C^{E \vee T} \rangle}{\mathsf{drop}(G)} \text{ ABANDON END GOAL} \tag{9}$$

Motivation: The agent may decide no longer to persist with its end goal. Note that an agent may also employ a structural rule to drop a goal without any condition.

It is necessary only that the rules cover *possible* combinations of goal and commitment states. For example, the $\langle G^A, C^V \rangle$ state is not possible since C can violate only after G satisfies; hence no rule is required.

Case II: From Commitments to Goals Here, an agent creates a goal to bring about its part (consequent if debtor, antecedent if creditor) in a commitment.

Consider commitment $C = \mathsf{C}(x, y, s, u)$ and goals $G_1 = \mathsf{G}(x, p, r, q, u, f)$ and $G_2 = \mathsf{G}(y, p', r', q', s, f')$. The practical reasoning rules for agent x are as follows.

– DELIVER: If G_1 is null and C is detached, then x considers and activates goal G_1 to bring about C's consequent.

$$\frac{\langle G_1^N, C^D \rangle}{\mathsf{consider}(G_1) \wedge \mathsf{activate}(G_1)} \text{ DELIVER} \tag{10}$$

DELIVER$'$: If G_1 is inactive and C is detached, then x activates goal G_1 to bring about C's consequent.

$$\frac{\langle G_1^I, C^D \rangle}{\mathsf{activate}(G_1)} \text{ DELIVER'} \tag{11}$$

Motivation: The agent is honest in that it activates a goal that would lead to discharging its commitment.

– BACK BURNER: If G_1 is active and C is pending, then x suspends G_1.

$$\frac{\langle G_1^A, C^P \rangle}{\mathsf{suspend}(G_1)} \text{ BACK BURNER} \tag{12}$$

Motivation: By suspending the goal, the agent may employ its resources to work on other goals.

– FRONT BURNER: If G_1 is suspended and C is detached, then x reactivates G_1.

$$\frac{\langle G_1^U, C^D \rangle}{\mathsf{reactivate}(G_1)} \text{ FRONT BURNER} \tag{13}$$

Motivation: An active goal is necessary for the agent to bring about its part in the commitment.

– ABANDON MEANS GOAL: If G_1 is active and C terminates (y releases x from C) or violates (x cancels C), then x drops G_1.

$$\frac{\langle G_1^A, C^{T \vee V} \rangle}{\mathsf{drop}(G_1)} \text{ ABANDON MEANS GOAL} \tag{14}$$

Motivation: The goal is not needed since the commitment for which it is created no longer exists.

– PERSIST: If G_1 fails or terminates and C is detached, then x activates goal G_1' identical to G_1.

$$\frac{\langle G_1^{T \vee F}, C^D \rangle}{\mathsf{consider}(G_1') \wedge \mathsf{activate}(G_1')} \text{ PERSIST} \tag{15}$$

Motivation: The agent persists in pursuing its part in the commitment.

– GIVE UP: If G_1 fails or terminates and C is detached, then x cancels C.

$$\frac{\langle G_1^{T \vee F}, C^D \rangle}{\mathsf{cancel}(C)} \text{ GIVE UP} \tag{16}$$

Motivation: x gives up pursuing its commitment by cancelling or releasing it.

Many of the practical reasoning rules for agent y are similar to x's.

– DETACH: If G_2 is null and C is conditional, then y considers and activates goal G_2 to bring about C's antecedent.

$$\frac{\langle G_2^N, C^C \rangle}{\mathsf{consider}(G_2) \wedge \mathsf{activate}(G_2)} \text{ DETACH} \tag{17}$$

DETACH$'$: If G_2 is inactive and C is conditional, then y activates goal G_2 to bring about C's antecedent.

$$\frac{\langle G_2^I, C^C \rangle}{\mathsf{activate}(G_2)} \text{ DETACH'} \tag{18}$$

Motivation: The creditor brings about the antecedent hoping to influence the debtor to discharge the commitment.

– BACK BURNER: If G_2 is active and C is pending, then y suspends G_2.

$$\frac{\langle G_2^A, C^P \rangle}{\mathsf{suspend}(G_2)} \text{ BACK BURNER} \tag{19}$$

Motivation: By suspending the goal, the agent may employ its resources to work on other goals.

– FRONT BURNER: If G_2 is suspended and C is conditional, y reactivates G_2.

$$\frac{\langle G_2^U, C^C \rangle}{\mathsf{reactivate}(G_2)} \text{ FRONT BURNER} \tag{20}$$

Motivation: An active goal is necessary for the agent to bring about its part in the commitment.

– ABANDON MEANS GOAL: If G_2 is active and C expires or terminates (either x cancels, or y releases x from C), then y drops G_2.

$$\frac{\langle G_2^A, C^{E \vee T} \rangle}{\mathsf{drop}(G_2)} \text{ ABANDON MEANS GOAL} \tag{21}$$

Motivation: The goal is not needed since the commitment for which it is created no longer exists.

– PERSIST: If G_2 fails or terminates and C is conditional, then y activates goal G_2' identical to G_2.

$$\frac{\langle G_2^{T \vee F}, C^C \rangle}{\mathsf{consider}(G_2') \wedge \mathsf{activate}(G_2')} \text{ PERSIST} \tag{22}$$

Motivation: The agent persists in pursuing its part (either antecedent or consequent) in the commitment.

– GIVE UP: If G_2 fails or terminates and C is conditional, y releases x from C.

$$\frac{\langle G_2^{T \vee F}, C^C \rangle}{\mathsf{release}(C)} \text{ GIVE UP} \tag{23}$$

Motivation: y gives up pursuing its commitment by cancelling or releasing it.

4 Convergence Properties

We would like to be assured that a coherent world state will be reached, no matter how the agents decide to act, provided that they act according to the rules we have given. We show that the practical rules are sufficient for an agent to reach a coherent state. Informally, in a *coherent* state, corresponding goals and commitments align.

Definition 1 *Let* $G = \mathsf{G}(x, p, r, q, s, f)$ *be a goal and* $C = \mathsf{C}(x, y, s, u)$ *a commitment. Then we say that any configuration that satisfies* $\langle G^A, C^A \rangle$, $\langle G^U, C^P \rangle$, $\langle G^{T \vee F}, C^{E \vee T \vee V} \rangle$, *or* $\langle G^S, C^S \rangle$ *is a* coherent state *of* G *and* C.

We have rules that can recreate goals and commitments (namely, PERSIST and NEGOTIATE). These rules could cause endless cycles; therefore we introduce:

Definition 2 *A* progressive rule *is any practical rule other than* PERSIST *and* NEGOTIATE. *The latter two rules we call* nonprogressive.

Theorems 1 and 2 capture the intuition of coherence of a single agent's configuration. All possible agent executions eventually lead to one of the coherent states if the agent obeys our proposed practical rules. They relate to the situations of Case I and Case II respectively.

Theorem 1. *Suppose* $G = \mathsf{G}(x, p, r, q, s, f)$ *and* $C = \mathsf{C}(x, y, s, u)$. *Then there is a finite sequence of progressive rules interleaving finitely many occurrences of nonprogressive rules that leads to a coherent state of* G *and* C. □

Theorem 2. *Suppose* $C = \mathsf{C}(x, y, s, u)$ *and* $G = \mathsf{G}(x, p, r, q, u, f)$. *Then there is a finite sequence of progressive rules interleaving finitely many occurrences of nonprogressive rules that leads to a coherent state of* G *and* C. □

Theorem 3 applies to the configurations of two agents related by a commitment. If the agents obey our proposed practical rules, then the state of the debtor's means goal follows the state of the creditor's end goal.

Theorem 3. (Goal convergence across agents) *Suppose* $G_1 = \mathsf{G}(x, p_1, r_1, q_1, s, f_1)$ *and* $G_2 = \mathsf{G}(y, p_2, r_2, q_2, s, f_2)$ *are goals, and* $C = \mathsf{C}(x, y, s, u)$ *is a commitment. Then there is a finite sequence of rules drawn from the practical rules that leads to* G_2*'s state equaling* G_1*'s state.* □

5 Illustrative Application

We illustrate the value of integrated reasoning over commitments and goals with a real-world scenario, drawn from European Union CONTRACT project [14] in the domain of aerospace aftermarket services.

Fig. 3 shows a high-level process flow of aerospace aftermarket services. The participants are an airline operator, an aircraft engine manufacturer, and a parts manufacturer. The engine manufacturer provides engines to the airline operator,

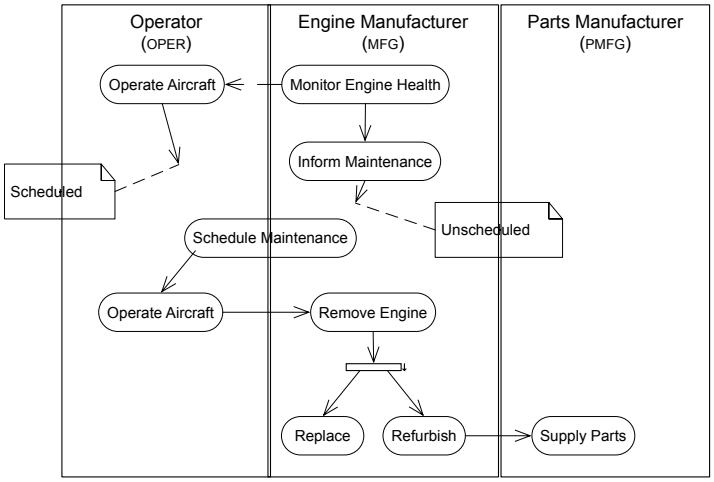


Fig. 3. A high-level model of the aerospace aftermarket process (verbatim from the Amoeba [5] paper, originally from CONTRACT project [14])

and additionally services the engines to keep them operational; in return, the operator pays the manufacturer. If a plane waits on the ground for an engine to be serviced, the manufacturer pays a penalty to the operator. As part of the agreement, the operator regularly provides engine health data to the manufacturer, and may proactively request the manufacturer to perform schedule engine maintenance. The manufacturer analyzes the health data and informs the operator of any required unscheduled engine maintenance. As part of servicing the engine, the manufacturer can either refurbish or replace it. The manufacturer maintains a supply of engines by procuring parts from a parts manufacturer.

Table 1 describes the goals and commitments that model this scenario. For reasons of space, we exclude the airline manufacturer purchasing parts from the parts manufacturer. In the table, service_promised proposition represents creation of C_3 and C_4, and health_reporting_promised represents creation of C_5.

Table 2 describes a possible progression of the aerospace scenario. Each step shows the structural or practical reasoning rule that the airline manufacturer (MFG) or the operator (OPER) employ, and how their configurations progress. For readability, we place new or modified state elements in bold, and omit satisfied commitments and goals in steps subsequent to their being satisfied.

In Steps 1 and 2, the airline manufacturer and the operator consider and activate goals G_1 and G_2. In Step 3, the manufacturer entices (ENTICE rule) the operator to create C_1, which would enable the manufacturer to satisfy G_1. Notice how ENTICE causes manufacturer's configuration to reach the coherent state $\langle\{G_1^A\},\{C_1^A\}\rangle$. Similarly in Step 4, operator creates C_2.

In Step 5, the manufacturer considers and activates G_4 to detach (DETACH rule) C_2. Observe how DETACH activates manufacturer's (debtor's) means goal

Table 1. Goals and commitments from the aerospace scenario.

ID	Goal, Commitment, or Event	Description
G_1	G(MFG, $\top$, $\top$, payment_made $\wedge$ health_reporting_promised, payment_made $\wedge$ health_reporting_promised, insufficient_money)	Airline manufacturer's (MFG's) goal to receive the payment and the promise to provide the health report
G_2	G(OPER, $\top$, $\top$, engine_provided $\wedge$ service_promised, engine_provided $\wedge$ service_promised, engine_not_provided)	Operator's (OPER's) goal to receive the engine and the promise to provide the service
G_3	G(OPER, $\top$, $\top$, payment_made $\wedge$ health_reporting_promised, payment_made $\wedge$ health_reporting_promised, insufficient_money)	Operator's goal to make the payment and the promise to provide the health report
G_4	G(MFG, $\top$, $\top$, engine_provided $\wedge$ service_promised, engine_provided $\wedge$ service_promised, engine_not_provided)	Airline Manufacturer's goal to provide the engine and the promise to provide the service
$G_5[i]$	G(OPER, service_needed[i], $\top$, service_requested[i], service_requested[i], service_not_requested[i])	Operator's goal to request the service; there is an instance of this goal for each occurrence of service needed
$G_6[i]$	G(MFG, service_requested[i], $\top$, service_provided[i] , service_provided[i], service_not_provided[i])	Manufacturer's goal to provide the service; there is an instance of this goal for each service request
$G_7[i]$	G(MFG, engine_down[i], $\top$, penalty_paid[i], penalty_paid[i], penalty_not_paid[i])	Manufacturer's goal to pay the penalty if the engine is down; there is an instance of this goal for each engine down occurrence
C_1	C(MFG, OPER, payment_made $\wedge$ health_reporting_promised, engine_provided $\wedge$ service_promised)	Mfr's commitment to operator to provide the engine and service if operator pays and promises to provide the health report
C_2	C(OPER, MFG, engine_provided $\wedge$ service_promised, payment_made $\wedge$ health_reporting_promised)	Operator's commitment to the mfr to pay and to provide the health report if the mfr provides the engine and service
$C_3[i]$	C(MFG, OPER, service_requested[i] $\wedge$ $\neg$expired, service_provided[i])	Mfr's commitment to the operator to provide the service if the operator requests service prior to the contract expiration
$C_4[i]$	C(MFG, OPER, engine_down[i] $\wedge$ $\neg$expired, penalty_paid[i])	Manufacturer's commitment to the operator to pay penalty if the engine is down prior to the contract expiration; there is an instance of this commitment for each occurrence of the engine downtime
$C_5[i]$	C(OPER, MFG, health_report_requested[i] $\wedge$ $\neg$expired, health_report_provided[i])	Operator's commitment to the manufacturer to provide the health report if the manufacturer requests the report; there is an instance of this commitment for each health report request

G_4, which corresponds to the operator's (creditor's) end goal G_2. In Step 6, the operator considers and activates G_3 to detach C_1.

In Step 7, due to other priorities, the operator decides to suspend G_2. The operator suspends C_2 (SUSPEND OFFER rule) in Step 8, which transitions its configuration to the coherent state $\langle\{G_2^U\},\{C_2^P\}\rangle$. In Step 9, the manufacturer suspends G_4 (BACK BURNER rule), which transitions its configuration to the coherent state $\langle\{G_4^U\},\{C_2^P\}\rangle$. Observe how the practical reasoning rules cause the manufacturer (debtor) to suspend its means goal G_4 in response to the operator (creditor) suspending its end goal G_2. In Step 10–11, the operator reactivates G_2, and reactivates (REVIVE rule) C_2. In Step 12, the manufacturer reactivates (REVIVE rule) G_4.

In Steps 13–15, the manufacturer provides engine (engine_provided) to the operator and creates C_3 and C_4. Recall that service_promised means creation of C_3 and C_4, and satisfaction condition of G_2 and G_4 is engine_provided $\wedge$ service_promised. Therefore, in Step 15, G_2 and G_4 are satisfied. Further since engine_provided $\wedge$ service_promised is consequent of C_1 and antecedent of C_2, in Step 15, C_1 is satisfied and C_2 is detached. In Steps 16–17, operator pays the manufacturer (payment_made), and creates C_5 (health_reporting_promised). This satisfies G_1, G_3, and C_2. Observe how, in Step 17, the practical reasoning rules cause the manufacturer's and the operator's configuration to reach the coherent states $\langle\{G_1^S\},\{C_1^S\}\rangle$ and $\langle\{G_2^S\},\{C_2^S\}\rangle$.

A *service_needed* event occurs at Step 18; it instantiates the parameter i with the value 1. In response, the operator activates $G_5[1]$, an instance of G_5, to request the service in Step 19. By its requesting the service, in Step 20 the operator satisfies $G_5[1]$ and detaches $C_3[1]$, an instance of C_3. To deliver upon its commitment, the manufacturer activates $G_6[1]$ in Step 21, and provides the service in Step 22. This satisfies $G_6[1]$, and $C_3[1]$. Finally, in Step 23, only the recurring commitments C_3, C_4, and C_5 remain in the agent configurations.

6 Related Work

Chopra et al. [3] formalize semantic relationship between agents and protocols encoded as goals and commitments, respectively to verify at design time if a protocol specification (commitments) supports achieving goals in an agent specification, and vice versa. In contrast, our semantics applies at runtime, and we propose practical reasoning rules that agents may follow to achieve coherence between related goals and commitments. Dalpiaz et al. [4] propose a model of agent reasoning based on pursuit of *variants*—abstract agent strategies for pursuing a goal. We conjecture that their approach can be expressed as sets of practical reasoning rules, such as those we described above.

Winikoff [15] develops a mapping from commitments to BDI-style plans. He modifies SAAPL, an agent programming language, to include commitments in an agent's belief-base and operational semantics update the commitments. Our operational semantics addresses goals (more abstract than plans) and commit-

Table 2. Progression of configurations in the aerospace scenario.

#	Event or Rule	MFG's Action	MFG's State	OPER's Action	OPER's State
1	(structural)	consider(G_1) ∧ activate(G_1)	$\langle\{\boldsymbol{G_1^A}\}\rangle$		$\langle\ \rangle$
2	(structural)		$\langle\{G_1^A\}\rangle$	consider(G_2) ∧ activate(G_2)	$\langle\{\boldsymbol{G_2^A}\}\rangle$
3	ENTICE	create(C_1)	$\langle\{G_1^A\},\{\boldsymbol{C_1^C}\}\rangle$		$\langle\{G_2^A\},\{\boldsymbol{C_1^C}\}\rangle$
4	ENTICE		$\langle\{G_1^A\},\{C_1^C,\boldsymbol{C_2^C}\}\rangle$	create(C_2)	$\langle\{G_2^A\},\{C_1^C,\boldsymbol{C_2^C}\}\rangle$
5	DETACH	consider(G_4) ∧ activate(G_4)	$\langle\{G_1^A,\boldsymbol{G_4^A}\},\{C_1^C,C_2^C\}\rangle$		$\langle\{G_2^A\},\{C_1^C,C_2^C\}\rangle$
6	DETACH		$\langle\{G_1^A,G_4^A\},\{C_1^C,C_2^C\}\rangle$	consider(G_3) ∧ activate(G_3)	$\langle\{G_2^A,\boldsymbol{G_3^A}\},\{C_1^C,C_2^C\}\rangle$
7	(structural)		$\langle\{G_1^A,G_4^A\},\{C_1^C,C_2^C\}\rangle$	suspend(G_2)	$\langle\{\boldsymbol{G_2^U},G_3^A\},\{C_1^C,C_2^C\}\rangle$
8	SUSPEND OFFER		$\langle\{G_1^A,G_4^A\},\{C_1^C,\boldsymbol{C_2^P}\}\rangle$	suspend(C_2)	$\langle\{G_2^U,G_3^A\},\{C_1^C,\boldsymbol{C_2^P}\}\rangle$
9	BACK BURNER	suspend(G_4)	$\langle\{G_1^A,\boldsymbol{G_4^U}\},\{C_1^C,C_2^P\}\rangle$		$\langle\{G_2^U,G_3^A\},\{C_1^C,C_2^P\}\rangle$
10	(structural)		$\langle\{G_1^A,G_4^U\},\{C_1^C,C_2^P\}\rangle$	reactivate(G_2)	$\langle\{\boldsymbol{G_2^A},G_3^A\},\{C_1^C,C_2^P\}\rangle$
11	REVIVE		$\langle\{G_1^A,G_4^U\},\{C_1^C,\boldsymbol{C_2^C}\}\rangle$	reactivate(C_2)	$\langle\{G_2^A,G_3^A\},\{C_1^C,\boldsymbol{C_2^C}\}\rangle$
12	REVIVE	reactivate(G_4)	$\langle\{G_1^A,\boldsymbol{G_4^A}\},\{C_1^C,C_2^C\}\rangle$		$\langle\{G_2^A,G_3^A\},\{C_1^C,C_2^C\}\rangle$
13	(structural)	engine_provided	$\langle\{G_1^A,G_4^A\},\{C_1^C,C_2^C\}\rangle$		$\langle\{G_2^A,G_3^A\},\{C_1^C,C_2^C\}\rangle$
14	(structural)	create(C_3)	$\langle\{G_1^A,G_4^A\},\{C_1^C,C_2^C,\boldsymbol{C_3^C}\}\rangle$		$\langle\{G_2^A,G_3^A\},\{C_1^C,C_2^C,\boldsymbol{C_3^C}\}\rangle$
15	(structural)	create(C_4)	$\langle\{G_1^A,\boldsymbol{G_4^S}\},\{\boldsymbol{C_1^S},\boldsymbol{C_2^D},C_3^C,\boldsymbol{C_4^C}\}\rangle$		$\langle\{\boldsymbol{G_2^S},G_3^A\},\{\boldsymbol{C_1^S},\boldsymbol{C_2^D},C_3^C,\boldsymbol{C_4^C}\}\rangle$
16	(structural)		$\langle\{G_1^A\},\{C_2^D,C_3^C,C_4^C\}\rangle$	payment_made	$\langle\{G_3^A\},\{C_2^D,C_3^C,C_4^C\}\rangle$
17	(structural)		$\langle\{\boldsymbol{G_1^S}\},\{\boldsymbol{C_2^S},C_3^C,C_4^C,\boldsymbol{C_5^C}\}\rangle$	create(C_5)	$\langle\{\boldsymbol{G_3^S}\},\{\boldsymbol{C_2^S},C_3^C,C_4^C,\boldsymbol{C_5^C}\}\rangle$
18	service_needed[1]		$\langle\{C_3^C,C_4^C,C_5^C\}\rangle$		$\langle\{C_3^C,C_4^C,C_5^C\}\rangle$
19	DETACH		$\langle\{C_3^C,C_4^C,C_5^C\}\rangle$	consider($G_5[1]$) ∧ activate($G_5[1]$)	$\langle\{\boldsymbol{G_5^A[1]}\},\{C_3^C,C_4^C,C_5^C\}\rangle$
20	(structural)		$\langle\{\boldsymbol{C_3^D[1]},C_3^C,C_4^C,C_5^C\}\rangle$	service_requested[1]	$\langle\{\boldsymbol{G_5^S[1]}\},\{\boldsymbol{C_3^D[1]},C_3^C,C_4^C,C_5^C\}\rangle$
21	DELIVER	consider($G_6[1]$) ∧ activate($G_6[1]$)	$\langle\{\boldsymbol{G_6^A[1]}\},\{C_3^D[1],C_3^C,C_4^C,C_5^C\}\rangle$		$\langle\{C_3^D[1],C_3^C,C_4^C,C_5^C\}\rangle$
22	(structural)	service_provided	$\langle\{\boldsymbol{G_6^S[1]}\},\{\boldsymbol{C_3^S[1]},C_3^C,C_4^C,C_5^C\}\rangle$		$\langle\{\boldsymbol{C_3^S[1]},C_3^C,C_4^C,C_5^C\}\rangle$
23			$\langle\{C_3^C,C_4^C,C_5^C\}\rangle$		$\langle\{C_3^C,C_4^C,C_5^C\}\rangle$

ments. It will be interesting to combine Winikoff's work with ours to develop a joint semantics for commitments, goals, and plans.

Avali and Huhns [1] relate an agent's commitments to its beliefs, desires, and intentions using BDI_{CTL*}. In contrast, we relate an agent's commitments to its goals. We consider goal lifecycle in our semantics, and propose practical reasoning rules for coherence with commitments.

Telang and Singh [11] enhance Tropos, an agent-oriented software engineering methodology, with commitments. They describe a methodology that starts from a goal model and derives commitments. Our operational semantics complements by providing a formal underpinning.

Telang and Singh [12] propose a commitment-based business metamodel, a set of modeling patterns, and an approach for formalizing the business models and verifying message sequence diagrams with respect to the models. Our combined operational semantics of commitments and goals can provide a basis for how a business model can be enacted and potentially support the derivation of suitable message sequence diagrams.

van Riemsdijk et al. [9] and Thangarajah et al. [13] propose abstract architectures for goals, on which is based the simplified goal lifecycle that we consider. These and other authors formalize the goal operationalization. In contrast, our work formalizes the combined operational semantics of goals and commitments. A future extension of our work is to address the different goal types that have been suggested [9, 17]. Our work is complementary also to exploration of goals that have temporal extent (e.g., [2, 7]). Moreover, we have considered each goal to be private to an agent. Works that study coordination of agents via shared proattitudes, such as shared goals, include for example [6, 8].

7 Conclusion and Future Work

This paper studied the complementary aspects of commitments and goals by establishing an operational semantics of the related lifecycles of the two concepts. We have distinguished the purely semantic aspects of their lifecycles from the pragmatic aspects of how a cooperative agent may reason, and demonstrated desirable properties such as convergence of mental states. From the viewpoint of agent programming, we have provided a foundational set of rules that is complete in a technical sense; their sufficiency in practice will be found through use.

Our work carries importance because of its formalization of the intuitive complementarity between goals and commitments. Directions for building on this foundation include considering a hierarchy of prioritized goals or commitments, and extending our semantics to include maintenance goals, shared goals, or plans. We are also interested in examining converge properties when there are more than two agents working collaboratively.

Acknowledgments. We gratefully acknowledge the suggestions of the anonymous reviewers.

References

1. Avali, V.R., Huhns, M.N.: Commitment-based multiagent decision making. In: Proc. Cooperative Information Agents. pp. 249–263 (2008)
2. Braubach, L., Pokahr, A.: Representing long-term and interest BDI goals. In: Proc. ProMAS Workshop (2009)
3. Chopra, A.K., Dalpiaz, F., Giorgini, P., Mylopoulos, J.: Reasoning about agents and protocols via goals and commitments. In: Proc. AAMAS. pp. 457–464 (2010)
4. Dalpiaz, F., Chopra, A.K., Giorgini, P., Mylopoulos, J.: Adaptation in open systems. In: Proc. 29th Conf. Conceptual Modeling. pp. 31–45 (2010)
5. Desai, N., Chopra, A.K., Singh, M.P.: Amoeba: A methodology for modeling and evolution of cross-organizational business processes. ACM Trans. Software Engineering and Methodology 19(2), 6:1–6:45 (2009)
6. Grosz, B., Kraus, S.: Collaborative plans for complex group action. Artificial Intelliegence 86(2), 269–357 (1996)
7. Hindriks, K.V., van der Hoek, W., van Riemsdijk, M.B.: Agent programming with temporally extended goals. In: Proc. AAMAS. pp. 137–144 (2009)
8. Lesser, V., Decker, K., Wagner, T., Carver, N., Garvey, A., Horling, B., Neiman, D., Podorozhny, R., NagendraPrasad, M., Raja, A., Vincent, R., Xuan, P., Zhang, X.: Evolution of the GPGP/TAEMS Domain-Independent Coordination Framework. J. Autonomous Agents and Multi-Agent Systems 9(1), 87–143 (2004)
9. van Riemsdijk, M.B., Dastani, M., Winikoff, M.: Goals in agent systems. In: Proc. AAMAS. pp. 713–720 (2008)
10. Singh, M.P.: An ontology for commitments in multiagent systems. AI and Law 7, 97–113 (1999)
11. Telang, P.R., Singh, M.P.: Enhancing Tropos with commitments. In: Conceptual Modeling: Foundations and Applications. pp. 417–435. LNCS 5600, Springer (2009)
12. Telang, P.R., Singh, M.P.: Specifying and verifying cross-organizational business models. IEEE Trans. Services Comput. 4 (2011)
13. Thangarajah, J., Harland, J., Morley, D., Yorke-Smith, N.: Operational behaviour for executing, suspending and aborting goals in BDI agent systems. In: Proc. DALT Workshop. pp. 1–17 (2010)
14. van Aart, C.J., Chábera, J., Dehn, M., Jakob, M., Nast-Kolb, K., Smulders, J.L.C.F., Storms, P.P.A., Holt, C., Smith, M.: Use case outline and requirements. Deliverable D6.1, IST CONTRACT Project (2007), http://tinyurl.com/6adejz
15. Winikoff, M.: Implementing commitment-based interactions. In: Proc. AAMAS. pp. 873–880 (2007)
16. Winikoff, M., Padgham, L., Harland, J., Thangarajah, J.: Declarative and procedural goals in intelligent agent systems. In: Proc. KR. pp. 470–481 (2002)
17. Winikoff, M., Dastani, M., van Riemsdijk, M.B.: A unified interaction-aware goal framework. In: Proc. ECAI. pp. 1033–1034 (2010)

Chapter 2

Applying (Multi-)Agent Oriented Programming

Developing a Knowledge Management Multi-Agent System Using *JaCaMo*

Carlos M. Toledo[1], Rafael H. Bordini[2], Omar Chiotti[1], and María R. Galli[1]

[1] INGAR-CONICET, Avellaneda 3657, Santa Fe Argentina
{cmtoledo, chiotti, mrgalli}@santafe-conicet.gov.ar

[2] INF-UFRGS, CP 15064, 91501-970 Porto Alegre RS, Brazil
r.bordini@inf.ufrgs.br

Abstract. Recent research on social and organisational aspects of multi-agent systems has led to practical organisational models and the idea of organisation-oriented programming. These organisational models help agents to achieve shared (global) goals of the multi-agent system. Having an organisational model is an important advance, but this model needs to be integrated to an environment infrastructure and agent-oriented programming platforms. The *JaCaMo* platform is the first fully operational programming platform that integrates three levels of multi-agent abstractions: an agent programming language, an organisational model, and an environment infrastructure. In this paper, with the aim to help showcase the advantages of a fully-fledged multi-agent platform, we have modelled a concrete agent-based architecture to proactively supply knowledge to knowledge-intensive workflows using *JaCaMo*.

Keywords: Multi-agent systems, Multi-agent organisation, Multi-agent programming platform, Knowledge management, Knowledge intensive workflow.

1 Introduction

Agent autonomy becomes a crucial problem when a global coherent behaviour has to be accomplished in multi-agent systems. Its importance is not to be discussed when it comes to achieving suitable flexible autonomous behaviour, but some degree of restriction on agent behaviour is required for a system to achieve shared global goals [11]. This problem is even worse in open Multi-Agent Systems (MAS), where the agents that could enter or leave the system is not known *a priori*. In this context, organisational models arise as a way to control the agents' behaviour, so they can work as a coordinated team achieving common goals.

Recent research on social and organisational aspects of multi-agent systems has led to concrete approaches to organisation-oriented programming, such as

Moise [12]. *Moise* is an organisational model that considers structural, functional, and deontic relations among agents describing organisational collaboration in order to address the collective behaviour. When an agent enters the system, it must comply with organisational rules constraining its autonomy. *Moise* offers an organisational modelling language and principles for defining these rules, organising the system, and ensuring organisational constraints by means of notions such as roles, groups, schemes, missions, and deontic relations such as obligations.

Having an organisational model is an important advance, but this model needs to be integrated with an environment infrastructure and an agent programming platform. Agents must reason about their organisation and interact with their environments where the organisation is situated. Recently, researchers have focused on the integration of theoretical organisational models with agent-oriented programming platforms so as to provide practical solutions for MAS development [12, 8, 10, 11]. With that purpose, the *JaCaMo* MAS programming platform[3] was developed and is, to our knowledge, the first fully-operational multi-agent programming platform to allow users to take advantage of first-class abstractions and declarative language constructs that encompass the three main levels of abstractions of a multi-agent system, namely agent, environment, and organisational levels. It was developed through the integration of three independent existing MAS technologies: *Jason* [5], *CArtAgO* [21], and *Moise* [12], addressing the agent, environment, and organisation levels respectively.

In this work, we present the concrete development of a MAS architecture using the *JaCaMo* platform. This architecture allows the proactive supply of knowledge to knowledge-intensive workflows by integrating the Business Process Management (BPM) and Knowledge Management (KM) infrastructures. The remainder of the paper is structured as follows: Section 2 describes the various features of the *JaCaMo* platform. Section 3 presents an agent-based architecture for BPM and KM integration. In Section 4, we put *JaCaMo*'s functionalities into practice by specifying the architecture of our application. Finally, Section 6 presents conclusions and future work.

2 *JaCaMo* Platform

JaCaMo[4] is a platform for multi-agent programming that combines three independent existing MAS technologies: *Jason*[5] [5], *CArtAgO*[6] [21], and *Moise*[7] [12]. *JaCaMo* allows developers to program organisation-aware agents using an agent programming language and supports the A&A meta-model [17] for the implementation of environment-based coordination mechanisms, and non-autonomous services and tools. *JaCaMo* addresses the three main levels of MAS abstractions

[3] http://jacamo.sourceforge.net
[4] Available at http://jacamo.sourceforge.net/
[5] Available at http://jason.sourceforge.net/
[6] Available at http://cartago.sourceforge.net/
[7] Available at http://moise.sourceforge.net/

providing high-level support for implementing BDI (Belief-Desire-Intention) agents, virtual artifact-based environments [26], and organisational models. It should be highlighted that *JaCaMo* is more than a set of MAS technologies, it is a unified and fully operational platform that supports designers in developing complex multi-agent oriented software using high-level abstractions in all three main dimensions of a multi-agent system. The following subsections detail the *JaCaMo* functionalities.

2.1 Organisational Model

JaCaMo uses the *Moise* model to specify MAS organisations. *Moise* is a model for the organisational dimension of multi-agent systems. It provides an organisation modelling language, an organisation management infrastructure, and supports organisation-based reasoning mechanisms at the agent level. *Moise* has three dimensions that deal with different aspects of the organisational specification: *structural*, *functional*, and *deontic*.

The *structural dimension* deals with the more static aspect of the organisational model, and it is built in three levels: the *individual level*, which deals with the responsibilities that an agent assumes when it adopts a role; the *social level*, which describes acquaintance, communication, and authority links between roles; and the *collective level*, which is responsible for the aggregation of roles into groups.

At the *individual level*, agents adopt roles in organisational groups, accepting some behavioural constraints related to such roles. This adoption is constrained by compatibility relations between roles, so an agent can play two or more roles only if they are compatibles. At the *collective level*, agents are divided into groups and sub-groups. Groups and sub-groups are considered well-formed if they comply with the constraints referred to maximum and minimum number of members playing each role. At the *social level*, roles are linked through authority, communication, and acquaintance relationships. These relationships indicate the degree of influence or the rights that an agent that plays a role has over other agents that play the linked roles. Relationships can be intra-group and inter-group depending if their linked roles belong to the same group or to different groups.

The *functional dimension* deals with the dynamic aspects of the organisational specification. It intends to make agents work together to achieve global goals. This dimension is composed of a set of *schemes* in which a goal is decomposed into social plans and distributed to agents through *missions* [12]. These schemes are modelled by trees in which the root is a global purpose and the leafs are goals. Achieving some or may be all of the goals make the scheme succeed.

Schemes also define execution plans that indicate partial orders for the achievement of goals. The execution of goals of a plan can be sequential, parallel, or alternative. In a sequence, a goal can be achieved only if the preceding goal has been previously achieved. In a parallel decomposition, two goals can be achieved at the same time. In an alternative decomposition, a goal is achieved if any of its sub-goals is achieved.

Moreover, each scheme defines a number of missions. A mission is a set of coherent goals that an agent playing a particular roles can/must achieve [12] within a given amount of time (time-to-fulfil or TTF). Agents commit to missions, so they are responsible for their fulfilment. Schemes can be represented diagrammatically, as in Figures 4 and 5 that we introduce later in this paper.

The *deontic dimension* links the *structural dimension* (roles and groups) with the *functional dimension* (schemes). It limits the autonomy of the agents by defining what missions an agent, playing a role in the organisation, is permitted to take on and to what missions it is obliged to commit. A relationship $permission(r, m)$ specifies that agents that play role r *can* commit to mission m, while a relationship $obligation(r, m)$ specifies that agents that play role r *must* commit to mission m.

2.2 Agent Platform

To develop the agents of the application, *JaCaMo* uses the *Jason* MAS platform [5]. *Jason* is an interpreter for an extended version of the AgentSpeak language [20] that provides a platform for the development of BDI MAS. *Jason* allows user-defined internal actions and it is fully customisable in Java (perception, selection functions, trust functions, belief-revision, agent communication, etc. are all customisable).

Moreover, *Jason* includes a clear notion of a multi-agent environment and the capability to program MAS distributed over a network, including through JADE [4]. Communication mechanisms of *Jason* allow agents to communicate with each other in a high-level way. *Jason* uses speech-act based communication among AgentSpeak agents, based on the Knowledge Query and Manipulation Language (KQML) performatives. For more details, see [5].

2.3 Environment as a First-Class Abstraction

In MAS development tools, ideally the environment should be not only the source of agent perceptions and the target of agent actions, but is also considered as an active abstraction, i.e. a first-class entity that encapsulates functionalities and services by supporting coordination, agent mobility, communication, security, and non-autonomous special functions [26].

JaCaMo provides an infrastructure to program the MAS environment adopting the A&A meta-model, which is implemented by the *CArtAgO* technology [21]. *JaCaMo* uses the environment to have access to external resources, which allows it to interact with hardware/software resources hiding away low-level aspects. The environment also provides a mechanism to have access to shared resources and mediate the interaction between agents.

This way, with *JaCaMo* users can program the environment by means of a set of first-class entities called artifacts. Each artifact represents a resource or tool that agents can instantiate, share, use, and perceive at runtime [17]. Artifacts are non-autonomous and function-oriented entities (similar to objects

in the oriented-object paradigm), unlike agents that are autonomous and goal-oriented entities. Unlike objects, artifacts are at the same level of abstraction as agents in declarative approaches to programming multi-agent systems.

Agents and artifacts communicate through *actions* and *perceptions*. Agents execute *actions* (or operations) of the artifact *usage interface* (or control interface) to modify the artifact state or to request a service. Changes in the state of an artifact are perceived by agents through observable events. These events are perceived by agents and potentially become agent beliefs.

Additionally, artifacts can be linked and work together accomplishing their purposes by executing operations of the *link interfaces* of other artifacts. When an agent (or another artifact) executes an operation over an artifact (of its usage or link interface), the operation request is suspended while the invoked operation is executed and a result (success or failure) is returned. Artifacts handle all aspects of concurrency control for the operations they provide.

Agents and artifacts are grouped in workspaces which act as their logical container. Workspaces define the MAS environment topology. They can be distributed while agents and artifacts work together by means of direct communication and sharing artifacts. An artifact can belong to a single workspace whereas an agent can inhabit one or more workspaces and share artifacts. Agents that work in the same workspace can use the same artifacts.

2.4 Three-Level Abstraction Integration

The three main MAS levels of abstractions (agent, organisation, and environment) are available in the *JaCaMo* platform. Agents are integrated to the environment (agent-environment integration) by joining and working in a workspace, using the artifact operations, and sensing the observable properties of artifacts. This fact allows heterogeneous agents to work in the same artifact-based environment, sharing the repertoire of actions and the observable events that agents can perceive.

The organisation-environment integration is based on the ORA4MAS infrastructure [10]. The organisation management infrastructure is developed as a set of *organisational artifacts* and *organisational agents*. They are responsible for encapsulating functionalities concerning the management and enactment of the organisation specification, implementing and controlling *regimentation* and *enforcement* mechanisms. Regimentation aims to prevent agents from performing actions that are forbidden by an organisational norm, whereas enforcement aims to give tools to detect possible norm violations, checking whether they were violations or not, and applying appropriate sanctions in case they were [10].

In ORA4MAS, there are two main types of *organisational artifacts*: *scheme artifact* and *group artifact*. They are responsible for implementing the structural, functional, and normative dimensions of *Moise* through a set of operations that allow agents to adopt a role, to join or leave a group, to commit to a mission, etc. On the other hand, the *organisational agents* are responsible for the creation and management of organisational artifacts and for performing *enforcement* activities. If a norm is violated, artifacts should detect and show this violation,

and the organisational agents should deal with this. Organisational agents and organisational artifacts are described in detail in [14].

The agent-organisation integration happens through the mapping of the agents' actions into organisational artifact operations. That is, an agent executes an operation (such as adopt a role, leave a mission, etc.) of an organisational artifact by means of its repertoire of environment actions. This provides a unified mechanism for taken normal environment actions as well as organisational actions (i.e., actions that change the state of an agent organisation).

3 An Agent-Based Architecture for BPM and KM Integration

These days, many organisations and institutions coordinate their activities through business processes. To accomplish the tasks involved in a business process, workers may need information that is scattered throughout the organisation. This need for information should be attended to in a contextualised way and be suitably delivered by the organisational Knowledge Management (KM) infrastructure [19]. Business Process Management (BPM) and KM are closely interrelated and are both important elements of an organisation.

Organisational activities structured through a business process and automated by a Workflow Management System (WfMS) should be considered as an opportunity to provide a KM infrastructure in order to bring the right knowledge to the right people in the right form and at the right time [9]. A WfMS defines, manages, and executes structured business processes (workflows) arranging task executions by means of a computational representation of the workflow logic. It should be a trigger of KM support activities and a distributor of organisational knowledge that provides workers with necessary information to make better judgements and decisions.

Some complex business processes rely on intensive information needs, such as design processes, government tenders, decision making activities, etc., in which workers would like WfMS to automatically and proactively offer the relevant knowledge [16]. When a worker selects a task for execution, the system should make available all relevant information for executing the selected task. It allows workers to automatically obtain relevant information, avoiding the waste of time with information searches.

Considerable work has dealt with the integration between KM and BPM in order to combine the advantages of both paradigms; however, most existing approaches are only theoretical approaches and they do not put forward any concrete solutions [19]. Other existing frameworks fail to support the diversity of knowledge domains, making static systems in which the knowledge domains are tied with the architecture [1, 13].

Toledo et al. [25] proposed an agent-based architecture for the integration of business processes orchestrated by a WfMS with the organisational knowledge repository administered by ontology-driven KM systems (Figure 1). This is the The architecture is based on a distributed organisational memory [22] and the

KM conceptual model proposed in [2]. This architecture consists of an *organisational knowledge repository* and one or more *business processes* automated by a *WfMS*.

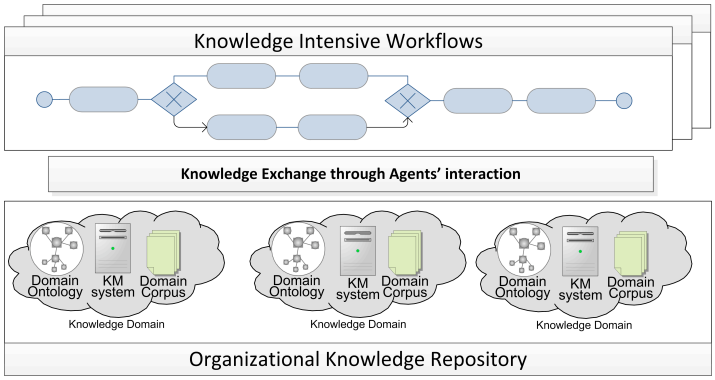


Fig. 1. Organisational Knowledge Management Architecture.

In the *organisational knowledge repository*, the organisation can be seen as an interrelated group of functional units, areas, or departments formally defined in the organisational structure, informal groups, or practice communities. Each of these units is considered an organisational knowledge domain which executes activities and processes that produce and consume information. These activities and processes are often unstructured, dynamic, unpredictable, and constantly changing; also, they are not included in a workflow. This need for information is satisfied in a reactive way through domain KM systems.

In KM systems are responsible for storing knowledge about their respective domains and providing information to requests from their own knowledge domain, to other knowledge domains, and, in the case of this proposal, to workflows as well. The set of all knowledge domains of the organisation constitutes the *organisational knowledge repository*, i.e., the organisational memory [15].

Each knowledge domain implements its own KM system. This KM system is part of a distributed organisational memory [22]. KM systems typically use an ontology (although this is not mandatory) to annotate and retrieve information based on a strategy that processes users' natural language queries.

In the *knowledge intensive workflows* component, there is a set of business processes automated as a workflow composed of one or more tasks that need and/or produce valuable enterprise knowledge as a result of their execution, the so called Knowledge Intensive Tasks (KITs). For each KIT, the KM architecture should proactively provide relevant information for its execution, considering the KIT context (profile of the organisation's worker, worker role profile, task specification, and workflow instance information) and workers' behaviour in previous executions. Also, it stores relevant information produced as a result of their execution in a suitable knowledge domain. Unlike organisation functional units (knowledge domain), which execute their processes in an unstructured

way, knowledge intensive workflows execute their tasks in a structured process orchestrated by a WfMS.

With the aim of exchanging knowledge between the two components and proactively and reactively delivering knowledge, the KM architecture implements a MAS. It enables the integration between the KM systems executed in the knowledge domains of the organisational knowledge repository and the knowledge intensive workflows orchestrated by a WfMS. Through suitable interaction protocols, agents exchange the necessary information and control data to satisfy the workers' need for information within the workflow execution. Conventional workflows are extended with KM-support automatic tasks that execute instances of software agents. These agents are responsible for retrieving knowledge and storing results.

The architecture uses MAS technologies for the following reasons. First, it is an open system; workflows and knowledge domains can be added automatically. Knowledge domains follow the autonomy principle [22] that enables each domain to manage the local information, providing the possibility of choosing more appropriate perspectives, mechanisms, and policies (e.g., security policies, domain ontology) to represent the local knowledge. Workflows and knowledge domains cannot be known in advance. Even the architecture can be extended to include foreign information sources, such as information from partner organisations used to provide knowledge to workflows that produce a product or service with/for the partner organisation. Second, agents can easily add machine learning techniques that allow the architecture to improve the knowledge provision and storing processes. Third, the involved entities should have autonomous behaviour in deciding to provide or store the knowledge independently from the rest of the architectural entities. The integration architecture needs to decide the most convenient way to store the generated information, and the most suitable information sources to consult.

In the following section, this architecture is specified using the *JaCaMo* platform.

4 Specifying the Architecture with the *JaCaMo* Platform

4.1 Environment Specification

The proposed MAS architecture is composed of a set of distributed artifact-based workspaces [18], which can be classified in three types: *KIT workspaces*, *knowledge domain workspaces*, and the *knowledge integration workspace*. Each workspace contains agents and artifact instances that supply agents with information and help them to coordinate their activities. Artifacts also encapsulate other non-autonomous services, such as the *KM system artifact* which is responsible for managing the local domain knowledge (Figure 2).

Each *KIT workspace* models the enactment of a KIT, and contains instances of artifacts and agents responsible for providing and storing information relevant to the KIT. The KIT workspace contains instances of the *knowledge provider*

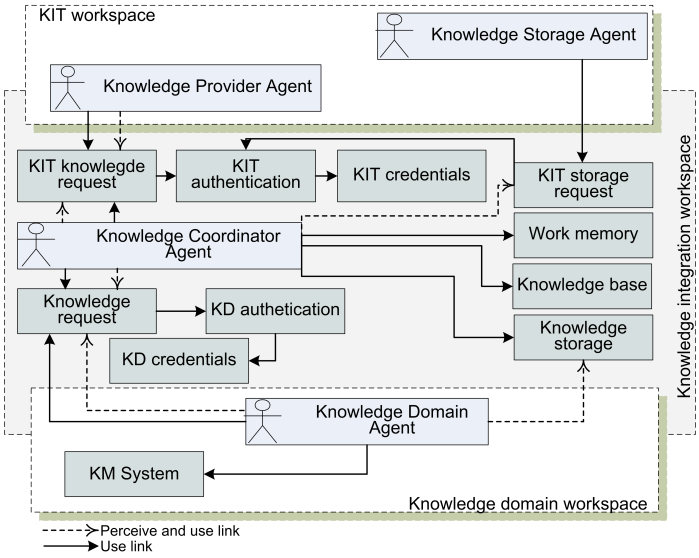


Fig. 2. MAS Topology

agent and the *knowledge storage agent*. The former is responsible for retrieving information relevant to the KIT, taking into account data related to the worker profile, the worker role profile, and the KIT. The KIT specification is responsible for keeping information about a KIT. For details of its structure, see [25].

The knowledge storage agent is responsible for collecting results obtained from the KIT execution. Once all the generated and/or modified information documents are collected, this agent establishes a communication process with the aim of storing results in the suitable knowledge domain. To determine the knowledge domain in which results will be stored, configuration parameters of the KIT specification artifact are taken into account.

The knowledge provider agent and the knowledge storage agent join the KIT workspace as well as the knowledge integration workspace. This allows agents to use artifacts of both workspaces and exchange data, enabling the integration of the architecture.

The knowledge domain workspace represents each organisational knowledge domain; therefore, there are as many knowledge domain workspaces in the architecture as there are knowledge domains in the organisation. Each knowledge domain workspace includes an instance of the KM system artifact that is responsible for providing and storing the domain knowledge (an approach for its implementation is detailed in [24]). The *knowledge domain agent* uses operations of the KM system artifact to consult information and then provides this information to a KIT through the *knowledge coordinator agent*. The knowledge domain agent joins the knowledge domain workspace and the knowledge integration workspace.

Finally, the MAS architecture has a single instance of the knowledge integration workspace which makes the integration between *organisational knowledge*

repositories and *knowledge intensive workflows* possible by means of artifact-based coordination and communication among agents. The knowledge coordinator agent is responsible for making possible the knowledge exchange among workflows and knowledge domains. It can also proactively offer knowledge to KIT workers. The knowledge coordinator agent also manages the organisation by applying sanctions and rewards, and controlling the access of KITs and Knowledge Domains (KDs) to the system.

The knowledge integration workspace is inhabited by a set of artifacts that allow the architecture to authenticate KIT and KD instances (KIT authentication and KD authentication artifacts), store access credentials (KIT credentials and KD credentials artifacts), coordinate agents and exchange knowledge (KIT knowledge storage request, KIT knowledge request, knowledge request, knowledge storage, and knowledge response artifacts), and help the knowledge coordinator agent to fulfil its responsibilities (work memory and knowledge base artifacts). The following code shows how the knowledge coordinator agent creates and focuses on the artifacts of its environment.

```
+!create_artifacts: true
 <- makeArtifact("KnowledgeProvider","kiWorkspace.KnowledgeProvider",[],KnowledgeProviderID);
    makeArtifact("KITAutentication","kiWorkspace.KITAutentication",[],KITAutenticationID);
    makeArtifact("KDAutentication","kiWorkspace.KDAutentication",[],KDAutentication ID);
    makeArtifact("KDCredentials","kiWorkspace.KDCredentials",[],KDCredentialsID);
    makeArtifact("KITCredentials","kiWorkspace.KITCredentials",[],KITCredentialsID);
    makeArtifact("KITStorageRequest","kiWorkspace.KITStorageRequest",[],KITStorageRequestID);
    makeArtifact("KnowledgeBase","kiWorkspace.KnowledgeBase",[],KnowledgeBaseID);
    makeArtifact("WorkMemory","kiWorkspace.WorkMemory",[],WorkMemoryID);
    makeArtifact("KnowledgeRequest","kiWorkspace.KnowledgeRequest",[],KnowledgeRequestID);
    makeArtifact("KnowledgeResponse","kiWorkspace.KnowledgeResponse",[],KnowledgeResponseID);
    makeArtifact("KnowledgeStorage","kiWorkspace.KnowledgeStorage",[],KnowledgeStorageID);
    focus(KnowledgeProviderID);
    focus(KITStorageRequestID);
    focus(KnowledgeRequestID);
    focus(KnowledgeResponseID).
```

Also, the knowledge coordinator agent creates two organisational artifacts to manage the organisational model: *group artifact* and *scheme artifact* (for the sake of clarity, these are not shown in the Figure 3). The scheme artifact keeps track of which goals are feasible and creates the respective obligations for agents, whereas the *group artifact* manages the organisational groups. The organisational specification (groups, goals, roles, etc.) is defined in the *organisation.xml* file.

```
+!create_organisational_artifacts: true
  <- makeArtifact("Scheme", "ora4mas.nopl.SchemeBoard",
       ["organisation.xml", scheme, false, true], SchemeID);
     makeArtifact("Group", "ora4mas.nopl.GroupBoard",
       ["organisation.xml", group, false, true], GroupID);
     focus(SchemeID);
     focus(GroupID).
```

Workspaces together with artifacts form the MAS environment. Agents have access only to artifacts of the workspace that they have joined, but an agent can join more than one workspace. This makes possible the coordination and communication through artifacts of agents that belong to different workspaces. The following code shows how the knowledge domain agent discovers the artifacts when it enters the system.

```
+!enter_kmArchitecture: true
  <- joinWorkspace("KnowledgeIntegration",kiID);
     !discover_artifact("Scheme");
     !discover_artifact("Group");
     !discover_artifact("KnowledgeRequest");
     !discover_artifact("KnowledgeStorage").
+!discover_artifact(ArtifactName): true
  <- lookupArtifact(ArtifactName,ArtifactId);
     focus(ArtifactId).
```

4.2 Structural Specification

In order to achieve global goals, agents' autonomy should be controlled by means of behavioural constraints [11]. These constraints are defined in the organisational specification of our system architecture. As discussed in Section 2, the *JaCaMo* platform follows the *Moise* model to assign an organisation to the system using groups, roles, and shared goals to coordinate the global behaviour.

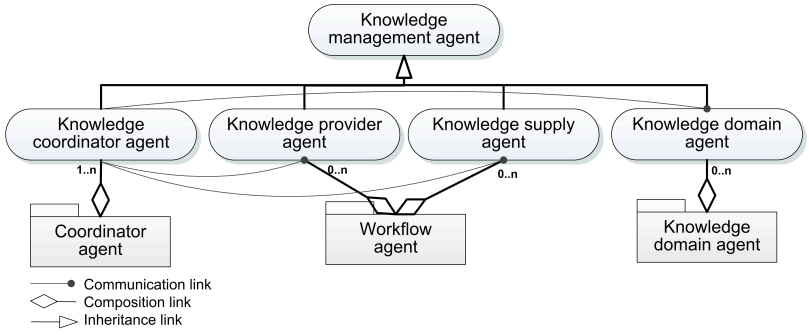


Fig. 3. Structural Specification Diagram.

At the *collective level* of the *Moise* structural dimension, the agents are divided into three groups: *coordinator agents*, *workflow agents*, and *domain agents* (see Figure 3). The *coordinator agent group* is formed by one agent role: the *knowledge coordinator agent*; at least one agent must play this role. The *workflow agent group* is formed by two agent roles: *knowledge provider* and *knowledge supply* agent. The architecture can have zero or more (up to k) KITs, where each KIT is composed of just one instance of a knowledge provider agent and one instance of a knowledge supply agent. The *domain agent group*, is composed of one or more (up to m) agents that play the *knowledge domain agent role*. All these agent roles inherit the properties of the *knowledge management agent role*.

At the *social level* of *Moise*, inter-group communication relationships specify the possibility that two agents can exchange data. The structural specification diagram (Figure 3) depicts that the knowledge coordinator agent can communicate with other agents, allowing the exchange of information between KITs and KDs.

4.3 Functional and Deontic Specifications

The MAS behaviour is determined by social plans that agents should execute together with the aim of achieving global goals. The *Moise* functional dimension decomposes (through plans) global goals and distributes them to agents (through missions) [12]. This architecture has two global goals: *store relevant information* and *provide relevant information*, depicted in Figures 4 and 5 respectively.

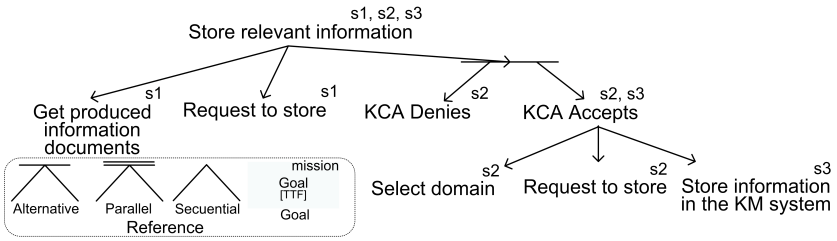


Fig. 4. Functional Specification of the "Store Relevant Information" Goal.

In the first case, when a workflow worker creates or modifies an information document, the knowledge storage agent can trigger the global goal "store relevant information" by committing to mission $s1$. In this case, other agents must commit to the another missions (see the deontic specification, Table 1), and the agents interact in the following way:

- The *knowledge storage agent* creates a request to store information in the *KIT storage request artifact*. This request includes storage criteria (included in the KIT specification), information about the workflow instance, and the information document to be stored (a text document, a spreadsheet, an email, etc.).
- The *KIT storage request artifact* validates the request. If the request comes from an authorised agent, the artifact generates an environment perception, which is perceived by *knowledge coordinator agent* instances.
- The *knowledge coordinator agent* receives the request. Here, it decides, based on information about the KIT (stored in the KIT specification) and its own knowledge (current active knowledge domains, preferred domains, frequently consulted domains, etc.), in which knowledge domain(s) the information will be stored.
- Once the target domain(s) are decided, the *knowledge coordinator agent* creates a storage request in the corresponding *domain storage request artifacts*.
- The *knowledge domain agents* receive this perception and check whether it is directed to them. Finally, the *knowledge domain agent* stores the information in its own *KM system*.

In the case of the second global goal, when a worker selects a KIT from the *work list* for its execution, agents interact with the aim of providing knowledge to the workflow worker in a proactive way, as follows:

– The *knowledge provider agent* makes a request in the *KIT knowledge request artifact*. It includes the KIT specification (which specifies knowledge preferences and information about the KIT), information about the workflow instance (for learning purposes), worker profile, and worker role profile.
– The *KIT knowledge request artifact* validates the request. If the request comes from an authorised agent, it generates an environment percept that is perceived by the *knowledge coordinator agent*.
– The *knowledge coordinator agent* receives the request. It decides (based on the KIT specification, worker profile, worker role profile, and its own knowledge) from which knowledge domain(s) the information will be consulted.
– Once the target domain(s) are decided, the *knowledge coordinator agent* creates a knowledge request in the *knowledge request artifact* of selected domains.
– Finally, the *domain agent* consults the information from its own *KM system* and retrieves the required information for the *knowledge coordinator agent*; then it supplies the information to *knowledge provider agent* that made the request.

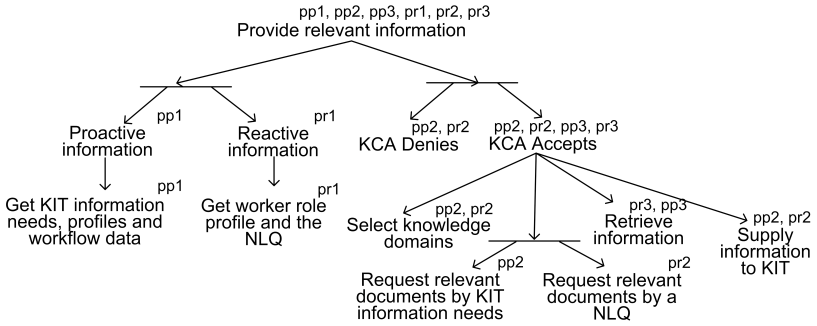


Fig. 5. Functional Specification of the "Provide Relevant Information" Goal.

Workflow workers can also request knowledge in a reactive way through Natural Language Queries (NLQs). In this case, workers make a query which is sent to knowledge domains to be processed and the information that answers the query can be retrieved.

Also, due to the characteristics of the multi-agent system technology and the features of *JaCaMo*, the knowledge coordinator agent can include machine learning strategies and knowledge recommendation mechanisms [27] to improve the knowledge supply. In the same way, as it is an open system where knowledge domains can be automatically added, the knowledge storage process can be improved including information about reliability, security, confidence, etc., information that the knowledge coordinator agent can use to decide in which domain the information will be stored or consulted.

Moreover, the use of a MAS and the encapsulation of the KM system in an artifact allow knowledge domains to implement different information retrieval

Table 1. Deontic specification

role	deontic relation	mission
Knowledge coordinator agent (KCA)	obligation	s2
Knowledge coordinator agent (KCA)	obligation	pp2
Knowledge coordinator agent (KCA)	obligation	pr2
Knowledge domain agent (KDA)	obligation	s3
Knowledge domain agent (KDA)	obligation	pp3
Knowledge domain agent (KDA)	obligation	pr3
Knowledge storage agent (KSA)	permission	s1
Knowledge provider agent (KPA)	permission	pp1
Knowledge provider agent (KPA)	permission	pr1

strategies, according to their own needs, in a simple way. Different knowledge domains can implement different strategies as long as they provide information fulfilling the operations of the KM system artifact. Some of these strategies for information retrieval that the system implements can be found in [2] and [24].

5 Related Work

Recently, some research has focused on the integration of theoretical organisational models with MAS platforms with the purpose of providing practical solutions for MAS development [7, 8, 10–12, 3, 23], but from the point of view of multi-agent system programming, to our knowledge *JaCaMo* is the first fully operational platform that integrates programming language abstractions at all three levels of a multi-agent system: agent, environment, and organisation. In this paper, we show how a practical MAS platform that is comprehensive like *JaCaMo* can significantly facilitate the development of complex multi-agent applications, in this case in the knowledge management application domain. As *JaCaMo* is the first comprehensive multi-agent programming platform that is fully operational, it was the only choice we had to develop the case study presented here.

Of course not all multi-agent system will need all the functionalities of full multi-agent oriented programming. *JaCaMo* is meant to be used when you need complex autonomous agents (and BDI is the best known approach to develop such agents), as well as a dynamic agent organisation situated in an environment that can support agent interaction at the right level of abstraction. We believe this case study has helped illustrate how we can obtain a sophisticated knowledge management system by creating autonomous agents, their organisation, and coordination through the environment using very high-level techniques for each of these levels of the multi-agent system. This makes it clearly easier to develop a system that is open and can evolve at runtime, again by human designers or agents themselves changing very high level specifications of agent, organisation, or environment behaviour.

6 Conclusion and Future Work

We have described an integral platform to program multi-agent systems: *JaCaMo*. This integrates three main levels for multi-agent system programming: agents, organisation, and environment. To our knowledge, *JaCaMo* is the first fully operational programming platform for comprehensive MAS development. It is more than a set of multi-agent technologies; it is a complete platform for multi-agent programming that allows programmers to develop agent-based systems at a high abstraction level.

With the aim of demonstrating the main features of *JaCaMo*, we specified a concrete case study of an agent-based architecture for business process management and knowledge management. This case study shows the functionalities available in *JaCaMo* for the design of multi-agent systems. As *JaCaMo* has been recently released, to our knowledge this is the first application to employ this approach.

In future work, we will define the concrete sanctions that the knowledge coordinator agent can apply when an agent does not obey an obligation, and a set of rewards that the knowledge domains will receive if they answer with suitable information. Also, due to the lack of a suitable modelling language, we plan to develop a modelling language based on MAS-ML [6], which will allow describing all static and dynamic characteristics of *JaCaMo* applications. This is likely to be a language based on the OMG's Meta Object Facility (MOF).

References

1. Abecker, A., Bernardi, A., Maus, H., Sintek, M., Wenzel, C.: Information supply for business processes: coupling workflow with document analysis and information retrieval. Knowledge-Based Systems 13(5), 271 – 284 (2000)
2. Ale, M.A.: An Organizational Knowledge Management Conceptual Model. PhD in information systems, National Technological University (2009)
3. Baldoni, M., Baroglio, C., Bergenti, F., Boccalatte, A., Marengo, E., Martelli, M., Mascardi, V., Padovani, L., Patti, V., Ricci, A., et al.: MERCURIO: An Interaction-oriented Framework for Designing, Verifying and Programming Multi-Agent Systems
4. Bellifemine, F.L., Caire, G., Greenwood, D.: Developing Multi-Agent Systems with JADE. Wiley (2007)
5. Bordini, R., Hübner, J., Wooldridge, M.: Programming multi-agent systems in AgentSpeak using Jason. Wiley (2007)
6. Da Silva, V., Choren, R., de Lucena, C.: Using the mas-ml to model a multi-agent system. Software Engineering for Multi-Agent Systems II pp. 349–351 (2004)
7. Dastani, M., Grossi, D., Meyer, J., Tinnemeier, N.: Normative multi-agent programs and their logics. Knowledge Representation for Agents and Multi-Agent Systems pp. 16–31 (2009)
8. Esteva, M., Rosell, B., Rodríguez-Aguilar, J.A., Arcos, J.L.: AMELI: An agent-based middleware for electronic institutions. Autonomous Agents and MAS, International Joint Conference on 1, 236–243 (2004)
9. Hollingsworth, D.: The workflow reference model. Tech. Rep. TC00-1003, Workflow Management Coalition (1995)

10. Hübner, J., Boissier, O., Kitio, R., Ricci, A.: Instrumenting multi-agent organisations with organisational artifacts and agents. AAMAS 20, 369–400 (2010)
11. Hübner, J., Sichman, J., Boissier, O.: S-Moise$^+$: A middleware for developing organised multi-agent systems. In: Boissier, O., Padget, J., Dignum, V., Lindemann, G., Matson, E., Ossowski, S., Sichman, J., Vzquez-Salceda, J. (eds.) Coordination, Organizations, Institutions, and Norms in Multi-Agent Systems, Lecture Notes in Computer Science, vol. 3913, pp. 64–78. Springer Berlin / Heidelberg (2006)
12. Hübner, J.F., Sichman, J.S., Boissier, O.: Developing organised multiagent systems using the Moise+ model: programming issues at the system and agent levels. Inter. Journal of Agent-Oriented Software Engineering 1(3/4), 370 – 395 (2007)
13. Jung, J., Choi, I., Song, M.: An integration architecture for knowledge management systems and business process management systems. Computers in Industry 58(1), 21 – 34 (2007)
14. Kitio, R., Boissier, O., Hübner, J., Ricci, A.: Organisational artifacts and agents for open multi-agent organisations: giving the power back to the agents. In: Inter. conference on COIN in agent systems III. pp. 171–186. Springer-Verlag (2007)
15. Kühn, O., Abecker, A.: Corporate memories for knowledge management in industrial practice: Prospects and challenges. Journal of Universal Computer Science 3(8), 929–954 (1997)
16. Lai, J., Fan, Y.: Workflow and knowledge management: Approaching an integration. In: Han, Y., Tai, S., Wikarski, D. (eds.) EDCIS. Lecture Notes in Computer Science, vol. 2480, pp. 16–29. Springer (2002)
17. Omicini, A., Ricci, A., Viroli, M.: Artifacts in the A&A meta-model for multi-agent systems. Autonomous Agents and Multi-Agent Systems 17(3), 432–456 (2008)
18. Piunti, M., Ricci, A.: Cognitive use of artifacts: Exploiting relevant information residing in mas environments. In: Meyer, J.J., Broersen, J. (eds.) Knowledge Representation for Agents and Multi-Agent Systems, Lecture Notes in Computer Science, vol. 5605, pp. 114–129. Springer Berlin / Heidelberg (2009)
19. Raghu, T., Vinze, A.: A business process context for knowledge management. Decision Support Systems 43(3), 1062 – 1079 (2007), integrated Decision Support
20. Rao, A.: AgentSpeak (L): BDI agents speak out in a logical computable language. Agents Breaking Away pp. 42–55 (1996)
21. Ricci, A., Piunti, M., Viroli, M.: Environment programming in multi-agent systems: An artifact-based perspective. Autonomous Agents and MAS pp. 1–35 (2010)
22. Souza, R.G.S.: Agent-oriented constructivist knowledge management. Ph.D. thesis, University of Twente, Enschede (2006)
23. Stratulat, T., Ferber, J., Tranier, J.: MASQ: towards an integral approach to interaction. In: Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems-Volume 2. pp. 813–820. International Foundation for Autonomous Agents and Multiagent Systems (2009)
24. Toledo, C.M., Ale, M., Chiotti, O., Galli, M.R.: An agent-based architecture for ontology-driven knowledge management. In: The V International Conference on Knowledge, Information and Creativity Support Systems. Thailand (2010)
25. Toledo, C.M., Chiotti, O., Galli, M.R.: Towards business process management and knowledge management integration through an agent-based architecture. In: XXIX International Conference of the Chilean Computer Society JCC 2010. Chile (2010)
26. Weyns, D., Omicini, A., Odell, J.: Environment as a first class abstraction in multiagent systems. Autonomous Agents and Multi-Agent Systems 14, 5–30 (2007)
27. Zhen, L., Huang, G.Q., Jiang, Z.: Recommender system based on workflow. Decision Support Systems 48(1), 237 – 245 (2009)

Notes on pragmatic agent-programming with Jason

Radek Píbil[1], Peter Novák[2], Cyril Brom[1], and Jakub Gemrot[1]

[1] Department of Software and Computer Science Education
Faculty of Mathematics and Physics, Charles University in Prague
Czech Republic

[2] Agent Technology Center, Department of Cybernetics
Faculty of Electrical Engineering, Czech Technical University in Prague
Czech Republic

Abstract *AgentSpeak(L)*, together with its implementation *Jason*, are one of the most influential agent-oriented programming languages. Besides having a strong conceptual influence on the niche of BDI-inspired agent programming systems, *Jason* also serves as one of the primary tools for education of and experimentation with agent-oriented programming. Despite its popularity in the community, relatively little is reported on its practical applications and pragmatic experiences with adoption of the language for non-trivial applications.
In this work, we present our experiences gathered during an experiment aiming at development of a non-trivial case-study agent application by a novice *Jason* programmer. In our experiment, we tried to use the programming language *as is*, with as few customisations of the *Jason* interpreter as possible. Besides providing a structured feedback on the most problematic issues faced while learning to program in *Jason*, we informally propose a set of ideas aimed at solving the discussed design problems and programming language issues.

1 Introduction

Jason [6] is an agent-oriented programming system implementing the agent programming language *AgentSpeak(L)* [14]. *AgentSpeak(L)* was proposed as a theoretical language, an articulation and operationalization of the Bratman's Belief-Desire-Intention architecture [7]. *Jason* is nowadays one of the most prominent approaches in the group of theoretically-rooted agent-oriented programming languages (APLs). Building on the foundations of formal logics, these languages serve as vehicles for study of both theoretical issues in agent systems (language features, generic programming constructs, reasoning, coordination, etc.), as well as practical aspects of their design and implementation (e.g., modularity, design, debugging, or code maintenance). To enable program verification, or model checking for more rigorous reasoning about agent programs, *Jason*, together with the majority of APLs in this class, e.g., *2APL*, *3APL*, *GOAL*, *Jazzyk*, *Golog* (for

an overview consult e.g., [3,5,4]) puts a strong emphasis on their rooting in computational logic and rigorous formal semantics. Unlike the more pragmatic approaches, such as *Jadex*, or *JACK* (cf. [13,15]), these languages, including *AgentSpeak(L)*, were constructed from scratch which also led to serious shortcomings with respect to practicality of their use.

While on one hand, pragmatic problems of agent design and implementation, such as e.g., code modularity, are gaining a more prominent role in the research community, on the other, the feedback on practical use of such APLs in more elaborated settings is rather scarce. *AgentSpeak(L)* often serves as a basic APL for various extensions and integration with 3rd party tools, however, little is reported in the research community on its practical applications, be it more involved applied research projects, or more significant close-to-real-world applications (cf. also the *Jason* related projects website [11]). The only report on pragmatic issues faced when using *Jason* in a more involved context is the recent study by Madden and Logan [12] in which the authors deal with problems of modularity in their application and in turn propose a corresponding improvements of the language itself. At the same time, to our knowledge, the most elaborated applications of the *Jason* programming system include the entries to the *Multi-Agent Programming Contest*, which already witnessed 8 submissions in years 2006-2010 altogether by three independent research groups. The reports on development of these applications, however, do not include discussion of the practical issues of agent program implementation, but rather focus on the analysis and design issues with an emphasis on the multi-agent coordination.

In this paper we discuss our experiences gathered during an experiment aiming at development of a non-trivial case-study agent application by a novice *Jason* programmer. The main goal of the undertaking was exploration of basic problems in multi-agent coordination in a simple simulated environment using the *Jason* programming system. In particular, we implemented an application involving a team of 8 agents collaboratively exploring a grid maze and subsequently traversing the environment while cooperatively maintaining a formation. Our experiment aimed at a naive, relatively conservative, use of the *Jason* programming system. I.e., we tried to use the programming language *as is*, with as few customisations of the *Jason* interpreter as possible. In contrast, most involved example applications published at the *Jason* project website [11] and submissions to the *AgentContest* employ extensive customisations of the *Jason* interpreter as an inherent part of the system implementation.

The contribution of the presented paper is twofold. Firstly, we provide a structured feedback on the most problematic issues faced while learning to program in *Jason*, so that it will be useful for the wider community involved in the research on agent-oriented programming languages and tools. Secondly, without an ambition to provide conclusive technical solutions, we rather informally propose a set of ideas aimed at solving the discussed design problems and programming language issues.

After a brief introduction into *AgentSpeak(L)* and *Jason* in Section 2 and the description of the implemented case-study (Section 3), in Section 4, the core of

this paper, we discuss a selection of problems we faced during the experiment. For each discussed issue, we firstly motivate and explain the problem on the background of the introduced case-study application, or its extension, and subsequently we discuss the possible solutions. The topics covered in the discussion include implementation of a simple loop design pattern, handling of interactions between several plans and interruptibility thereof, usage of mental notes as local variables in plans and two technical issues arising from implementation of agents embodied in dynamic environment and the unclear boundary between *Jason* programming language itself and its underlying extension/customisation API in *Java*. We conclude the paper by final remarks in Section 5.

2 AgentSpeak(L) and Jason

AgentSpeak(L) is a theoretical agent-oriented programming language introduced by Rao in [14]. It can be seen as a flavour of logic programming implementing the core concepts of the BDI agent architecture, a currently dominant approach to design of intelligent agents. Structurally, an *AgentSpeak(L)* agent is composed of a *belief base* and a *plan library*. The belief base, essentially a set of belief literals, provides the initial beliefs of the agent. The plan library serves as a basis for action selection, as well as for steering the evolution of the agent's mental state over time. The plans of the agent are rules of the form `event : context` $\leftarrow$ `plan`. The rule denotes a plan, a sequence of basic actions and/or subgoals, which is applicable in reaction to the triggering event if the context condition, a conjunction of belief literals, is satisfied.

AgentSpeak(L) agents are reactive planning systems which react to events occurring in their environment, or are generated as subgoals internally by the agent as a result of a deliberative change in its own goals. The dynamics of the agent system is facilitated by i) instantiation of abstract plans as intentions relevant in particular contexts, and subsequently ii) gradual execution of the intentions leading to their subsequent decomposition into more and more concrete subgoal invocations and finally atomic action executions. In each deliberation cycle, such an agent performs the following sequence of steps:

1. *perceive* the environment and update the belief base accordingly
2. *select an event* to handle
3. retrieve all *relevant plans*
4. *select an applicable plan* and *update the intentions* accordingly
5. *select an intention* for further execution
6. *execute one step* of an intention and modify the intention base and the set of events accordingly

Jason is a *Java*-based programming system implementing the *AgentSpeak(L)* with various extensions and includes an integration with several multi-agent middleware platforms such as *JADE*, or *Moise+*. In its original incarnation, *AgentSpeak(L)* is underspecified in several points of the deliberation cycle, namely

how exactly the three selection functions $\mathcal{S}_{\mathcal{E}}$, $\mathcal{S}_{\mathcal{P}}$ and $\mathcal{S}_{\mathcal{I}}$, denoting the selection of events, applicable plans and intentions respectively, are implemented. In *Jason*, these are customizable functions which can be implemented as *Java* methods. Furthermore, *AgentSpeak(L)* disregards the implementation details of agent's interaction with its environment. In particular, the interpreter assumes that the belief base was updated according to agent's percepts at the beginning of each deliberation cycle. *Jason* extends the framework for reasoning about agent's beliefs in that it incorporates a *Prolog* interpreter in the belief base and also provides a toolbox for implementation of custom belief bases meant as a means for representing complex beliefs, such as the topology of environments, or interface to relational databases. Finally, *Jason* provides a framework for implementation of perception handlers and external events as *Java* methods, together with an API for implementation of customised exogenous actions embodying the behaviours of the agent in its implementation.

The customisation interfaces of the *Jason* interpreter provide a means to tailor the deliberation cycle to the domain specific requirements, as well as to improve the efficiency of the agent program execution. Our motivation in the presented experiment was to explore the issues faced in the course of agent program implementation using the vanilla *Jason* interpreter with minimal customisations required to make the implemented agents interact with their environment.

3 The case-study

The *Cows & Cowboys* problem of the *Multi-Agent Programming Contest* editions 2009 and 2010 (cf. [2], scenarios for the 2009-10 editions) is a challenging scenario for benchmarking cooperative multi-agent teams. In the *Cows & Cowboys* scenario, two teams of agents, herders, compete for a shared resource, cows. The environment is a grid, usually a square with a size approximately 100 cells wide. Each cell can be either empty, or can contain an object which can be either a tree, a fence, an agent, or a cow. Trees serve as obstacles in the environment and are arranged so that the freely traversable space forms a kind of a maze. Agents can move between empty cells and are able to open fences located in the environment, by pushing a button at the edge of each fence. Similarly to agents, cows are also able to move between empty cells, however their movement is steered by the environment and takes into account their mutual distances, as well as the distances from the agents and the trees the cow can see. Agents and cows have a limited view, and in each simulation step receive a perception containing cells in their vicinity (agents see a square of 17×17 cells centred at the agent's position, cows see a square of 11×11 cells). The task of each agent team is to herd as many cows as possible into a corral belonging to the team. As cows are afraid when they see an agent, they can be pushed by a coordinated team of agents in a particular direction.

For the purposes of the here reported case-study, we implemented a fragment of the *Cows & Cowboys* scenario. The concrete problem was to implement a team of agents, which cooperatively explore the maze, find some pre-determined

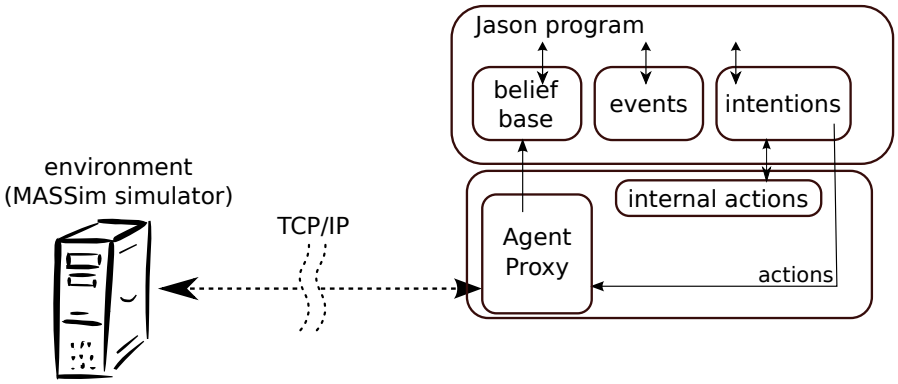


Figure 1. The scheme of the architecture of a single *Jason* agent interacting with the simulated environment.

landmarks and then traverse the maze from one landmark to another while maintaining a formation of a particular shape.

The simulated environment was provided by the MASSim server [1]. The scheme of the implemented system is depicted in Figure 1. The interaction with the simulator was implemented in *Java* by the class AgentProxy derived from the AbstractAgent example stub class provided together with the MASSim package. In each simulation step, the MASSim server sends to each agent an XML message encoding the agent's perception, essentially the content of the cells the agent sees. VisionProcessor, a component of the AgentProxy object, then decodes the XML message and updates the belief base of the agent accordingly.

Upon an update of the belief base, the *Jason* interpreter triggers a set of belief update events, which serve to maintain up-to-date state and consistency of the belief base, as well as to pre-compute answers to some often requested and at the same time computationally-intensive queries. The belief base itself is customised to support unique beliefs. Its implementation replicates the belief base from Gold Miners II example from the standard distribution of *Jason*. Unique beliefs are agent's location, its team mates' position, timestep and similar.

Following the updates of the belief base, an action from the previous timestep, if there was a one, is marked as executed and *Jason* thread continues its deliberation upon new percepts. The AgentProxy control thread then goes to sleep for 2000 milliseconds (the server sends new percepts every 2500 milliseconds) unless it is woken up by the *Jason* thread upon an invocation of an exogenous action from within some intention of the agent. The AgentProxy then validates the action (correct timestep), and if it is valid, it is then translated into the corresponding XML message and sent back to the server. The only exogenous actions the agent can execute are moves in the eight directions: *north*, *east*, *south*, *west* and the diagonal moves *north-east*, *north-west*, *south-east* and *south-west*.

The toolbox of internal actions includes most importantly the implementation of the path planning algorithm A*, together with a few auxiliary functions

such as the lottery-like mechanism for choosing the formation leader, queries for contents of map cells, etc.

One of the most important constraints on the implementation of the case-study was that we do not customise the *Jason* interpreter itself, neither the event, plan and intention selection functions $\mathcal{S}_{\mathcal{E}}$, $\mathcal{S}_{\mathcal{P}}$, $\mathcal{S}_{\mathcal{I}}$.

4 Issues faced

In the following, we discuss a set of problems we encountered in the course of implementing the case-study described above in Section 3 by a programmer learning the *Jason* language along the way. As the authoritative source and documentation, the book *Programming multi-agent systems in AgentSpeak using Jason* [6] was used. For clarity, the discussion of each issue includes a brief motivation and explanation of the particular design problem, subsequently followed by a discussion on the available solutions, their consequences and wherever appropriate an informal proposal for an improved solution to the issue.

4.1 Loop implementation

Often a designer needs to implement some kind of a loop design pattern. E.g., in a maze-like environment, the agent calculates a path from a point A to a point B using some path planning algorithm and then it should follow the path. This design could be implemented by the following algorithm in an imperative language:

```
before−loop−code
while not loop−condition do
        loop−body
end
after−loop−code
```

Jason does not feature a loop programming construct[3], however it can be implemented by the following *Jason* code:

```
event: context ←
        before−loop−plan;
        !loop;
        after−loop−plan.
+!loop: not loop−condition ←
        loop−body;
        !loop.
```

This design implements the idea of tail recursion. However, as in the current versions of *Jason*, the interpreter does not feature a special treatment of tail recursion, according to the language semantics, this design unfolds into an ever growing intention stack. At the bottom of the stack is the plan `after−loop−plan` with a series of invocations of `!loop` of length equal the number of iterations of the loop. In order to facilitate correct plan failure handling, *Jason* interpreter

[3] From the version 1.3.4 *Jason* interpreter includes an explicit loop programming construct. The update was however released only after finishing the here described experiment and the authoritative source on *Jason* [6] does not discuss this issue.

does not remove top-level event invocation from the intention stack. In the path-following scenario, if the path is of length 1000, the intention stack would grow to the size 1000 plus the length of after−loop−plan. In cases with extremely high number of loop iterations, e.g., several dozen thousands path steps is not that extreme for large grid environments, the intention stack growth can lead to high memory consumption, and perhaps more importantly, upon reaching the loop condition, prolonged intention stack cleanup before the interpreter continues with the after−loop−plan.

A naive attempt by a novice programmer could be a loop implementation using the asynchronous event invocation !!loop, straightforward use of which however is inappropriate in this context as besides invoking the loop, it would lead to immediate continuation with the after−loop−plan.

We propose the following implementation of the loop design pattern, which uses higher order variables feature of *Jason* (cf. [6], Chapter 3) to implement a kind of a callback scheme:

```
event: context ←
        before−loop−plan;
        !!loop(after−loop−event).
+!after−loop−event: true ← after−loop−plan
+!loop(Callback): not loop−condition ←
        loop−body;
        !!loop(Callback).
+!loop(Callback): loop−condition ← !!Callback.
```

The above loop implementation is well-formed and a valid program according to the *Jason* syntax and semantics. Instead of a synchronous event invocation, we invoke the loop in an asynchronous manner !!loop and provide it with an argument, which is a string denoting the event which should be invoked after the loop finishes, i.e., after−loop−event. When the loop termination condition becomes true, the pattern simply invokes the event stored as the callback. The advantage of this loop implementation is that it does not lead to intention stack length increase, while at the same time still allows for plan failure handling as in the standard loop implementation.

In the pattern above, the loop recurring event carries with it the appropriate callback to invoke upon the loop's successful termination. An extension of this callback design solution allows a programmer to introduce a powerful plan failure handling mechanism as follows:

```
event: context ←
        before−loop−plan;
        !!loop(after−loop−event, fail−loop−event).
+!after−loop−event: true ←
        after−loop−plan.
+!fail−loop−event: true ←
        loop−failure−plan.
+!loop(SuccessCallback, FailCallback): not loop−condition & loop−continuation−condition ←
        loop−body;
        !!loop(SuccessCallback, FailCallback).
+!loop(_, FailCallback): not loop−condition & not loop−continuation−condition ←
        !!FailCallback.
+!loop(SuccessCallback, _): loop−condition ←
        !!SuccessCallback.
```

Loop is a handy and often used design pattern. However, for a novice programmer, loop implementation in *Jason* is rather unintuitive and its implementation often leads to a confusion. One of the straightforward solutions, well in the spirit of BDI architecture, would be use of persistent goals, such as in 3APL. Another way to deal with this confusion would be to implement a built-in loop programming construct, or a macro pre-processor construction similar to the various types of goals and commitment strategies discussed in [6], Chapter 8.

4.2 Interruptions and plan interactions

Among other desirable properties, intelligent agents are supposed to be able to follow long term goals, but at the same time should be reactive to events in the environment and proactively seek opportunities for action whenever they arise in an appropriate context. Consider the following slight extension of the case-study scenario. The team of agents is moving through the environment in a formation, however, agents are also capable of picking up objects, let's say garbage, from the cells they stand on. Let's also assume, an agent perceives the object to pick, only when it is located in the same cell as the object and it can pick up an object only after it closely inspected it. In *Jason*, a straightforward and naive implementation of the two behaviours would look like as follows:

```
+!formation_loop : not aligned ←
        /* calculate the move action towards formation position */
        move;
        !formation_loop.
+see(Object) : true ←
        inspect(Object);
        pick(Object).
```

The above naive implementation does not work properly using the vanilla *Jason* interpreter. The reason is that after the new intention leading to picking up the object from the cell is formed, it is not ensured that in the same deliberation cycle, the intention selection function $\mathcal{S}_{\mathcal{I}}$ selects the same intention for execution. In the case $\mathcal{S}_{\mathcal{I}}$ selects for execution first the intention for keeping the formation aligned, it can happen that at the moment the agent wants to inspect, or pick up the object, the plan fails since the agent is no more located in the same cell as the object – the plan for keeping the formation aligned moved it away.

The implementation problem described above is that of interacting plans, which can mutually interrupt each other. In *Jason*, similarly to most state-of-the-art BDI-based agent programming languages, plans are considered implicitly interruptible. However, having several plans involved in the same context, i.e., modifying the same aspect of agent's state, which can be instantiated as intentions in parallel, the problem is *how to determine the priority of execution of the corresponding intentions?*.

There are basically three solutions to this problem. The straightforward solution would be to use some kind of plan synchronisation mechanism. *Jason* provides atomic, a pre-defined plan annotation construct ensuring that the intention instantiated from an atomic plan is executed without interruption until it finishes. The following code shows use of the construct:

```
@object_picking[atomic]
+see(Object) : true ←
        inspect(Object);
        pick(Object).
```

While simple and straightforward, this solution of the plan interaction does not scale with the number of involved interacting plans. Consider that our agent should be able to quickly renegotiate the details of formation location and its heading with the team. While interdependent with the formation alignment behaviour, it is independent to the object picking behaviour. In result, we would like to impose the following ordering on the three behaviours: the formation alignment behaviour is preceded by the opportunistic object picking, which is in turn preceded by the negotiation. However, the atomic construct applied to the object picking behaviour would cause it to be non-interruptible, hence the negotiation could not take place.

Another possibility to deal with interacting plans would be to let the program handle the situations, in which they can be interrupted, not the plans themselves. I.e., all plans would be considered implicitly non-interruptible and at every point when a plan can be interrupted by a higher-priority event, there would be an explicit check for all the possibilities of such interruptions, a synchronous invocation of the interrupting event, followed by an explicit check for preconditions of the remaining plan. The following code snippet demonstrates use of such a technique:

```
+!formation_alignment : context ←
        align−plan−start;
        !pick_object; !negotiation;
        align−plan−rest.
+!pick_object : see(Object) ←
        pick−plan−start;
        !negotation;
        pick−plan−rest.
+!negotiation: request(Sender, Msg) ←
        negotiation−plan.
```

Obviously, this technique leads to implementation of agent behaviours in terms of finite state machines and consequently to brittle, non-elaboration-tolerant, code. In order to add a new behaviour, interactions with all the other existing behaviours have to be considered and these have to be modified accordingly.

The only scalable and flexible mechanism solving the problem of interacting plans is the customisation of the intention selection function $\mathcal{S}_{\mathcal{I}}$ in *Java* so that it prioritises the running intentions appropriately according to the particular application domain. The downside of this, rather heavyweight, solution is that it renders the resulting *Jason* program to be not unambiguously readable and understandable in isolation. An important part of the program semantics is this way shifted to the *Java* side and the *Jason* program cannot be fully comprehended without understanding the *Java* code functionality.

Finally, in [6], authors discuss the plan annotation priority reserved for future use. The annotation is intended to instruct the plan selection and intention selection functions $\mathcal{S}_{\mathcal{P}}$ and $\mathcal{S}_{\mathcal{I}}$ about the plan, resp. intention selection priority. However, they also note that the mechanism is not implemented in *Jason*

programming system yet and do not provide enough technical detail on its functionality.

Above, we tried to show that the problem of steering plan interactions and interruptions is an important one, yet not solved appropriately in the current incarnation of *Jason*. On one hand, an intuitive and clean mechanism for plan interaction is vital in BDI-style agent programming, where several plans might be running in parallel and interleave their executions. On the other, plans can interact in too many different ways. To strike balance between the two requirements, as an informal attempt, we call for a conservative extension of *Jason* allowing to impose partial ordering of plans in a program. While certainly not a mechanism general enough (consider e.g., specification of the priority of program modules, similar to the one proposed in [12]), such a mechanism would, in many cases, help avoid customisation of the intention selection function $\mathcal{S}_{\mathcal{I}}$, which we consider a bad design for the reasons discussed above.

4.3 Mental notes and plan destructors

Mental notes are a means for an agent to modify its belief base from its plans in run time. This way the agent can remind itself about status of its own execution and thus partially treat the above discussed problem of plan interactions. Another use of mental notes is transfer of complex information between a behaviour and its invoked subgoals. In result, the mental notes can be used as a kind of local variables of plans. Often, after plan completion, the belief base should be cleaned up, i.e, a programmer would like to retract the set of "local variables" corresponding to the plan. If implemented carefully, *Jason* provides a means to implement such a mechanism. Consider the following code:

```
+!eventX: context ←
        +eventX(note1);
        ...;
        +eventX(note2);
        ...;
        −eventX(_).
```

I.e., each mental note local to the plan triggered by the event eventX is of a particular form, allowing later a bulk retract of all the beliefs in that form from the belief base.

While relatively straightforward, this technique can lead to difficulties in the case of plan failure. Firstly, upon plan failure the local mental notes have to be cleaned up as well, i.e., always when using such mental notes, a plan failure code similar to the following should be used:

```
−!eventX: context ←
        ...;
        −eventX(_).
```

Besides code duplication, a novice *Jason* programmer can simply forget to implement the appropriate plan failure mechanism. Another problem of this technique is that it might be necessary to use different mental note forms for alternative plans handling the event eventX. However, upon plan failure, it is no

longer possible to recognise which particular plan handling the event failed, what can lead to difficulties with the belief base clean up.

We informally propose a language extension similar to exception handling programming construct try−catch−finally present in many imperative languages, as well as in some niche agent programming languages, such as e.g., *StorySpeak* [9]. Consider the following code snippet:

```
+!event: context ←
        try {
                plan−body;
        } finally {
                −eventX(_)
        }.
```

In the finally block, code includes a plan destructor, i.e., a subplan which should be invoked upon plan termination, regardless of its success, or a failure. The advantage of this construct is that the plan destructor is associated with the particular plan variant handling the event unlike when using the standard *Jason* plan failure event invocation −!event .

4.4 Jason agents vs. external environment

In the implemented case-study, the agents had a time limit imposed on the length of their deliberation. In particular, they were allowed 2500ms to decide upon their next actions. If the action was not issued within the timeout, the simulated environment went on as if the agent executed the action skip and discarded any action reply delivered after the timeout. In such environments, it is vital for the agent programmer firstly, to be able to optimise and speed up the agent's deliberation as much as possible, and secondly, to be able to steer the deliberation cycle of the agent from within its plans.

In the implemented case-study, it was necessary for the agent to reason about complex aspects of the environment, such as the form of obstacle structures ahead, fence structures, etc. In order to speed up the deliberation of the agent, we implemented a relatively complex mechanism of belief updating. Upon each belief update, the agent triggered an event to pre-calculate answers to often-queried context conditions and stored them as mental notes in its belief base. While serving the solution, this mechanism led to relatively complex belief base handling within the agent. However, even with this optimisation, it often happened that the implemented agents were not able to reply to the server within the set time limit.

To solve this problem, we propose two extensions of the *Jason* programming system. Prolonged reasoning over the agent's beliefs is often invoked from the rule context conditions (e.g., deliberation over complex aspects of the environment, such as the form of obstacles ahead, path calculation, etc.). In order to speed up such *Prolog* query evaluations, we propose to implement a RETE-style mechanism [8] for context conditions which can be calculated only once and treated as constant queries for the rest of the deliberation cycle. In result, we propose introduction of annotations of rules, or their context conditions, with a

flag denoting their constant value throughout a deliberation cycle, or even until a special belief update event is triggered.

Current implementation of the *Jason* programming system provides the internal action .drop_intention facilitating forceful intention cancellation from within a plan of the agent. The straightforward use of this mechanism is however not well suited for the case-study application. It would require implementation of a recurring goal, a loop like pattern, regularly checking whether the timeout already passed, or not. Another option would be to add the timestep mechanism handling to the environment implementation, annotate the relevant plans with a particular name pattern and finally enhance the agent program with a plan similar to the following one:

```
+timestep: true ←
        .drop_intention(...);
        /* possibly restart some of the intentions afresh */
```

The invocation of the action .drop_intention(...) drops all intentions matching the pattern provided as the argument.

Usage of design solutions such as the two introduced in the previous paragraph, however, would interact with other plans as discussed in Subsection 4.2 and would be difficult without an appropriate customisation of the intention selection function. Secondly, and perhaps more importantly in the case of the first solution, regularly checking the timeout could lead to further slow-down of the deliberation cycle.

We propose an extension of the *Jason* annotation mechanism to include a possibility to annotate agent's intentions with integer values, timesteps. At the point when the system timestep value is incremented, either by the agent program itself, or from within the underlying *Java* code interfacing the agent with the environment, all the intentions annotated with lower timestep value should automatically fail, because they are no longer relevant.

To conclude this part, in its current incarnation, *Jason*, similarly to many other agent-oriented programming languages, is rather *introverted*. In particular, the programming system implicitly assumes that the agent acts in a synchronous manner with respect to the environment. This assumption holds when the speed of the agent's deliberation is relatively higher, or at least matching the rate of change, resp. speed of update, of the environment. However, in cases when the agent deliberation struggles to match the rate of change imposed by the environment, the current implementation of the *Jason* programming system does not provide enough optimisation mechanisms to deal with the issue (we discuss possibilities to deal with this problem in the context of videogame bots in [10]).

4.5 Jason vs. Java

As already remarked above, *Jason* programming system is tightly integrated with the underlying *Java* environment. This setup allows interfacing the implemented agents with their environments in a flexible way, as well as it provides great possibilities with respect to customisation of the language interpreter for

the particular application domain in terms of custom belief bases, and specially tailored event, plan and intention selection functions $\mathcal{S}_{\mathcal{E}}$, $\mathcal{S}_{\mathcal{P}}$ and $\mathcal{S}_{\mathcal{I}}$ respectively.

We argue, that the flexibility of this setup, however, is also a major drawback. As already discussed above, such customisations lead to an unclear boundary between *Java* and *Jason* parts of the implemented agent program. Often, significant and important parts of the agent program functionality are implemented in *Java* code what renders the *Jason* program itself only hardly understandable in isolation.

Another point, especially relevant for novice *Jason* programmers, is the question *what are the guidelines regarding which aspects of the agent program should be implemented in Java and which in Jason?* In an extreme case, this might lead to a trivial Jason program of the form:

```
!main.
+!main: true ← .main.
```

I.e., there is a single event invoked at the start of the program which leads to invocation of an internal action `main` implementing the whole functionality of the agent as a *Java* code. While such a *Jason* program is absurd, it illustrates the point. The possibility to shift pieces of functionality between *Java* and *Jason* and at the same time not having clear guidelines regarding what belongs where, leads to confusion of programmers.

Bordini, Hübner and Wooldridge touch on this issue in [6], Chapter 11. They seem to take a puristic stance, since they argue that programmers should resist the temptation to enhance environments with "fake" actions and other user customisations leading to "cheating" in *Jason* programming. While the point is fair, pragmatic use of the *Jason* language by a relatively inexperienced programmer facing design issues such as those discussed in this section might lead to a series of implementation stages characterised by a growing frustration with the programming system concluded by an escape to the path of "minimal effort", i.e., using a more familiar tool, in this case *Java* programming language.

4.6 Minor technical and methodological issues

Finally, let us conclude the core discourse by listing of some minor issues a programmer learning the *Jason* programming system encounters.

Debugging Debugging of BDI agent systems is a problem known and discussed in the community. Apart from deeper discussion on particular debugging methods, one of the issues are the debugging tools available within the particular programming platform. *Jason* provides a tool for stepping through the agent's reasoning cycle, display its current belief base, the pursued intentions and events awaiting evaluation. Apart from problems with stability of the tool, one of the main difficulties with this style of program debugging is that in situations with relatively short time limit on agent's deliberation, this approach is unusable. A more appropriate technique in such situations is to use a logging facility.

However, in the *Jason* implementation ver. 1.3.3, which has been used for this study, the provided logger does not provide enough information for the programmer. It is not comprehensible enough, as, apart from user defined outputs, it only reports selected events and plans, percepts and execution control messages. It would be useful to export the whole current state of the agent, provided the user is allowed to specify different levels of detail for logging (dynamically during the execution), as output of whole states could be sometimes space intensive.

Integrated Development Environment Even though the provided Eclipse plug-in is reasonably comfortable, it does not follow some of the established patterns for plug-ins of the same category for Eclipse IDE. Instead of adding run options directly to the project options, it has them attached to the context menu of a *mas2j* file. An ordinary Eclipse plug-in would try and replicate the selection of main class of *Java* program, which has essentially the same objective.

Another issues is the lack of code completion function in the standard *Jason* IDE, which rather slows down agent program implementation.

Educational material One of the most difficult aspects of programming in *Jason* was actually learning it. There is only limited material freely available. Thus, along with generated documentation for the source code (javadoc), examples and demos, the most useful resource is the book *"Programming Multi-agent Systems in AgentSpeak Using Jason"* [6]. While the book provides a complete description of the programming system itself, it is still relatively difficult to use it as a pedagogical tool. It imposes a strong emphasis on the theoretical part of *Jason*, without introducing the student into pragmatics of building more complex agent systems. To improve the situation, availability of several authoritative tutorials on incremental building complex agent systems would definitely help to promote the correct programming techniques in the language. As of now, the initial barrier between first working plans and first complex interacting plans is tremendous and requires a lot of trial and error approach on the side of the novice *Jason* programmer. In our opinion, it is much greater than for other languages, such as *Java*, *C++* or *Python*.

5 Final remarks

In the above sections, we discussed some of the most problematic issues we faced during the experiment. In particular, the experiment aimed at implementation of a relatively complex case-study application by a programmer without a prior knowledge of *Jason* language. To keep the experience as relevant to *Jason*-style agent programming as possible, one of the strict prior requirements was to try to use *Jason* programming system as is, i.e., with as few customisations as possible. In particular, we decided not to customise the deliberation cycle of the *Jason* interpreter and the only parts implemented as *Java* code were those facilitating the interaction with the simulated environment, i.e., a set of internal actions

implementing e.g., path planning algorithms and some arithmetic calculations and the customised belief base handling the availability of perceptions received from the simulated environment from within *Jason* code.

Since we used the simulated environment provided in the *Multi-Agent Programming Contest* (*AgentContest*), the complexity of the implemented case-study is directly comparable to the implementations of *AgentContest* entries in its last few editions, which featured the *Cows & Cowboys* scenario, i.e., the same simulated environment. For comparison, our implementation resulted in codebase involving 1127 lines of code, while the *AgentContest* entries to editions 2009 and 2010, presented by teams involving the Jason platform developers, included 1416 and 1648 lines of code respectively. The *AgentContest* entries, however, aimed at the full-featured cows herding scenario, while our case-study implemented only a fragment of the scenario, environment exploration and movement in a formation through the environment. The independent entry to the 2010 edition of the *AgentContest* by the team of the *Technical University of Denmark* featured only 173 lines of *Jason* code and most of the team functionality was thus implemented on *Java* side. If our assumption, that the *AgentContest* entries are the largest, publicly available, applications written to date, is correct, then our case-study resulted in one of the most extensive *Jason* codebases to date.

In parallel to conducting the here reported *Jason* implementation, several students implemented the same case-study application in *Java* in the context of Multi-Agent Systems course at CTU in Prague. Interestingly, while most of them considered the task quite work-intensive and reported a workload in range of 40-60 hours of programming and testing to complete the undertaking, the *Jason* implementation took more than 120 hours to complete for an experienced *Java* programmer. The average *Java* codebase resulting from the exercise involved more than 4000 lines of code. While no hard conclusion can be drawn from this remark, it can serve as an indicator that learning *Jason* on a non-trivial example application is definitely a hard task and the community should also invest more effort in promoting educational material and more extensive tutorials on teaching agent-oriented programming.

The discussion in this paper does not aim at providing a significant scientific contribution. However, we believe that reports, such as this, contribute to the on-going discussion in the community on usefulness, relevance and pragmatics of agent-oriented programming systems, tools and languages, as well as to the future developments of the field.

Acknowledgements We are grateful to Jomi F. Hübner (*Federal University of Santa Catarina*, Brasil) and Jørgen Villadsen (*Technical University of Denmark*) for the permission to study and use the code of their entries to the AgentContest.

Authors of the presented work were supported by the *Czech Ministry of Education* grants MSM6840770038 and MSM0021620838, the *Grant Agency of the Czech Technical University in Prague*, grant SGS10/189/OHK3/2T/13, the *Grant Agency of Czech Republic* grant P103/10/1287 and the *Grant Agency of Charles University in Prague* 0449/2010/A-INF/MFF. Preparing this text and presenting this work was also sup-

ported by the project CZ.2.17/3.1.00/31162 that is financed by the *European Social Fund* and the *Budget of the Municipality of Prague* (for Radek Pibil).

References

1. Tristan M. Behrens, Jürgen Dix, Mehdi Dastani, Michael Köster, and Peter Novák. MASSim: Technical Infrastructure for AgentContest Competition Series. http://www.multiagentcontest.org/, 2009.
2. Tristan Marc Behrens, Jürgen Dix, Mehdi Dastani, Michael Köster, and Peter Novák. Multi-Agent Programming Contest. http://www.multiagentcontest.org/, 2009.
3. Rafael H. Bordini, Lars Braubach, Mehdi Dastani, Amal El Fallah Seghrouchni, Jorge J. Gomez-Sanz, João Leite, Gregory O'Hare, Alexander Pokahr, and Alessandro Ricci. A survey of programming languages and platforms for multi-agent systems. *Informatica*, 30:33–44, 2006.
4. Rafael H. Bordini, Mehdi Dastani, Jürgen Dix, and Amal El Fallah-Seghrouchni, editors. *Multi-Agent Programming: Languages, Tools and Applications*. Springer, Berlin, 2009.
5. Rafael H. Bordini, Mehdi Dastani, Jürgen Dix, and Amal El Fallah Seghrouchni. *Multi-Agent Programming Languages, Platforms and Applications*, volume 15 of *Multiagent Systems, Artificial Societies, and Simulated Organizations*. Kluwer Academic Publishers, 2005.
6. Rafael H. Bordini, Jomi Fred Hübner, and Michael Wooldridge. *Programming Multi-agent Systems in AgentSpeak Using Jason*. Wiley Series in Agent Technology. Wiley-Blackwell, 2007.
7. Michael E. Bratman. *Intention, Plans, and Practical Reason*. Cambridge University Press, March 1999.
8. Charles Forgy. Rete: A fast algorithm for the many patterns/many objects match problem. *Artif. Intell.*, 19(1):17–37, 1982.
9. Jakub Gemrot. Joint behaviour for virtual humans. Master's thesis, Faculty of Mathematics and Physics, Charles University, Prague, 2009.
10. Jakub Gemrot, Cyril Brom, and Tomáš Plch. A periphery of pogamut: From bots to agents and back again. In Frank Dignum, editor, *Agents for Games and Simulations II: Trends in Techniques, Concepts and Design*, volume 6525 of *Lecture Notes in Computer Science*, pages 19–37. Springer Verlag, 2011.
11. Jason Developers. Jason, a Java-based interpreter for an extended version of AgentSpeak. http://jason.sourceforge.net/, 2011.
12. Neil Madden and Brian Logan. Modularity and Compositionality in Jason. In Lars Braubach, Jean-Pierre Briot, and John Thangarajah, editors, *PROMAS*, volume 5919 of *Lecture Notes in Computer Science*, pages 237–253. Springer, 2009.
13. Alexander Pokahr, Lars Braubach, and Winfried Lamersdorf. *Jadex: A BDI Reasoning Engine*, chapter 6, pages 149–174. Volume 15 of *Multiagent Systems, Artificial Societies, and Simulated Organizations* [5], 2005.
14. Anand S. Rao. AgentSpeak(L): BDI Agents Speak Out in a Logical Computable Language. In Walter Van de Velde and John W. Perram, editors, *MAAMAW*, volume 1038 of *Lecture Notes in Computer Science*, pages 42–55. Springer, 1996.
15. Michael Winikoff. *JACK^{TM} Intelligent Agents: An Industrial Strength Platform*, chapter 7, pages 175–193. Volume 15 of *Multiagent Systems, Artificial Societies, and Simulated Organizations* [5], 2005.

Chapter 3

Programming Languages and Platforms

The agent programming language Meta-APL

Thu Trang Doan, Natasha Alechina, and Brian Logan

University of Nottingham, Nottingham NG8 1BB, UK
{ttd,nza,bsl}@cs.nott.ac.uk

Abstract. We describe a novel agent programming language, Meta-APL, and give its operational semantics. Meta-APL allows both agent programs and their associated deliberation strategy to be encoded in the same programming language. We define a notion of equivalence between programs written in different agent programming languages based on the notion of weak bisimulation equivalence. We show how to simulate (up to this notion of equivalence) programs written in other agent programming languages by programs of Meta-APL. This involves translating both the agent program and the deliberation strategy under which it is executed into Meta-APL.

1 Introduction

In this paper we sketch the agent programming language Meta-APL. Its distinguishing feature is that the agent's deliberation strategy can be encoded as part of the agent program. Meta-APL is designed to form part of a platform for verifying multi-agent systems where the agents are implemented in different (BDI-based) agent programming languages. As part of the verification process, the agent programs will be translated into Meta-APL and verification tools designed for Meta-APL will be used to verify the system. Similar approaches have been proposed before, see for example [4]. The distinguishing feature of our approach is that Meta-APL is itself an agent programming language (rather than a special purpose library as in [4]) and that the deliberation strategy of the target agent can be expressed in the Meta-APL program along with the agent program itself. It is clearly necessary to encode the deliberation strategy correctly, as executing the same program under different strategies may give very different results. Other agent programming languages have also been used to program agent deliberation, see for example [1]. However Meta-APL is specifically designed with this capability in mind, for example it is possible to write rules in Meta-APL which match against the contents of the agent's plan base.

This paper describes the first step towards this long term goal, concentrating on the design and the operational semantics of Meta-APL itself. We also define a notion of equivalence or bisimulation between programs written in different agent programming languages. We sketch how programs written in other agent programming languages plus their deliberation strategies can be translated in Meta-APL so that the resulting Meta-APL program is equivalent to the original program together with its deliberation strategy.

The remainder of this paper is organised as follows. In section 2 we introduce the syntax of Meta-APL. In section 3 we define its operational semantics. In section 4 we

define the notion of equivalence between programs and show how to simulate programs written in 3APL [2]. We conclude and outline directions for future work in section 5.

2 Syntax of Meta-APL

The language of beliefs and goals in Meta-APL is similar to that of other BDI-based agent programming languages, for example, 3APL [2], but we assume a propositional language for ease of presentation (to avoid extra notation to do with substitutions etc. The actual implementation has Prolog-like syntax for beliefs).

We assume that beliefs and goals are built using propositional atoms from a finite set *Prop*. Beliefs are either atoms p, or Horn clauses $p \texttt{:-} q_1, \ldots, q_n$. A belief base is a finite set of beliefs. A belief query ϕ is defined as $\phi ::= p \mid \texttt{not}\ p \mid \phi_1\ \texttt{and}\ \phi_2 \mid \phi_1\ \texttt{or}\ \phi_2$, where `not` is negation as failure. Goals are atoms or conjunctions of atoms. A goal base is a finite set of goals.

Before we introduce plans, we define the notion of a plan body. A plan body is a finite sequence of basic actions, test actions, meta-actions and sub-goals, where

- A basic action has the form of $\#a$ where the symbol $\#$ is used to indicate that this is a basic action, and a is the name of the basic action.
- A test action has the form of $?\varphi$ where $?$ is used to indicate that this is a test action, and φ is a belief query.
- A sub-goal has the form of $!g$ where ! is used to indicate that this is a sub-goal, and g is a goal.
- Meta-actions are actions that allow the agent to add and delete beliefs and goals, and to delete and execute applicable plans. The set of meta-actions is as follows:
  - `add-bel`(d) is for adding a belief d into a belief base.
  - `del-bel`(d) is for deleting a belief d from a belief base.
  - `add-goal`(d) is for adding a goal d into a goal base.
  - `del-goal`(d) is for deleting a goal d from a goal base.
  - `del-plan`(i) is for deleting an applicable plan i from a plan base.
  - `exec`(i) is for executing an applicable plan i.
  - `step`(i) is for executing a single step of an applicable plan i.

A plan in Meta-APL represents the triggering conditions and execution state of a plan body. A plan is a tuple of the form (g, b, π, x, π') where:

- g is a goal,
- b is a context query (defined below),
- π is an initial plan-body,
- x is a flag for specifying the state of the plan which can have one of the following values: a (to say that it is active), ex (to say that it is executed), na (to say that it is not active).
- π' is a partially executed plan-body.

A plan base is a finite set of plans.

Context queries are evaluated against the agent's belief and plan bases. In order to define context queries, we first need to introduce the notion of plan-body terms, goal

terms and flag terms. Informally, a plan-body term is a plan-body except that variables may occur where plan-bodies normally are. Given a set *Vars* of variables, the syntax of plan-body terms is as follows:

$$t_\pi ::= X \mid !s \mid ?\varphi \mid \#a \mid \texttt{ma}(d) \mid \texttt{mb}(t_\pi) \mid t_\pi; t'_\pi$$

where $X \in$ *Vars*, $!s$ is a sub-goal, $?\varphi$ is a test action, $\#a$ is a basic action, $\texttt{ma}(d)$ is a meta-action with $\texttt{ma} \in \{\texttt{add-bel}, \texttt{del-bel}, \texttt{add-goal}, \texttt{del-goal}\}$, d is a belief, and $\texttt{mb}(t_\pi)$ is also a meta-action with $\texttt{mb} \in \{\texttt{del-plan}, \texttt{exec}, \texttt{step}\}$.

A goal term is either a variable or a goal. A flag term is either a variable or a flag.

Then, context queries are defined by the following syntax:

$$b ::= X \mid t_1 = t_2 \mid t_1 \neq t_2 \mid \varphi \mid p(t_g, b, t_\pi, t_x, t_{\pi'}) \mid \neg p(t_g, b, t_\pi, t_x, t_{\pi'}) \mid b \& b'$$

where $X \in$ *Vars*, t_1, t_2, t_π and $t_{\pi'}$ are plan-body terms, φ is a belief query, t_g is a goal term, and t_x is a flag term. Evaluation of context queries will be defined in the next section when we define the operational semantics of Meta-APL. Informally, a belief query is evaluated against the belief base in a standard way, $t_1 = t_2$ is used to check if two terms t_1 and t_2 are unifiable, $p(t_g, b', t_\pi, t_s, t_{\pi'})$ means that there is a plan in the plan base that can unify with $(t_g, b', t_\pi, t_s, t_{\pi'})$; and $\neg p(t_g, b', t_\pi, t_s, t_{\pi'})$ means there is no applicable plan in the plan base which can unify with $(t_g, b', t_\pi, t_s, t_{\pi'})$.

A plan (g, b, π, x, π') is effectively identified by the first three components g, b and π. It can not happen that two applicable plans have the same first three components in a plan base at the same time. During the existence of the plan, its components g, b, π stay unchanged.

In Meta-APL, plans are generated by means of rules. A rule has the form:

$$g, b \rightarrow \pi.$$

where g is a goal and is optional, b is a context query where variables are not allowed to occur outside the scope of the atom $p(\ldots)$, and π is a plan-body. When a rule has no goal, we define that its "hidden" goal is $\top$.

3 Operational semantics of Meta-APL

An agent program consists of an initial belief base, an initial goal base and a set of rules. The initial plan base is empty.

A configuration is a tuple of the form $\langle \sigma, \gamma, \Pi, D \rangle$ where σ is a belief base, γ is a goal base, Π is a plan base and D is a phase indicator of the deliberation cycle which can have one of the following values to indicate in which phase the configuration is: UpdatePercept, ApplyRule, and Exec.

Informally, an agent runs by repeatedly performing a deliberation cycle. In the deliberation cycle, there are three main phases: updating percepts (UpdatePercept), matching and applying rules (ApplyRule), and executing executable intentions (Exec). At the beginning of a deliberation cycle, the agent first updates its percepts where the belief base of the agent is updated according to the percepts collected from the environment.

In this phase, the agent also updates its goal base by adding new goals which are received from outside and dropping goals which become achieved after the belief base is updated. In the next phase, the agent looks for applicable rules from the set of rules against the belief base, the goal base and the plan base. Then, an arbitrary applicable rule is applied to add a new applicable plan into the plan base. Notice that this newly added applicable plan may enable or disable the applicability of other rules. The agent then repeats looking for applicable rules and applying them, one by one, until no more are found. This is when the agent switches to the next phase where it executes executable intentions. In this phase, an executable applicable plan is selected and executed. After a plan is executed, the flag of the plan is set to be "ex". The phase is continued until all executable plans are marked with "ex". Then, the phase is changed to Update-Percept for starting a new deliberation cycle and all "ex" applicable plans are changed to "a".

In the following, we discuss each phase of the deliberation cycle in more detail and define the operational semantics by describing transition rules which transform one configuration to another.

First we give a definition of evaluation of beliefs, goals and belief queries. For a belief base σ and a belief or goal d, we say that $\sigma \models_{Pr} d$ iff d is propositionally entailed by σ. For a goal base γ and a goal d, we say that $\gamma \models_g d$ iff d propositionally follows from one of the goals in γ. Finally, for belief query ϕ and a belief base σ, we define $\sigma \models_{naf} \phi$ as follows:

$\sigma \models_{naf} p$ iff $\sigma \models_{Pr} p$.
$\sigma \models_{naf} \neg p$ iff $\sigma \not\models_{Pr} p$.
$\sigma \models_{naf} \phi_1$ `and` ϕ_2 iff $\sigma \models_{naf} \phi_1$ and $\sigma \models_{naf} \phi_2$.
$\sigma \models_{naf} \phi_1$ `or` ϕ_2 iff $\sigma \models_{naf} \phi_1$ or $\sigma \models_{naf} \phi_2$.

Next we define how to evaluate a context query, where variables can occur only within the scope of the literal $p(\ldots)$, against a configuration $\langle \sigma, \gamma, \Pi, D \rangle$. We write $t = g \mid \theta$ to say that two terms t and g are unifiable by the most general unifier (mgu) θ. When two terms t and g fail to unify, we write $t \neq g$. We evaluate a context query against a configuration $\langle \sigma, \gamma, \Pi, D \rangle$ inductively as follows:

- $\langle \sigma, \gamma, \Pi, D \rangle \models \varphi \mid \emptyset$ iff $\sigma \models_{naf} \varphi$ where $\emptyset$ is used to denote an empty substitution
- $\langle \sigma, \gamma, \Pi, D \rangle \models t_1 = t_2 \mid \theta$ iff the two terms t_1 and t_2 are unifiable by the mgu θ, that is $t_1 = t_2 \mid \theta$.
- $\langle \sigma, \gamma, \Pi, D \rangle \models t_1 \neq t_2 \mid \emptyset$ iff the two terms t_1 and t_2 are not unifiable.
- $\langle \sigma, \gamma, \Pi, D \rangle \models p(t_g, b', t_\pi, t_s, t_{\pi'}) \mid \theta$ iff there exists an applicable plan $i \in \Pi$ such that $p(t_g, b', t_\pi, t_s, t_{\pi'}) = i \mid \theta$.
- $\langle \sigma, \gamma, \Pi, D \rangle \models \neg p(t_g, b', t_\pi, t_s, t_{\pi'}) \mid \emptyset$ iff for all applicable plans $i \in \Pi$, we have that $p(t_g, b', t_\pi, t_s, t_{\pi'}) \neq i$.
- $\langle \sigma, \gamma, \Pi, D \rangle \models b_1 \& b_2 \mid \theta$ iff $\langle \sigma, \gamma, \Pi, D \rangle \models b_1 \mid \theta$ and $\langle \sigma, \gamma, \Pi, D \rangle \models b_2 \mid \theta$

Given a plan base Π, in order to determine which plans can be executed, we define the relation ">" over plans in Π as follows. Given

$$i_1 = (g_1, b_1, \pi_1, s_1, \pi_1')$$
$$i_2 = (g_2, b_2, \pi_2, s_2, \pi_2')$$

we say that $i_1 > i_2$ iff $p(g_2, b_2, \pi_2, \_, \_)$ occurs in b_1 (is a subformula of b_1). We use the notation underscore $\_$ as in Prolog for representing any value. This means i_1 is created because of the existence of i_2 and we shall call i_1 to be the meta plan (of i_2). In each deliberation cycle of the agent, we only execute the maximal meta applicable plan, which could indirectly lead to the execution of the lower meta plans through the help of the meta-actions for executing intentions.

Then, we define the set of active plans (those with the flag to be a), the set of roots (the most meta applicable plan), the set of leafs (the least meta applicable plans or the object applicable plans), and the set of executable applicable plans (only roots which are not leafs are allowed to execute), respectively, with respect to a plan base Π as follows:

$$\begin{aligned} active(\Pi) &= \{(g, b, \pi, a, \pi') \in \Pi\} \\ root(\Pi) &= \{i \in active(\Pi) \mid \nexists i' \in active(\Pi) : i' > i\} \\ leaf(\Pi) &= \{i \in active(\Pi) \mid \nexists i' \in active(\Pi) : i > i'\} \\ executable(\Pi) &= root(\Pi) \setminus leaf(\Pi) \end{aligned}$$

Before defining transition rules for the operational semantics of Meta-APL, let us model an environment by two functions `env_percept` and `env_perform`. The effect of each function is as follows:

- `env_percept`(σ, γ, e) takes a belief base σ, a goal base γ, and the environment e as arguments. This function returns a pair of an updated belief base and an updated goal base. The current implementation assumes that percepts are atomic formulas, and the updated belief base is obtained by adding new beliefs and removing incorrect beliefs according to the percepts from the environment. Likewise, the updated goal base is obtained by adding new goals and removing achieved goals, also, according to the percepts from the environment.
- `env_action`(α, e) takes a basic action and the environment as argument. This function performs the action on the environment. For the moment, we assume that this function returns the value *true* or *false* where it only returns *true* iff the basic action is supported by the environment.

At the beginning of a deliberation cycle, an agent always updates its belief base and goal base. The transition rule for the phase of updating percepts is as follows.

$$\frac{\texttt{env\_percept}(\sigma, \gamma) = (\sigma', \gamma')}{\langle \sigma, \gamma, \Pi, \mathsf{UpdatePercept} \rangle \rightarrow \langle \sigma', \gamma', \Pi, \mathsf{ApplyRule} \rangle} \quad (1)$$

The transition above expresses the phase UpdatePercept where the belief base and goal base are updated with the percepts. Apart from the belief base and the goal base being updated, the phase indicator also changes from UpdatePercept to ApplyRule so that the agent can start the next phase.

We say that a rule $g, b \rightarrow \pi$ is applicable with respect to a configuration $\langle \sigma, \gamma, \Pi, D \rangle$ by a substitution θ iff the following conditions hold:

- $\gamma \models_g g$,

– $\langle \sigma, \gamma, \Pi, D \rangle \models b \mid \theta$,
– There is no applicable plan $(g, b\theta, \pi\theta, X, \pi') \in \Pi$ where $\pi' \neq \epsilon$.

Let $Applicable(\langle \sigma, \gamma, \Pi, D \rangle)$ be the set of pairs of (ρ, θ) where ρ is an applicable rule with respect to $\langle \sigma, \gamma, \Pi, D \rangle$ and θ is the corresponding substitution. Moreover, the last condition is for avoiding the case when a rule may be fired more than once to produce the same applicable plan. The transitions rules for the phase ApplyRule are as follows:

$$\frac{\exists((g, b \rightarrow \pi), \theta) \in Applicable(\langle \sigma, \gamma, \Pi, \mathsf{ApplyRule} \rangle)}{\langle \sigma, \gamma, \Pi, \mathsf{ApplyRule} \rangle \rightarrow \langle \sigma, \gamma, \Pi \cup \{(g, b\theta, \pi\theta, a, \pi\theta)\}, \mathsf{ApplyRule} \rangle} \tag{2}$$

The phase will change to the next one when there are no more applicable rules.

$$\frac{Applicable(\langle \sigma, \gamma, \Pi, \mathsf{ApplyRule} \rangle) = \emptyset}{\langle \sigma, \gamma, \Pi, \mathsf{ApplyRule} \rangle \rightarrow \langle \sigma, \gamma, \Pi, \mathsf{Exec} \rangle} \tag{3}$$

Notice that programmers have the responsibility to make sure that the loop of applying applicable rules terminates. A neglectful design of rules can easily cause the phase ApplyRule to run forever, for example if one of the rules is $p(G, B, \Pi, a, X) \rightarrow \texttt{exec}(G, B, \Pi, a, X)$.

In the phase of executing applicable plans, we execute the first step of every executable plan (those which are active, root and not leaf). Let us define transition rules corresponding to each type of the first step as follows.

The following rule is for executing a basic action.

$$\frac{i = (g, b, \pi, a, \#\alpha; \pi') \in executable(\Pi) \text{ and } \texttt{env\_action}(\alpha) = true}{\langle \sigma, \gamma, \Pi, \mathsf{Exec} \rangle \rightarrow \langle \sigma, \gamma, \Pi \setminus \{i\} \cup \{(g, b, \pi, ex, \pi')\}, \mathsf{Exec} \rangle} \tag{4}$$

When the basic action is not allowed (or supported) by the environment, the applicable plan is put into the inactive state as follows:

$$\frac{i = (g, b, \pi, a, \#\alpha; \pi') \in executable(\Pi) \text{ and } \texttt{env\_action}(\alpha) = false}{\langle \sigma, \gamma, \Pi, \mathsf{Exec} \rangle \rightarrow \langle \sigma, \gamma, \Pi \setminus \{i\} \cup \{(g, b, \pi, na, \#\alpha; \pi')\}, \mathsf{Exec} \rangle} \tag{5}$$

The test action simply checks if the belief query is true against the belief base.

$$\frac{i = (g, b, \pi, a, ?\varphi; \pi') \in executable(\Pi) \text{ and } \sigma \models_{naf} \varphi}{\langle \sigma, \gamma, \Pi, \mathsf{Exec} \rangle \rightarrow \langle \sigma, \gamma, \Pi \setminus \{i\} \cup \{(g, b, \pi, ex, \pi')\}, \mathsf{Exec} \rangle} \tag{6}$$

When it is not, the test action is not removed from the applicable plan.

$$\frac{i = (g, b, \pi, a, ?\varphi; \pi') \in executable(\Pi) \text{ and } \sigma \not\models_{naf} \varphi}{\langle \sigma, \gamma, \Pi, \mathsf{Exec} \rangle \rightarrow \langle \sigma, \gamma, \Pi \setminus \{i\} \cup \{(g, b, \pi, na, ?\varphi; \pi')\}, \mathsf{Exec} \rangle} \tag{7}$$

We execute a sub-goal by simply leaving it there and the programmer needs to define a suitable rule to process the subgoal.

$$\frac{i = (g, b, \pi, a, !h; \pi') \in executable(\Pi)}{\langle \sigma, \gamma, \Pi, \mathsf{Exec} \rangle \rightarrow \langle \sigma, \gamma, \Pi \setminus \{i\} \cup \{(g, b, \pi, ex, !h; \pi')\}, \mathsf{Exec} \rangle} \tag{8}$$

The following rule is for the case of the meta-action for deleting beliefs:

$$\frac{i = (g, b, \pi, a, \texttt{del-belief}(d); \pi') \in executable(\Pi)}{\langle \sigma, \gamma, \Pi, \mathsf{Exec} \rangle \rightarrow \langle \sigma \setminus \{d\}, \gamma, \Pi \setminus \{i\} \cup \{(g, b, \pi, ex, \pi')\}, \mathsf{Exec} \rangle} \tag{9}$$

The effect of the above rule is to remove the belief d from the belief base. Similarly, we have the following rules for the case for adding a new belief.

$$\frac{i = (g, b, \pi, a, \texttt{add-belief}(d); \pi') \in executable(\Pi)}{\langle \sigma, \gamma, \Pi, \mathsf{Exec} \rangle \rightarrow \langle \sigma \cup \{d\}, \gamma', \Pi \setminus \{i\} \cup \{(g, b, \pi, ex, \pi')\}, \mathsf{Exec} \rangle} \tag{10}$$

where $\gamma' = \gamma \setminus \{g \in \gamma \mid \sigma \cup \{d\} \models g\}$. Besides the effect of adding new beliefs, we also remove achieved goals from the goal base. The following rule is for deleting a goal from the goal base:

$$\frac{i = (g, b, \pi, a, \texttt{del-goal}(d); \pi') \in executable(\Pi)}{\langle \sigma, \gamma, \Pi, \mathsf{Exec} \rangle \rightarrow \langle \sigma, \gamma \setminus \{d\}, \Pi \setminus \{i\} \cup \{(g, b, \pi, ex, \pi')\}, \mathsf{Exec} \rangle} \tag{11}$$

Similarly, we have the following rule for adding a new goal into the goal base:

$$\frac{i = (g, b, \pi, a, \texttt{add-goal}(d); \pi') \in executable(\Pi)}{\langle \sigma, \gamma, \Pi, \mathsf{Exec} \rangle \rightarrow \langle \sigma, \gamma', \Pi \setminus \{i\} \cup \{(g, b, \pi, ex, \pi')\}, \mathsf{Exec} \rangle} \tag{12}$$

where $\gamma' = \gamma \cup \{d\}$ iff $\sigma \not\models d$; otherwise, $\gamma' = \gamma$. This means d is added into the goal base only when it is not an achieved goal.

The next transition rule is for deleting an applicable plan.

$$\frac{i = (g, b, \pi, a, \texttt{del-plan}(i'); \pi') \in executable(\Pi)}{\langle \sigma, \gamma, \Pi, \mathsf{Exec} \rangle \rightarrow \langle \sigma, \gamma, \Pi \setminus \{i, i'\} \cup \{(g, b, \pi, ex, \pi')\}, \mathsf{Exec} \rangle} \tag{13}$$

Then, we define the transition rules for the meta-actions for executing applicable plans. In principle, the "exec" meta-action makes similar effect as executing the first step of an intention.

For convenience, we also define a different transition rule, denoted as $\rightarrow^i$ for executing an active applicable plan i which is not required to be a root, but has to be active (i.e. the flag is a). The definitions are the repetition of those above for the execution phase where we replace the condition $i \in executable(\Pi)$ by $i \in active(\Pi)$. Then for every transition rule of the form $\langle \sigma, \gamma, \Pi, \mathsf{Exec} \rangle \rightarrow \langle \sigma', \gamma', \Pi \setminus \{i\} \cup\{\ i'\}, \mathsf{Exec} \rangle$, we also define:

$$\frac{i \in active(\Pi)}{\langle \sigma, \gamma, \Pi, \mathsf{Exec} \rangle \rightarrow^i \langle \sigma', \gamma', \Pi \setminus \{i\} \cup \{i'\}, \mathsf{Exec} \rangle} \tag{14}$$

Notice that the transition rules $\rightarrow^i$ are not the operational semantics of Meta-APL but we use them as auxiliary transition rules for defining the operational semantics of the meta actions `exec` and `step` of Meta-APL. We shall define the transition rule for the meta-action $exec(i')$ based on $\rightarrow^{i'}$ as follows:

$$\frac{\begin{array}{c} i = (g, b, \pi, a, \texttt{exec}(i'); \pi') \in executable(\Pi) \text{ and} \\ \langle \sigma, \gamma, \Pi, \mathsf{Exec} \rangle \rightarrow^{i'} \langle \sigma, \gamma', \Pi \setminus \{i'\} \cup \{i''\}, \mathsf{Exec} \rangle \end{array}}{\langle \sigma, \gamma, \Pi, \mathsf{Exec} \rangle \rightarrow \langle \sigma', \gamma, \Pi \setminus \{i, i'\} \cup \{i'', (g, b, \pi, ex, \pi'')\}, \mathsf{Exec} \rangle} \tag{15}$$

where $\pi'' = \texttt{exec}(i''); \pi'$ if i'' is not an empty plan; otherwise $\pi'' = \pi'$. The above rule means that if we have an executable applicable plan which starts with $\texttt{exec}(i')$, and i' can be executed by means of $\rightarrow^{i'}$ to become i'', then $\texttt{exec}(i')$ means to execute i' and to change to $\texttt{exec}(i'')$. The semantics $\texttt{step}(i')$ is similar to $\texttt{exec}(i')$ except that the action $\texttt{step}(i')$ does not execute the remainder i'' of i'. The transition rule for this meta-action is as follows:

$$\frac{\begin{array}{c} i = (g, b, \pi, a, \texttt{step}(i'); \pi') \in \mathit{executable}(\Pi) \text{ and} \\ \langle \sigma, \gamma, \Pi, \mathsf{Exec} \rangle \rightarrow^{i'} \langle \sigma', \gamma', \Pi \setminus \{i\} \cup \{i''\}, \mathsf{Exec} \rangle \end{array}}{\langle \sigma, \gamma, \Pi, \mathsf{Exec} \rangle \rightarrow \langle \sigma, \gamma, \Pi \setminus \{i, i'\} \cup \{i'', (g, b, \pi, ex, \pi')\}, \mathsf{Exec} \rangle} \tag{16}$$

In both cases of the transition rules for exec and step, if $i' \notin \Pi$ or is inactive, then i also becomes an inactive plan.

$$\frac{i = (g, b, \pi, a, \texttt{exec}(i'); \pi') \in \mathit{executable}(\Pi) \text{ and } i' \notin \mathit{active}(\Pi)}{\langle \sigma, \gamma, \Pi, \mathsf{Exec} \rangle \rightarrow \langle \sigma, \gamma, \Pi \setminus \{i\} \cup \{(g, b, \pi, na, \texttt{exec}(i'); \pi')\}, \mathsf{Exec} \rangle} \tag{17}$$

We also have:

$$\frac{i = (g, b, \pi, a, \texttt{step}(i'); \pi') \in \mathit{executable}(\Pi) \text{ and } i' \notin \mathit{active}(\Pi)}{\langle \sigma, \gamma, \Pi, \mathsf{Exec} \rangle \rightarrow \langle \sigma, \gamma, \Pi \setminus \{i\} \cup \{(g, b, \pi, na, \texttt{step}(i'); \pi')\}, \mathsf{Exec} \rangle} \tag{18}$$

Notice that the new definition of the transition rules $\rightarrow$ for the meta-actions `exec` and `step` also implicitly gives extra definitions of the transition rule $\rightarrow^i$. This helps us to define further transition rules for nested meta-actions `exec` and `step`.

Then, when there is no more executable plans, the phase turns back to UpdatePercept for a new deliberation cycle. All plans have the flag "ex" are also changed to "a" for further execution in the next deliberation cycle. We also have:

$$\frac{\mathit{executable}(\Pi) = \emptyset}{\langle \sigma, \gamma, \Pi, \mathsf{Exec} \rangle \rightarrow \langle \sigma, \gamma, \Pi', \mathsf{UpdatePercept} \rangle} \tag{19}$$

where $\Pi' = (\Pi \setminus \{(g, b, \pi, ex, \pi') \in \Pi\}) \cup \{(g, b, \pi, a, \pi') \mid (g, b, \pi, ex, \pi') \in \Pi \wedge \pi' \neq \epsilon\} \cup \{(g, b, \pi, na, \epsilon) \mid (g, b, \pi, ex, \epsilon) \in \Pi\}$.

In the above transition rule, when transiting from the phase Exec back to UpdatePercept in the deliberation cycle, all applicable plans in the plan base with the flag being ex are changed back to a so that they are ready for further execution, except those have an empty plan (denoted as ϵ) which are changed to the state inactive.

4 Simulating 3APL

In this section we show how to translate agent programs of 3APL into Meta-APL. Both languages share similar features, but have different deliberation cycles. We show how meta rules can be used to simulate deliberation cycle of other languages in Meta-APL.

First we define what we mean by simulating one program by another program. We use the concept of *weak bisimulation* [5]. We treat transitions other that basic actions as *internal* or τ actions. By a *run of a program* we will mean a sequence

$s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \ldots$ where s_i are agent's configurations and a_i are transitions of the agent's operational semantics which are either basic actions or other internal τ transitions. Two runs $r = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \ldots$ and $r' = s'_0 \xrightarrow{a_0} s'_1 \xrightarrow{a_1} s'_2 \ldots$ are *equivalent* if there is a symmetric relation R (later referred to as equivalence) between the configurations in r_1 and r_2 such that:

- $R(s_0, s'_0)$
- if $R(s, s')$ then the agent's beliefs about environment in s and s' are the same
- if $R(s, s')$ and $s \xrightarrow{\tau^*} t_1 \xrightarrow{a} t_2$ in r, then $s' \xrightarrow{\tau^*} t'_1 \xrightarrow{a} t'_2$ in r' and $R(t_2, t'_2)$, where τ^* is a sequence of 0 or finitely many internal transitions, and a is the first basic action occurring after s

Intuitively, $R(s, s')$ means that in configurations s and s' the agent has the same beliefs and goals.

We say that a program p_1 simulates another program p_2 iff:

- For every run r_1 of p_1, there is a run r_2 of p_2 such that r_1 and r_2 are equivalent.
- For every run r_2 of p_2, there is a run r_1 of p_1 such that r_2 and r_1 are equivalent.

In this paper we show how to translate a program p_2 of some agent programming language into a program p_1 of Meta-APL so that p_1 simulates p_2.

4.1 3APL

In this paper, we refer to 3APL as the agent programming language which was presented in [2]. Since the version of Meta-APL introduced in this paper for simplicity only allows propositional beliefs and goals, we show how to simulate propositional 3APL programs, but the extension to full 3APL beliefs and goals is straightforward. Moreover, we also slightly modify 3APL in order to omit the plan constructs `if-then-else` and `while-do`.

An agent in 3APL can have three types of rules:

- PG (plan generation) rule: $g \leftarrow b \mid \pi$
- GR (goal revision) rule: $g \leftarrow b \mid g'$
- PR (plan revision) rule: $\pi \leftarrow b \mid \pi'$

Where g and g' are goals, b is a belief query, π and π' are plans. A plan π is a sequence of basic actions, test actions and abstract plans. Moreover, test actions are of the form $B(\varphi)$ only. Branching and looping constructs (if-then-else and while-do) in a plan are not allowed as they can be translated into rules in 3APL using abstract plans. Note that omitting `if-then-else` and `while-do` in plans does not reduce the expressiveness of 3APL as they can be represented by abstract plans by using PG and PR rules. For example, the following PG rule of 3APL:

$$g \leftarrow b \mid \pi; \texttt{if } b' \texttt{ then } \pi_1 \texttt{ else } \pi_2 \texttt{ end-if}; \pi'.$$

is translated into an abstract plan by one PG rule and two PR rules as follows:

$$\begin{aligned} g &\leftarrow b \mid \pi; abs; \pi'. \\ abs &\leftarrow b' \mid \pi_1. \\ abs &\leftarrow \neg b' \mid \pi_2. \end{aligned}$$

Similarly, the `while-do` construct

$$g \leftarrow b \mid \pi; \texttt{while } b' \texttt{ do } \pi_1 \texttt{ end-while}; \pi'.$$

is translated into abstract plans as follows

$$\begin{aligned} g &\leftarrow b \mid \pi; abs; \pi'. \\ abs &\leftarrow b' \mid \pi_1; abs. \\ abs &\leftarrow \neg b' \mid \epsilon. \end{aligned}$$

In order to prove the correctness of the simulation, we need to show the equivalence between runs in 3APL and those by the simulation. Firstly, we specify a deliberation cycle of 3APL. A basic deliberation cycle of 3APL is as follows:

1. Apply applicable rules.
2. Execute plans.

However, we have not defined in detail each of the above two stages. In the stage of applying applicable rules, two extreme approaches are either to apply only one rule, or to apply all the rules until no more are applicable (more precisely, compute a set of applicable rules, if it is not empty, choose a rule and apply it, recompute the set of applicable rules, etc. until the set of applicable rules is empty). In the stage of plan execution, a common approach is to pick a plan and to execute one step. If plans for executing a step are selected randomly, we call this an interleaved execution method. Otherwise, if a selected plan is executed completely before any other plan's steps are selected, we called this a non-interleaved execution method.

In order to demonstrate how to simulate 3APL by means of Meta-APL, we choose the following approach to define how the rule application is done:

1. Apply all applicable PG rules (in any order).
2. Continuously pick an applicable PR rule and apply until no more are applicable.

The two different methods of plan executions are considered in the next sections.

4.2 Simulating interleaved deliberation cycle of 3APL

Firstly, we translate a plan π in 3APL into Meta-APL by translating each element of π. Let us denote the translated plan as $tr(\pi)$.

We translate each type of rules into the corresponding ones in Meta-APL as follows:

- A PG rule $g \leftarrow b \mid \pi$ is translated into: $g, b \rightarrow tr(\pi)$.
- A GR rule $g \leftarrow b \mid g'$ is translated into the following rule:

  $$g, b \,\&\, \neg p(g, \_, \texttt{del-goal}(g); X, a, \_) \rightarrow \texttt{del-goal}(g); \texttt{add-goal}(g').$$

  The above rule is for replacing a goal g in the goal base with another goal g'. Comparing to the original rule in 3APL, the guard of the translated rule has an extra condition $\neg p(\ldots)$ which is for preventing from applying the same rule and other rules for revising the same goal g once this rule has been fired.

– A PR rule $\pi \leftarrow b \mid \pi'$ is translated into the following rule:

$$p(G, B, P, a, tr(\pi)) \;\&\; b \;\&\; \\ \neg p(G', p(G, B, P, a, tr(\pi)) \;\&\; B', P', a, \texttt{del-plan}(G, B, P, a, tr(\pi)); X) \\ \rightarrow \texttt{del-plan}(G, B, P, a, tr(\pi)); tr(\pi').$$

The above rule is for revising a plan $tr(\pi)$ in the plan base with another plan $tr(\pi')$. Similar to the case of GR rules, the guard of the translated rule also has an extra condition $\neg p(\ldots)$ which is for preventing from applying the same rule and other rules for revising the same plan $tr(\pi)$ once this rule has been fired.

Then, we implement the interleaved deliberation cycle. Below are the rules for implementing the interleaved deliberation cycle of 3APL:

$$\neg p(G1, B1, P1, a, \texttt{step}(G2, B2, P2, a, Y)) \;\&\; p(G3, B3, P3, a, X) \\ \rightarrow \texttt{step}(G3, B3, P3, a, X).$$

$$p(G1, B1, P1, a, \texttt{step}(G2, B2, P2, a, X)) \\ \;\&\; p(G3, B3, P3, a, \texttt{del-plan}(G2, B2, P2, a, X); X') \\ \rightarrow \texttt{del-plan}(G1, B1, P1, a, \texttt{step}(G2, B2, P2, a, X)).$$

$$p(G1, B1, P1, a, \texttt{step}(G2, B2, P2, a, X)) \\ \;\&\; p(G3, B3, P3, a, \texttt{del-plan}(G2, B2, P2, a, X); X') \\ \rightarrow \texttt{step}(G3, B3, P3, a, \texttt{del-plan}(G2, B2, P2, a, X); X').$$

$$p(G1, B1, P1, a, \texttt{step}(G2, B2, P2, a, X)) \\ \;\&\; X \neq \texttt{del-plan}(G3, B3, P3, a, Y); Y' \\ \;\&\; \neg p(G4, B4, P4, a, \texttt{del-plan}(G2, B2, P2, a, X); X') \\ \;\&\; p(G5, B5, P5, a, \texttt{del-plan}(G6, B6, P6, a, Z); Z') \\ \rightarrow \texttt{step}(G5, B5, P5, a, \texttt{del-plan}(G6, B6, P6, a, Z); Z').$$

$$p(G1, B1, P1, a, \texttt{step}(G2, B2, P2, a, X)) \\ \;\&\; X \neq \texttt{del-plan}(G3, B3, P3, a, Y); Y' \\ \;\&\; \neg p(G4, B4, P4, a, \texttt{del-plan}(G2, B2, P2, a, X); X') \\ \;\&\; p(G5, B5, P5, a, \texttt{exec}(G6, B6, P6, a, Z)) \\ \rightarrow \texttt{step}(G5, B5, P5, a, \texttt{exec}(G6, B6, P6, a, Z)).$$

We also have the following rule for executing the rules which translate GR rules:

$$p(G1, B1, P1, a, \texttt{del-goal}(G); Z) \rightarrow \texttt{exec}(G1, B1, P1, a, \texttt{del-goal}(G); Z).$$

The first rule selects an arbitrary plan in the plan base for execution but of only one step. Then, once applied, the rule is not applicable any more in that cycle, hence, we

can prevent selecting another plan to apply at the same cycle. When a plan is selected to execute but it is also selected to be revised by another rule, we transfer the selection to the newly revised plan by means of the next two rules. Finally, the last two rules are there blocking the above selection when there are the application of rules to revise plans or goals where they generate plans starting with either `del-plan` or `exec`.

Let us revisit the example in [3] as an illustration of how the translation from 3APL into Meta-APL works. The example is about moving blocks on a floor to a desired configuration by an agent which has the power to put a block which has nothing on the top on the floor or on top of other block which also has no block on top. In the example, there are three blocks namely a, b and c. The initial setting is that a and b are on the floor while c is on top of a. The desired setting is that c is on the floor, b is on c and a is on b.

In 3APL, the program of the agent is as follows:

- Belief base: $on(a, floor)$, $on(b, floor)$, $on(c, a)$
- Goal base: $on(c, floor) \wedge on(b, c) \wedge on(a, b)$
- Rule base:

$$\begin{aligned} on(X,Y) \leftarrow \neg on(X,Y) \mid\ & clear(X); clear(Y); move(X,Y). \\ clear(X); Z \leftarrow on(Y,X) \wedge X \neq floor \mid\ & clear(Y); move(Y, floor); Z. \\ clear(X); Z \leftarrow \neg on(Y,X) \mid\ & Z. \\ clear(floor); Z \leftarrow \top \mid\ & Z. \end{aligned}$$

The set of rules above is translated into Meta-APL as follows:

$$\begin{aligned} & on(X,Y), \neg on(X,Y) \\ & \quad \rightarrow !clear(X); !clear(Y); \#move(X,Y). \end{aligned}$$

$$\begin{aligned} & p(G,B,P,a,!clear(X);Z)\ \&\ on(Y,X)\ \&\ X \neq floor \\ & \ \&\ \neg p(G', p(G,B,P,a,!clear(X);Z)\ \&\ B', P', a, \\ & \qquad\qquad \texttt{del-plan}(G,B,P,a,!clear(X);Z); Z') \\ & \quad \rightarrow \texttt{del-plan}(G,B,P,a,!clear(X);Z); !clear(Y); \#move(Y, floor); Z. \end{aligned}$$

$$\begin{aligned} & p(G,B,P,a,!clear(X);Z)\ \&\ \neg on(Y,X) \\ & \ \&\ \neg p(G', p(G,B,P,a,!clear(X);Z)\ \&\ B', P', a, \\ & \qquad\qquad \texttt{del-plan}(G,B,P,a,!clear(X);Z); Z') \\ & \quad \rightarrow \texttt{del-plan}(G,B,P,a,!clear(X);Z); Z. \end{aligned}$$

$$\begin{aligned} & p(G,B,P,a,!clear(floor);Z) \\ & \ \&\ \neg p(G', p(G,B,P,a,!clear(floor);Z)\ \&\ B', P', a, \\ & \qquad\qquad \texttt{del-plan}(!clear(floor);Z); Z') \\ & \quad \rightarrow \texttt{del-plan}(!clear(floor);Z); Z. \end{aligned}$$

Notice that the above rules are in the abbreviated form.

We sketch here the proof that one cycle in 3APL corresponds to one or more cycles in the simulation by Meta-APL (hence the runs of the two programs are equivalent). There are two cases:

1. Consider a cycle in 3APL, if there are no PR and GR rules applicable, at the end of the cycle, a plan is selected for execution. The corresponding run in Meta-APL also contains only a single cycle, as no rules into which PR and GR rules are translated are applicable, there is no plan starting with `del-plan` or `exec`, hence, a plan which is selected for execution of one step is not blocked. The corresponding cycle is chosen by selecting the corresponding plan in the case of 3APL.
2. Consider a cycle in 3APL where some PR or GR rules are applicable, at the stage of applying PR and GR rules, it is repeated until no more PR and GR rules are applicable. We define sequences of PR rule application which are sequences of plans $\pi_0, \ldots, \pi_k$ where π_i is obtained by appling some PR rule to revise π_{i-1} for all $i \geq 1$. We also define a sequence of GR rule application as a sequence of goals $g_0, \ldots, g_m$ where g_i is replaced by applying some GR rule for all $i \geq 0$. Then, let $n + 1$ be the length of the longest sequence among sequences of PR rule applications, then we construct a run in Meta-APL containing $n + m + 1$ cycles where the starting and ending configurations are equivalent to configurations in 3APL before and after the cycle, respectively. In the first cycle, we apply all translated PR rules which are applicable by following the order of PR rule application in 3APL (ignore those which is not applicable yet). Of course, since some PR rule is applied, any plan which is selected for execution is blocked. We repeat this again and again and after n cycles, we must reach a configuration where no more PR rules are applicable. Then, the next cycles are for applying GR rules in the order of the corresponding to the sequence of GR rule application. The final cycle is just for selecting the corresponding plan in 3APL for execution (and it is not blocked as no more PR and GR rules are applicable).

The reverse direction can be shown similarly.

4.3 Simulating non-interleaved deliberation cycle of 3APL

In this section, we simulate the non-interleaved deliberation cycle of 3APL. In this deliberation cycle, we keep executing a plan and any plans which revise this plan until it becomes empty. The implementation of the non-interleaved deliberation cycle is quite similar to the case of interleaved deliberation cycle as in the previous section except that we interchange the use of the meta-actions `step` and `exec`.

In particular, we keep the function tr to translate plans, the translation of PG, GR and PR rules from 3APL to Meta-APL unchanged. The only difference comparing to the case of the interleaved deliberation cycle of 3APL is the implementation of the non-interleaved deliberation cycle. The rules to implement the non-interleaved deliberation cycle of 3APL are as follows:

$$\neg p(G1, B1, P1, a, \texttt{exec}(G2, B2, P2, a, Y)) \;\&\; p(G3, B3, P3, a, X)$$

$$\rightarrow \texttt{exec}(G3, B3, P3, a, X).$$

$$p(G1, B1, P1, a, \texttt{exec}(G2, B2, P2, a, X))$$
$$\& \; p(G3, B3, P3, a, \texttt{del-plan}(G2, B2, P2, a, X); X')$$
$$\rightarrow \texttt{del-plan}(G1, B1, P1, a, \texttt{exec}(G2, B2, P2, a, X)).$$

$$p(G1, B1, P1, a, \texttt{exec}(G2, B2, P2, a, X))$$
$$\& \; p(G3, B3, P3, a, \texttt{del-plan}(G2, B2, P2, a, X); X')$$
$$\rightarrow \texttt{exec}(G3, B3, P3, a, \texttt{del-plan}(G2, B2, P2, a, X); X').$$

$$p(G1, B1, P1, a, \texttt{exec}(G2, B2, P2, a, X))$$
$$\& \; X \neq \texttt{del-plan}(G3, B3, P3, a, Y); Y'$$
$$\& \; \neg p(G4, B4, P4, a, \texttt{del-plan}(G2, B2, P2, a, X); X')$$
$$\& \; p(G5, B5, P5, a, \texttt{del-plan}(G6, B6, P6, a, Z); Z')$$
$$\rightarrow \texttt{step}(G5, B5, P5, a, \texttt{del-plan}(G6, B6, P6, a, Z); Z').$$

$$p(G1, B1, P1, a, \texttt{exec}(G2, B2, P2, a, X))$$
$$\& \; X \neq \texttt{del-plan}(G3, B3, P3, a, Y); Y'$$
$$\& \; \neg p(G4, B4, P4, a, \texttt{del-plan}(G2, B2, P2, a, X); X')$$
$$\& \; p(G5, B5, P5, a, \texttt{step}(G6, B6, P6, a, \texttt{add-goal}(G); Z))$$
$$\rightarrow \texttt{step}(G5, B5, P5, a, \texttt{step}(\texttt{add-goal}(G); Z)).$$

$$p(G1, B1, P1, a, \texttt{exec}(G2, B2, P2, a, X))$$
$$\& \; X \neq \texttt{del-plan}(G3, B3, P3, a, Y); Y'$$
$$\& \; \neg p(G4, B4, P4, a, \texttt{del-plan}(G2, B2, P2, a, X); X')$$
$$\& \; p(G5, B5, P5, a, \texttt{step}(G6, B6, P6, a, \texttt{del-goal}(G); Z)))$$
$$\rightarrow \texttt{step}(G5, B5, P5, a, \texttt{step}(G6, B6, P6, a, \texttt{del-goal}(G); Z))).$$

Similar to the implementation of the interleaved deliberation cycle of 3APL, the first rule is also to select a plan to execute by using the meta action `exec`. Since `exec` is used, this selection of the plan to execute is still kept in the next deliberation cycle if it does not become empty. We also have the next two rules is for changing the selection of a plan to its parents when it is revised by some rule. Finally, the last three rules are also for blocking the selected plan from being executed if some plans or goals are revised. Notice that we have more than one rule comparing to the implementation of the interleaved deliberation cycle of 3APL.

In order to execute the rules which translate GR rules, we have the following rules:

$$p(G1, B1, P1, a, \texttt{del-goal}(G); Z) \rightarrow \texttt{step}(G1, B1, P1, a, \texttt{del-goal}(G); Z).$$
$$p(G1, B1, P1, a, \texttt{add-goal}(G); Z) \rightarrow \texttt{step}(G1, B1, P1, a, \texttt{add-goal}(G); Z).$$

It is straightforward to prove that the translated program in Meta-APL simulates the non-interleaved deliberation cycle of 3APL. The proof is similar to that of the interleaved case in the previous section.

5 Conclusions and future work

We have introduced the syntax and operational semantics of Meta-APL. We have sketched how it can be used to simulate programs written in other agent programming languages together with their operational semantics. In our future work, we plan to develop automatic methods for producing provably equivalent translations of agent programs in Meta-APL and a set of tools for automatically verifying properties of agent systems implemented in Meta-APL.

References

1. M. Dastani, F. de Boer, F. Dignum, and J.J. Meyer. Programming agent deliberation: an approach illustrated using the 3APL language. In *The Second International Joint Conference on Autonomous Agents & Multiagent Systems, AAMAS 2003, July 14-18, 2003, Melbourne, Victoria, Australia, Proceedings*, pages 97–104. ACM, 2003.
2. M. Dastani, F. Dignum, and J.J. Meyer. 3APL: A Programming Language for Cognitive Agents. *ERCIM News, European Research Consortium for Informatics and Mathematics, Special issue on Cognitive Systems*, 2000.
3. Mehdi Dastani, Birna van Riemsdijk, Frank Dignum, and John-Jules Ch. Meyer. A Programming Language for Cognitive Agents Goal Directed 3APL. In Mehdi Dastani, Jürgen Dix, and Amal El Fallah-Seghrouchni, editors, *Programming Multi-Agent Systems, First International Workshop, PROMAS 2003, Melbourne, Australia, July 15, 2003, Selected Revised and Invited Papers*, volume 3067 of *Lecture Notes in Computer Science*, pages 111–130. Springer, 2003.
4. Louise A. Dennis, Berndt Farwer, Rafael H. Bordini, and Michael Fisher. A flexible framework for verifying agent programs. In Lin Padgham, David C. Parkes, Jörg P. Müller, and Simon Parsons, editors, *7th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2008), Estoril, Portugal, May 12-16, 2008, Volume 3*, pages 1303–1306. IFAAMAS, 2008.
5. R.J. Van Glabbeek and W.P. Weijland. Branching time and abstraction in bisimulation semantics. *Journal of the ACM (JACM)*, 43(3):555–600, 1996.

BDI4JADE: a BDI layer on top of JADE

Ingrid Nunes[1,2], Carlos J.P. de Lucena[1], and Michael Luck[2]

[1] PUC-Rio, Computer Science Department, LES - Rio de Janeiro, Brazil
{ionunes,lucena}@inf.puc-rio.br
[2] King's College London, Strand, London, WC2R 2LS, United Kingdom
michael.luck@kcl.ac.uk

Abstract. Several agent platforms that implement the belief-desire-intention (BDI) architecture have been proposed. Even though most of them are implemented based on existing general purpose programming languages, e.g. Java, agents are either programmed in a new programming language or Domain-specific Language expressed in XML. As a consequence, this prevents the use of advanced features of the underlying programming language and the integration with existing libraries and frameworks, which are essential for the development of enterprise applications. Due to these limitations of BDI agent platforms, we have implemented the BDI4JADE, which is presented in this paper. It is implemented as a BDI layer on top of JADE, a well accepted agent platform.

Keywords: Multi-agent Systems, Agent Platforms, Agent Programming, BDI Architecture, BDI4JADE, JADE.

1 Introduction

With the popularity of the web, complex systems has become a reality. These are characterized by being distributed and composed of multiple autonomous entities, which interact with each other. Multi-agent systems are considered a promising approach for developing this kind of systems [17], by decomposing them into agents, each of which with its own thread of execution, possibly a proactive behavior, thus aiming to achieve its individual goals, and able to perceive its surrounding environment and respond in a timely fashion to changes to it. From a software engineering perspective, multi-agent systems can be seen as a paradigm in which systems are decomposed into autonomous and proactive software components, namely agents.

Due to the complexity associated with the development of multi-agent systems, which typically involves thread control, message exchange across the network, cognitive ability, and discovery of agents and their services, several architectures and platforms have been proposed. One of the widely known architectures for designing and implementing cognitive agents is the belief-desire-intention (BDI) architecture, following a model initially proposed by Bratman [3], which consists of beliefs, desires and intentions as mental attitudes that deliberate human action. Rao & Georgeff [15] adopted this model and transformed it into in a formal theory and an execution model for BDI agents that serves as a basis for the implementation of several BDI agent platforms.

Examples of agent platforms that implement the BDI architecture include Jason [2], JACK [7], Jadex [14], and the 3APL Platform[3]. In particular, these four platforms are based on the Java language. However, even though the underlying language is a general purpose programming language, agents are implemented in these platforms in a new programming language – AgentSpeak(L) [16], JACK Agent Language, a Domain-specific Language (DSL) writen in XML, and 3APL [5], respectively. Source code written in these languages is either precompiled or processed at runtime by the agent platform. As a consequence, the adoption of this approach prevents developers from using advanced features of the Java language, such as reflection and annotations, and it makes it complicated to integrate the implementation of a multi-agent system with existing technologies. Both issues are essential in the context of the development of large scale enterprise applications. Due to these limitations of existing platforms, we have implemented a new BDI agent platform, namely BDI4JADE. Our implementation is a layer on top of an existing agent platform, JADE [1], which provides a robust infrastructure to implement agents, but not follow the BDI architecture. We built a BDI reasoning mechanism for JADE agents implemented directly in the Java language, thus addressing the aforementioned problems.

The remainder of this paper is organized as follows. We first provide an overview of the BDI4JADE in Section 2, and then detail its individual components in Section 3. Section 4 discusses relevant aspects of our BDI implementation on top of JADE, followed by Section 5, which describes related work. Finally, Section 6 presents final remarks.

2 BDI4JADE: an Overview

As stated in the introduction, our motivation for implementing a new BDI agent platform is that the languages provided by existing platforms, even though based on general purpose programming languages, limit integration with up-to-date available technologies, and also the use of advanced features of the underlying programming language. We faced problems of this nature while implementing different multi-agent systems [11–13], and also in our current research [10], which involves dynamic adaptations of BDI agent architectures. These problems are detailed in Section 4.

An agent framework that fulfils the requirement of not relying on a DSL is JADE [1]. JADE is not based on the BDI model, which is our target architecture, but implements a task-oriented model, in which agents have a set of behaviors. No cognitive abilities, such as a reasoning cycle, are provided for agents. However, JADE is a robust and mature infrastructure, and provides many features that are needed for implementing multi-agent systems, which include the yellow pages service and message exchange. In addition, it provides a behavior scheduler that can be used to control the execution of plans of BDI agents. So, instead of developing an agent platform from scratch, we implemented the BDI architecture as a layer on top of JADE. Agents implemented with BDI4JADE use only the constructions provided by the Java language, which makes it easy to integrate with existing applications and reusable software assets (frameworks, components, libraries). Next, we briefly introduce the main BDI4JADE components.

[3] http://www.cs.uu.nl/3apl/

BDI agent. A BDI agent represents an agent that follows the BDI architecture. It aggregates a reasoning cycle, responsible for driving agent behavior, strategies, and capabilities.

Capability. A BDI agent does not directly include a belief base and a plan library, but these are part of a capability. A capability [4] is a self-contained part of an agent, consisting of (i) a set of plans, (ii) a fragment of the knowledge base that is manipulated by these plans and (iii) a specification of the interface to the capability. Capabilities have been introduced into some multi-agent systems as a software engineering mechanism to support modularity and reusability, while still allowing meta-level reasoning.

Strategies. A BDI agent is associated with different strategies, which are points for customizing the reasoning cycle, and can have their default behavior modified by developers. They are related to the revision of beliefs, the generation and deliberation of goals and selection of plans.

Goal. Goals represent the motivational state of the system. It is an entity that represents a desire that the agent wants to achieve.

Intention. An intention captures the deliberative component of the system. An intention is a goal that the agent is committed to achieve, i.e. when an agent has an intention, it will select plans to try to achieve this intention, until the associated goal is achieved, no longer desired or considered unachievable.

Belief Base and Belief. Beliefs represent environment characteristics, which are updated accordingly after the perception of changes on it. Beliefs can be seen as the informative component of the system. A belief base is a set of beliefs, each of which has a name and a value.

Plan Library and Plan. BDI4JADE provides an infrastructure to implement reactive planning systems, in which plans are not generated but selected from an existing plan library. Plans contain a set of actions and are executed with the aim of achieving a specific goal.

Events. BDI4JADE provides means for creating observers (listeners) of beliefs and goals, in order to notify them when these concepts are updated, so they can update their state accordingly. Any component that registers itself as an observer is notified when beliefs are created, update or removed, and when goals changed their status.

These components are used in the reasoning cycle of our BDI agents, which is based on the BDI-interpreter algorithm presented in [15]. This cycle is implemented in six major steps:

1. *Revising beliefs.* This first step of the cycle consists of revising agent beliefs. In the default implementation, nothing is done at this step, but developers can specify a customized strategy for specific agents.
2. *Removing finished goals.* Before the cycle is executed, goals might have "finished," i.e. they may be achieved, no longer desired or considered unachievable. These are removed from the set of goals of the agent, and observers of these goals are notified about the event.
3. *Generating options.* In this step, the goals available to the agent are determined (its desires). It can generate new desired goals, determine that existing goals are no longer desired, or keep existing goals that are still desired.

4. *Removing dropped goals.* When a goal, or set of goals, is determined as no longer desired in the previous step, it is removed from the set of goals of the agent, and observers are notified about the occurrence of this event.
5. *Deliberating goals.* In this step, the current agent goals are partitioned into two subsets: (i) goals to be tried to be achieved (intentions); and (ii) goals to *not* be tried to be achieved. The last will remain as an agent desire, but the agent is not committed to achieve it at the moment.
6. *Updating goals status.* Based on the partition performed in previous step, the status of the goals are updated. Selected goals are updated to the status of trying to achieve, and unselected goals are updated to the status of waiting. When a goal has the status trying to achieve, the agent will select plans for achieving that goal.

3 Detailing BDI4JADE Components

The previous section provided an overview of the main components of our implementation of the BDI architecture, which was slightly modified, for instance, by the addition of capabilities. It also described the implemented reasoning cycle in a high-level way. In this section, we provide further details of our JADE extension, BDI4JADE. We first present the core of our implementation, which consists of agents, intentions, capabilities and the reasoning cycle, and its whole structure. Then, we describe individual BDI4JADE components – how they were implemented and how to extend them.

Most os the concepts presented in previous section, and their relationships, are depicted in Figure 1, which shows the class diagram of BDI4JADE. Due to space restrictions, it contains only the main components of our implementation, and it presents only methods from interfaces, and not from classes.

3.1 BDI4JADE Core

A BDI agent in our platform must extend the `BDIAgent` class, which in turn is an extension of the `Agent` class from JADE. Therefore, as JADE agents, a BDI agent has its own thread of control, managed by JADE. A `BDIAgent` (from now on, we refer it to as agent) is composed of a set of intentions (`Intention` class) and a set of capabilities (`Capability` class).

When a goal is added to an agent, a new intention is created and attached to it. Intentions have a status associated with them, which are: (i) *Achieved* – the goal associated with that intention was achieved; (ii) *No longer desired* – the goal associated with that intention is no longer desired; (iii) *Plan failed* – the agent is trying to achieve the goal associated with that intention, but the last executed plan has failed; (iv) *Trying to achieve* – the agent is trying to achieve the goal associated with that intention, but it is executing a plan for achieving it; (v) *Unachievable* – all available plans were executed to try to achieve the goal associated with that intention, but none of them succeeded; and (vi) *Waiting* – the agent has the goal, but it is not trying to achieve it.

In the BDI architecture, an intention is a goal that an agent is committed to achieve. Our implementation does not make this distinction explicitly, but implicitly. Table 1 shows how concepts of the BDI architecture are related to the status of BDI4JADE

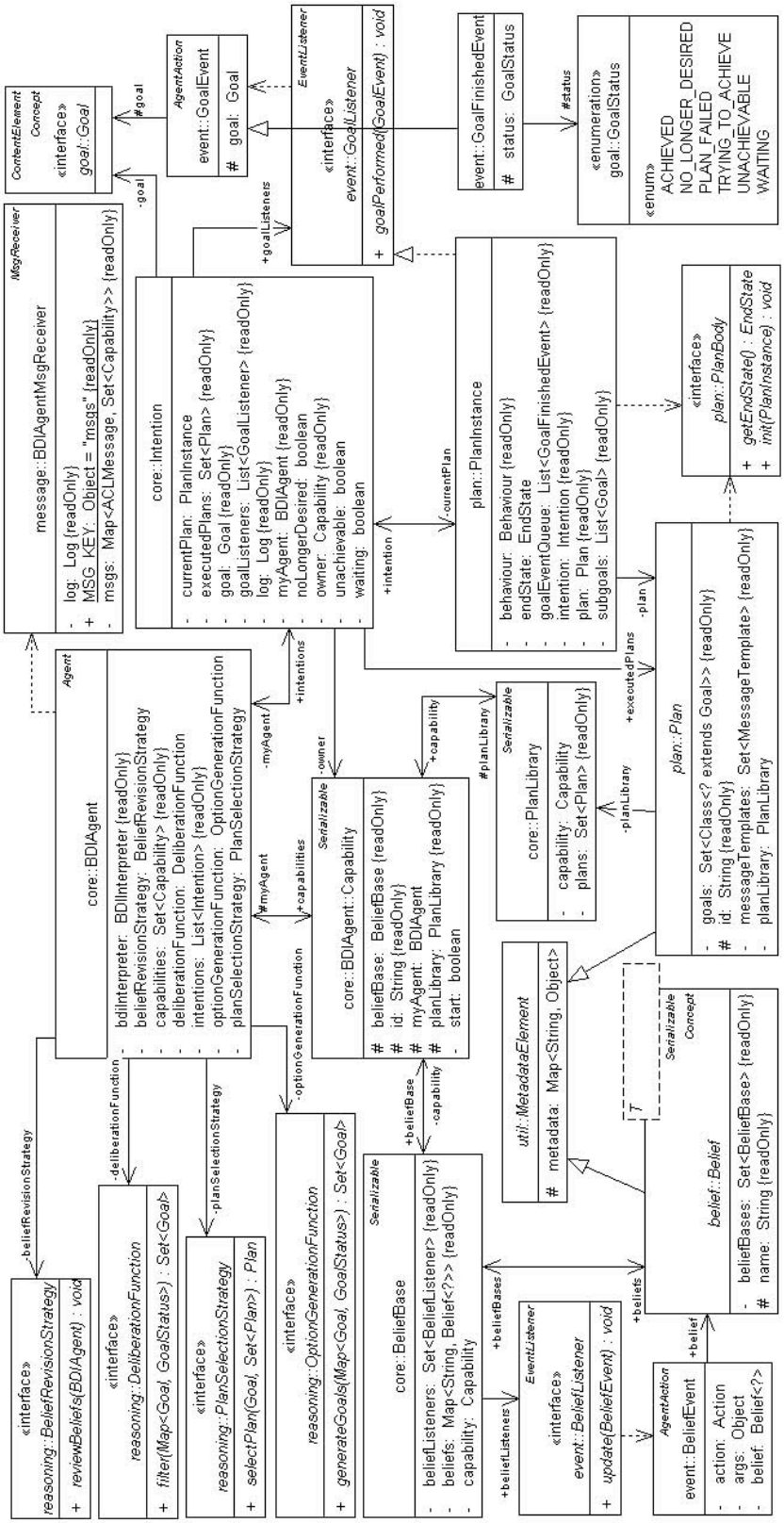


Fig. 1: Class diagram – BDI4JADE main classes and interfaces.

Status of the BDI4JADE Intention	BDI Architecture Concept
Waiting	Goal
Plan failed	Intention
Trying to achieve	Intention
Achieved	- (was an Intention)
No longer desired	- (was an Intention)
Unachievable	- (was an Intention)

Table 1: Intention Status x BDI Architecture Concept.

intentions. This approach was chosen to facilitate the implementation of the reasoning cycle. The last three status shown in Table 1 represent intentions/goals in a final state, and intentions with such status are removed from the agent in the next reasoning cycle.

As introduced before, beliefs and plans are not part of an agent (directly), as proposed in the BDI architecture, but part of capabilities. This concept is also implemented by JACK and Jadex agent platforms. As opposed to these platforms, beliefs and plans in our platform are not part of capabilities *and* agents, but only capabilities. However, a belief, or a plan, can be part of an agent if all capabilities contain that belief, or that plan. As we deal with Java objects, this can be easily done, because all capabilities will have a pointer for the same object. Shared belief bases are also possible.

A capability of our JADE extension is essentially composed of a belief base and plan library. The first is a collection of beliefs (see Section 3.3), and the latter a collection of plans (see Section 3.4). BDI4JADE does not provide means for explicitly defining capability interfaces, but they are exposed by documenting the capability. As a capability is associated with a set of plans, and these in turn are associated with the goals they can achieve, this set of goals indicates the goals that the capability can achieve. In addition, plans of a capability might require that other subgoals must be achieved when they are executed, so this set of goals indicates the goals that external components should achieve in order for the capability to be able to execute properly. These two set of goals can be seen as the provided and required interfaces of the capability, and should be part of the capability documentation.

All these components – capability, belief base and plan library – can be implemented either by *extension* or by *instantiation*. A developer can extend these components in the code and override the empty implementations of the `setup()` method for capabilities and the `init()` method for belief bases and plan libraries to initialize these components. The other option is to instantiate these components and add beliefs and plans through method invocation.

As opposed to typical BDI platforms, ours does not have an explicit declaration of goals in agents and capabilities. This binding occurs only at runtime. This provides more flexibility, because plans can be added (learned) to plan libraries at runtime and goals (which can be unknown at development time) can be added (desired) and achieved at runtime. This does not prevent the addition of goals at the agent initialization.

Reasoning Cycle. An essential part of a BDI agent platform is the reasoning cycle that it provides as part of agents. We previously presented how we implemented it in a high-level way. Next, we provide additional details. Listing 1.1 shows the source code of the reasoning cycle implemented in our platform.

Listing 1.1: BDI4JADE Reasoning Cycle.

```
public void action() {
    beliefRevisionStrategy.reviewBeliefs(BDIAgent.this);

    synchronized (intentions) {
      Map<Goal,GoalStatus> goalStatus =new HashMap<Goal,GoalStatus>();
      Iterator<Intention> it = intentions.iterator();
      while (it.hasNext()) {
          Intention intention = it.next();
          GoalStatus status = intention.getStatus();
          switch (status) {
          case ACHIEVED:
          case NO_LONGER_DESIRED:
          case UNACHIEVABLE:
              intention.fireGoalFinishedEvent();
              it.remove();
              break;
          default:
              goalStatus.put(intention.getGoal(), status);
              break;
          }
      }

      Set<Goal> generatedGoals = optionGenerationFunction
              .generateGoals(goalStatus);
      Set<Goal> newGoals = new HashSet<Goal>(generatedGoals);
      newGoals.removeAll(goalStatus.keySet());
      for (Goal goal : newGoals) {
          addGoal(goal);
      }
      Set<Goal> removedGoals = new HashSet<Goal>(goalStatus.keySet());
      removedGoals.removeAll(generatedGoals);
      for (Goal goal : removedGoals) {
          it = intentions.iterator();
          while (it.hasNext()) {
              Intention intention = it.next();
              if (intention.getGoal().equals(goal)) {
                  intention.noLongerDesire();
                  intention.fireGoalFinishedEvent();
                  it.remove();
              }
          }
      }

      goalStatus = new HashMap<Goal, GoalStatus>();
      for (Intention intention : intentions) {
          goalStatus.put(intention.getGoal(), intention.getStatus());
      }
      Set<Goal> selectedGoals=deliberationFunction.filter(goalStatus);
      for (Intention intention : intentions) {
          if (selectedGoals.contains(intention.getGoal())) {
              intention.tryToAchive();
          } else {
              intention.doWait();
          }
      }

      if (intentions.isEmpty()) {
          this.block();
      }
  }
}
```

The first step (line 2) invokes the belief revision function. It is performed by invoking the method `void reviewBeliefs(BDIAgent)` of an implementation of the `BeliefRevisionStrategy` interface. Next (lines 6-21), all finished intentions, i.e. intentions whose status is *achieved*, *no longer desired* or *unachievable*, are removed from the set of intentions of the agent, and a map `goalStatus` is created to store the status of each current goal of the agent.

The method `Set<Goal> generateGoals(Map<Goal, GoalStatus>)` of an instance of the `OptionGenerationFunction` interface is then invoked (lines 23-24) to create new goals or to drop existing ones. Based on the set of goals received as output, two actions are performed: (i) **new** goals are added to the agent, and consequently associated intentions are created (lines 25-29); and (ii) **removed** goals are set as no longer desired and removed from the agent (lines 30-42). Existing but not removed goals remain unchanged. The `goalStatus` is then updated (lines 44-47).

Next, it is time for the deliberation process, in which the agent selects the goals it will be committed to achieve. This is performed by invoking the method `Set<Goal> filter(Map<Goal, GoalStatus>)` of an instance of the deliberation function interface (line 48). It selects a set of goals that must be tried to achieve (intentions) from the set of goals. Selected goals and associated intentions will be set to trying to achieve, and unselected goals and associated intentions will be set to a waiting state. The invocation of the methods in lines 51 and 53 correctly adjusts the new state of the intention.

This reasoning cycle is implemented as part of a `CyclicBehaviour` of JADE, therefore it is performed continuously. In addition, it is added to all instances of `BDIAgent`. The `if` condition in line 57 tests if the agent has no current intentions, and, if so, it blocks the behavior. This avoids this behavior to be continuously executed while there are no intentions and goals. In case a new intention is added to the agent, the reasoning cycle is resumed.

Plan Selection. When the intention status is set to *trying to achieve* or *plan failed*, the private method `void dispatchPlan()` of the `Intention` class is invoked in order to select and execute a plan to try to achieve the goal associated with the intention.

This method first retrieves all plans that can achieve the goal, and then removes from this set of plans all plans that were already executed. The set of all plans that can achieve the goal is generated each time the `dispatchPlan()` method is executed because while a previous plan was being executed, new plans can be added to any capability of the agent. If there is no plan that can achieve the goal, the intention is set to *unachievable*. Otherwise, a plan will be selected by invoking the method `Plan selectPlan(Goal goal, Set<Plan>)` of the plan selection strategy of the agent. After the plan selection, it will be instantiated and started.

Extension points. While describing the implemented reasoning cycle, we mentioned four strategies: `BeliefRevisionStrategy`, `OptionGenerationFunction`, `DeliberationFunction` and `PlanSelectionStrategy`, but we did not detail it. These are Java interfaces, and are extension points of our platform. Developers can customize a `BDIAgent` by setting the implementation to be used during the reasoning cycle of a specific agent. BDI4JADE provides a default implementation for each of these strategies:

- `DefaultBeliefRevisionStrategy`: the `void reviewBeliefs()` method of the `BeliefBase` class of all capabilities is invoked;
- `DefaultOptionGenerationFunction`: it returns the current set of goals, i.e. it does not drop any of them and does not create any new goal;
- `DefaultDeliberationFunction`: it returns the whole set of goals, i.e. all goals will be set to a trying to achieve status; and
- `DefaultPlanSelectionStrategy`: it returns `null` if the set of plans is empty, and the first plan retrieved from the set, otherwise.

This way of extending and customizing agents is an implementation of the strategy design pattern [6].

3.2 Goals

A goal in BDI4JADE can be any Java object, with the condition that it must implement the `Goal` interface. Therefore, a class implementing this interface can be created and attributes can be added to it as inputs and outputs of the goal. We also provide a set of predefined goals to be used in applications:

`BeliefGoal`. The input of this goal is the name of a belief. This goal is achieved when a belief with the provided name is part of the agent's beliefs.

`BeliefSetValueGoal<T>`. The input of this goal is the name of a belief and a value. This goal is achieved when the belief with the provided name is part of the agent's beliefs and has the provided value.

***`CompositeGoal`*.** This class represents an abstract goal that is a composition of other goals (subgoals). It has two subclasses, which indicate if the goals must be achieved in a parallel or sequential way.

`ParallelGoal`. This class represents a goal that aims at achieving all goals that compose it in a parallel way. It is a subclass of the *`CompositeGoal`*.

`SequentialGoal`. This class represents a goal that aims at achieving all goals that compose it in a sequential way. It is a subclass of the *`CompositeGoal`*.

`MessageGoal`. This goal is created when a message is received by the agent. It stores the message received. How this goal will be achieved is described in Section 3.5.

In order to add a new goal to an agent, the only thing that must be done is to invoke the method `void addGoal(Goal goal)` of an instance of the `BDIAgent`.

3.3 Beliefs

The `BeliefBase` class offers methods to manipulate beliefs, such as add, remove and update beliefs. Beliefs can store any kind of information and are associated with a name. If the value of a belief is retrieved, it must be cast to its specific type, as it is the case in Jadex. We have used Java generics to capture incorrect castings at compile time, so beliefs in the BDI4JADE are instances of subclasses of `Belief<T>`.

A belief has two main properties: a name and a value. The belief name must be unique in the scope of a belief base. There are two main characteristics about beliefs

to be described: (i) its class is generic, i.e. it receives a type when it is instantiated. Therefore, when a belief is declared in a plan or somewhere else, no type casting must be performed to retrieve its value; and (ii) it extends the class `MetadataElement`, which is a class of metadata – a map from string to objects. Metadata can be used for specific purposes of applications, for instance, time can be added to beliefs, so they can be forgotten after a certain amount of time.

The `Belief<T>` is an abstract class, because it does not specify how the value is stored, but defines methods that must be implemented by subclasses to retrieve and set the value associated with the belief. Currently, there is only one form of storing beliefs, which is implemented by the `TransientBelief<T>` class. This class stores the value of the type `T` in memory, and there is no persistence mechanism.

In addition, there is a particular type of belief to store sets – the `BeliefSet<T>`, which extends `Belief<Set<T>>`. As the `Belief<T>` class, it is abstract and can have different subclasses to store belief values. The `BeliefSet<T>` defines methods to retrieve, store and iterate belief values, and has an implementation that stores values in memory – the `TransientBeliefSet<T>` class.

3.4 Plans

The representation of plans in the BDI4JADE is not associated with one but with a set of classes. One of the reasons is that our goal is to reuse JADE as much as possible in order to: (i) facilitate the learning process of developers already familiar with JADE; (ii) take advantage of the family of JADE behaviors; and (iii) exploit reuse benefits – which is higher quality due to the use of a piece of software used a lot of times, and reduced development costs. Plans to be executed (plan bodies) in our platform are instances of the JADE behavior, and their execution is controlled by the JADE scheduler.

Our platform has three main classes associated with plans:

`Plan`. A `Plan` does not state a set of actions to be executed in order to achieve a goal, but has some information about it, which is: (i) the plan id; (ii) the plan library that is belongs to; (iii) the goals that it is able to achieve; and (iv) the message templates it can process. In addition, it defines some important methods to be implemented by subclasses:

- `public abstract Behaviour createPlanBody()` – this method returns an instance of a JADE behavior, which corresponds to the body to be executed to achieve the goal. This behavior instance must also implement the `PlanBody` interface (verification made at runtime). This method must be implemented, because it is an abstract method, and therefore the `Plan` class is also abstract.
- `protected void initGoals()` – this method must be overridden by subclasses to initiate the set of type of goals that this plan can achieve.
- `protected void initMessageTemplates()` – this method must be overridden by subclasses to initiate the set of message templates (from JADE) that this plan can process.
- `protected boolean matchesContext(Goal goal)` – this method verifies a context to determine if the plan can achieve the goal according to the

Listing 1.2: Verifying if a plan can achieve a goal.

```
1 public boolean canAchieve(Goal g) {
2   if (g instanceof MessageGoal) {
3       return canProcess(((MessageGoal) g).getMessage());
4   } else {
5       return goals.contains(g.getClass()) ? matchesContext(g) : false;
6   }
7 }
```

current situation of the environment. The default implementation returns always `true`.

Listing 1.2 presents the method that is executed to verify if a plan can achieve a given goal. If the goal is an instance of `MessageGoal`, i.e. it is the goal of processing a received message, it verifies if any of the message templates of the plan matches the received message. Otherwise, it checks if the goal has a type that can be achieved by the plan, and if so, it verifies if the context required by the plan matches the current context.

Our platform provides a concrete implementation of `Plan`, the `SimplePlan`. This class has a `Class<? extends Behaviour>` associated with it, which must also implement the `PlanBody` interface (test made at runtime). When the `createPlanBody()` is invoked, an instance of the class associated with the `SimplePlan` will be created. This class in turn has two subclasses used to achieve generically sequential and paralell goals (see Section 3.2). In addition, we also provide plans for achieving *`CompositeGoal`* goals.

`PlanInstance`. This class, as the name indicates, is an instance of a plan, which is created to achieve a particular goal, according to a specification of a plan. It has the following attributes: (i) `Behaviour behaviour` – the behavior being executed to achieve the goal associated with the intention; (ii) `Intention intention` – the intention whose goal is trying to be achieved; (iii) `Plan plan` – the plan that this plan instance is associated with; (iv) `EndState endState` – the end state of the plan instance (`FAILED` or `SUCCESSFUL`), or `null` if it is currently being executed; (v) `List<Goal> subgoals` – the subgoals dispatched by this plan. In case of the goal of the intention associated with this plan of this plan instance is dropped, all subgoals are also dropped; and (iv) `List<GoalFinishedEvent> goalEventQueue` – when this plan instance dispatches a goal, it can be notified when the dispatched goal finished.

`PlanBody`. As we established that JADE behaviors would be used to execute plans and that we aimed at reusing the JADE behaviors hierarchy, we could not extend the `Behaviour` class of JADE, due to Java limitations regarding multiple inheritance. So, our decision was to define an interface to be implemented by plan bodies, besides extending a JADE behavior. Two methods should be implemented by plan bodies: (i) `EndState getEndState()` – it returns the end state of the plan body. If it has not finished yet, it should return `null`. This shows that the platform detects that a goal was achieved when the selected plan finished with a

Listing 1.3: Dispatching and waiting for subgoals.

```
switch (state) {
    case 0:
        planInstance.dispatchSubgoalAndListen(subgoal);
        state++;
        break;
    case 1:
        GoalFinishedEvent goalEvent = planInstance.getGoalEvent();
        if (goalEvent != null) {
            if (GoalStatus.ACHIEVED.equals(goalEvent.getStatus())) {
                (...)
            } else {
                (...)
            }
        }
        break;
}
```

`SUCCESSFUL` state; and (ii) `void init(PlanInstance planInstance)` – this method is invoked when the plan body is instantiated. This is used to initialize it, for instance retrieving parameters of the goal to be achieved.

In order to dispatch a goal and wait for its end, we adopted a mechanism similar to the one of receiving messages in JADE. The developer, after dispatching the goal, should retrieve a goal event and test if it is `null` (no goal event received yet) or not (an event was received). Listing 1.3 shows an example of how it can be done. The method `dispatchSubgoalAndListen()` blocks the behavior in case there is no goal event when it was invoked (a timeout can be provided for the method). The behavior will become active again when a goal event is received.

3.5 Messages

Messages are received and sent in the BDI4JADE basically as it is done in JADE. Conversations are made by sending messages, and using the `receive(MessageTemplate)` method to receive a reply. Additionally, BDI4JADE provides an additional mechanism for processing messages that are received. Every `BDIAgent` has a behavior `BDIAgentMsgReceiver` associated with it, which extends the `MsgReceiver` class from JADE. The latter is a behavior that handles a message when the match expression of the behavior returns a `true` value related to the analysis of the message received. The match expression of the `BDIAgentMsgReceiver` class checks if any of the capabilities of the agent have at least one plan that can process the received message. If so, the expression returns `true`. After that, the behavior adds a `MessageGoal` to the agent, with the received message associated with it. Eventually, the reasoning cycle will select a plan that can process the message to perform it.

3.6 Events

Our platform implements the observer design pattern [6] in some points to enable the observation of events that occur in an agent. Currently, there are two kinds of events:

belief and goal events. Belief listeners can be associated with a belief base, and whenever a belief is added, removed or changed, the listener will be notified. It is important to highlight that a belief can have its value changed simple by invoking the `void setValue(T)` method of the `Belief` class, and in this case, the listeners will not be notified. Goal listeners in turn are associated with an intention. It is used to observe changes in the status of the intention. An example of its use was presented in Section 3.4, in order to detect when a subgoal is achieved (or finished with another status).

4 Discussion

In this section, we discuss relevant aspects related to our JADE extension. These aspects are mainly associated with current limitations of BDI4JADE and development experiences with it.

Not implemented yet. There are improvements that we aim at developing for BDI4JADE, but they have not been implemented yet and will be future extensions of the platform. They are: (i) **Persistent beliefs** – currently, our platform only provides transient beliefs. We intend to incorporate the Hibernate[4] framework to our platform to facilitate the creation of beliefs that are persisted in databases; (ii) **Control of intention/goal owners** – we have created the `InternalGoal` interface to denote a goal that is internal to a capability. Plans that are being executed are associated with a plan library, which is in turn associated with a capability. Therefore, if the plan dispatches a goal, this goal is under the scope of this capability. This information is not being currently stored. Our goal is to limit the scope of the searching space of plans to the capability that dispatched the goal, when the goal is an internal goal. This helps creating encapsulated capabilities and improving reuse; and (iii) **Indexes for plan libraries** – every time that a plan must be selected for achieving a goal, the plan library is asked to provide the list of plans that can achieve that goal. We aim at creating indexes for speeding up this process.

As we have not implemented (ii) yet, we also did not consider nested capabilities. The difference between adding two capabilities to an agent, or adding one capability to another, and the last to an agent is that when an internal goal is dispatched by the parent capability, it can be achieved by the plans that are part of it, or part of sub-capabilities. Without goal owners control, nested capabilities will present the same behavior of capabilities added to the same agent.

Debugging BDI4JADE agents. Most of existing BDI platforms provide tools to debug the implemented multi-agent systems and to inspect current state of agents. We have not developed any tools for supporting the development of agents. Nevertheless, as BDI4JADE agents are fully developed with Java, its debugger already provides information for debugging agents. The agent current state can be inspected with existing tools, typically attached to Java IDEs. In addition, tools provided by the JADE platform can also be adopted. They allow not only monitoring messages exchange, but also active plans, as they are implemented as JADE behaviors.

Testing BDI4JADE. In order to test our implementation, we have developed several example applications that test different parts of BDI4JADE. The tests included messages

[4] http://www.hibernate.org/

exchange (ping application), different aspects of the reasoning cycle (trying different plans, dropping goals, and so on), and subgoals and composite goals. In addition, we have implemented the typical BDI application "Blocks world," which consists on moving blocks in an initial configuration to a target configuration. Moreover, BDI4JADE is been used in the context of our current research work [10]. It involves the development of agent-based software to assist users in routine tasks that users can customize based on a high-level language. This requires dynamic adaptation of agents architectures, and for that we adopt enterprise frameworks such as the Spring framework,[5] and therefore having agents implemented in "pure" Java is essential.

We have not run any stress test in BDI4JADE in order to test its scalability and performance, and compare these aspects with other existing platforms. We did not prioritize this kind of test because our main motivation with this work is to improve the development of multi-agent systems from a developer perspective, but, as thread control in BDI4JADE is performed by JADE, and this is the main issue related to the performance of multi-agent systems, we believe that systems implemented with BDI4JADE tend to have a performance similar to the ones implemented with JADE. However, further studies must be performed in this direction.

Relevance of the Integration with Existing Technologies. Our major concern while developing BDI4JADE was to provide an infrastructure that can be easily integrated with existing technologies. We identified this need during the development of multi-agent applications [11–13]. They involve the development of web-based systems that require the integration with: web application frameworks (help on managing web requests and creating dynamic web pages); Spring framework[5] (provides transaction management and dependency injection); software aspects [8] (a modularization technique for crosscutting concerns); and persistence frameworks (deal with database access and persisting entities). These technologies are commonly used for developing large scale enterprise applications and they are essential to increase software quality and reduce development costs, as these technologies have already been widely tested and provide ready-to-use infrastructures. In addition, software evolution is a reality, and agent technology must be able to be smoothly integrated to existing systems.

The main issue related to the integration with software frameworks is that, as opposed to libraries that are invoked by specific applications, they adopt the Hollywood principle: "Don't call us, we call you." This means that application-specific components are instantiated and invoked by the framework, and this usually requires components to implement interfaces. With existing agent platforms, this is not possible because components are not implemented with Java classes, and also the platform components are instantiated and manipulated by the agent platform, and its it usually hard to identify pointers to the platform component instances to make advances manipulations with them. In the case of software aspects, one of the most widely used implementations of it, AspectJ[6], requires exposing Java interfaces for the specification of join points, and again, without the implementation of Java classes, this is not possible.

[5] http://www.springsource.org/
[6] http://www.eclipse.org/aspectj/

5 Related Work

Different BDI agent platforms have already been proposed. Nevertheless almost all of them require the implementation of agents in a new programming language or a DSL, even though the implementation of the underlying agent platform is expressed in a general purpose programming language. This is the case of Jason [2], whose agents are implemented in an extension of the AgentSpeak language [16]; JACK [7], that has an specific language, the JACK Agent Language, which is precompiled for Java; and 3APL [5], an agent programming language with a platform implemented in Java.

The framework that has more similarities to BDI4JADE is Jadex [14], which uses JADE as a middleware. Our experience with the development of different applications using Jadex [11, 12] was also a motivation for developing our implementation. The main benefit of Jadex is that it provides the concepts of the BDI architecture for developers. In addition, it provides the capability concept, which allows for packaging a subset of beliefs, plans, and goals into an agent module and to reuse this module wherever needed. As a consequence, one can easily (un)plug capabilities to agents and reuse them.

However, Jadex defines agents through XML files, and this leads to drawbacks during the implementation. Programming an agent using XML prevents the use of features of the underlying programming language, and the integration with existing technologies becomes a challenge. Another disadvantage is that finding errors in XML files is a tedious task. Additionally errors are not captured during compilation time, because typos may occur even though the document is valid according to its DTD. For instance, if a goal is referenced within the XML file with a wrong letter, an error will occur only during execution time, and the message only says that the XML file has errors. As a consequence, the developer has to find the error manually. Moreover, even though plans are Java classes, beliefs and parameters are retrieved by methods that return an object of the class `Object`, so there must be type casting while invoking these methods. This leads again to capturing errors only at runtime.

6 Final Remarks

In this paper we presented BDI4JADE, an agent platform that implements the BDI architecture. As opposed to different BDI platforms that have been proposed, it does not introduce a new programming language nor rely on a DSL written in terms of XML files. Because agents are implemented with the constructions of the underlying programming language, Java, we bring two main benefits: (i) features of the Java language, such as annotations and reflection, can be exploited for the development of complex applications; and (ii) it facilitates the integration of existing technologies, e.g. frameworks and libraries, what is essential for the development of large scale enterprise applications, which involve multiple concerns such as persistence and transaction management. This also allows a smooth adoption of the agent technology. BDI4JADE is a BDI layer on top of JADE, and it leverages all the features provided by the framework and reuses it as much as possible. Other highlights of our JADE extension, besides providing BDI abstractions and reasoning cycle, include: (i) use of capabilities: agents aggregate a set of capabilities, which are a collection of beliefs and plans. This allows modularizing

particular behaviors of agents; (ii) Java generics for beliefs – beliefs can store any kind of information and are associated with a name. If the value of a belief is retrieved, it must be cast to its specific type. We have used Java generics to capture incorrect castings at compile time; and (iii) plan bodies as instances of JADE behaviors: in order to better exploit JADE features, in particular its behaviors hierarchy, plan bodies are subclasses of JADE behaviors. Our platform as well as examples of its use are available in [9]. BDI4JADE is being used in the context of our current research work [10].

References

1. Bellifemine, F.L., Caire, G., Greenwood, D.: Developing Multi-Agent Systems with JADE (Wiley Series in Agent Technology). John Wiley & Sons (2007)
2. Bordini, R.H., Wooldridge, M., Hübner, J.F.: Programming Multi-Agent Systems in AgentSpeak using Jason. John Wiley & Sons (2007)
3. Bratman, M.E.: Intention, Plans, and Practical Reason. Harvard University Press, Cambridge, MA (1987)
4. Busetta, P., Howden, N., Rönnquist, R., Hodgson, A.: Structuring BDI agents in functional clusters. In: ATAL '99. pp. 277–289 (2000)
5. Dastani, M., van Riemsdijk, M.B., Dignum, F., Meyer, J.J.C.: A programming language for cognitive agents goal directed 3apl. In: Programming Multi-Agent Systems, LNCS, vol. 3067, pp. 111–130. Springer Berlin / Heidelberg (2004)
6. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-oriented Software. Addison-Wesley (1995)
7. Howden, N., Rnnquista, R., Hodgson, A., Lucas, A.: Jack intelligent agents™: Summary of an agent infrastructure. In: The Fifth International Conference on Autonomous Agents. Montreal, Canada (2001)
8. Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-Oriented Programming. In: ECOOP 1997. vol. 1241, pp. 220–242. Springer-Verlag, Berlin, Heidelberg, and New York (June 1997)
9. Nunes, I.: A bdi extension for jade (2010), http://www.inf.puc-rio.br/˜ionunes/bdi4jade/
10. Nunes, I., Barbosa, S., Lucena, C.: Increasing users' trust on personal assistance software using a domain-neutral high-level user model. In: ISoLA 2010, LNCS, vol. 6415, pp. 473–487. Springer (2010)
11. Nunes, I., Cirilo, E., Lucena, C.: Developing a family of software agents with fine-grained variability: an exploratory study. In: SEAS 2009. pp. 71–82 (2009)
12. Nunes, I., Kulesza, U., Nunes, C., Cirilo, E., Lucena, C.: Extending web-based applications to incorporate autonomous behavior. In: Proc. of the 14th Brazilian Symposium on Multimedia and the Web (WebMedia'08). pp. 115–122 (2008)
13. Nunes, I., Nunes, C., Kulesza, U., Lucena, C.: Agent-oriented software engineering ix. chap. Developing and Evolving a Multi-agent System Product Line: An Exploratory Study, pp. 228–242. Springer-Verlag, Berlin, Heidelberg (2009)
14. Pokahr, A., Braubach, L.: Jadex user guide. Tech. Rep. 0.96, University of Hamburg, Hamburg, Alemanha (2007)
15. Rao, A.S., Georgeff, M.P.: BDI-agents: from theory to practice. In: Proceedings of the First Intl. Conference on Multiagent Systems. San Francisco (1995)
16. Rao, A.S.: Agentspeak(l): Bdi agents speak out in a logical computable language. In: MAAMAW '96. pp. 42–55. Springer-Verlag (1996)
17. Zambonelli, F., Jennings, N.R., Omicini, A., Wooldridge, M.: Agent-oriented software engineering for internet applications. In: Coordination of Internet Agents, pp. 326–346. Springer Verlag (2001)

Integrating Expectation Handling into Jason

Surangika Ranathunga, Stephen Cranefield, and Martin Purvis

Department of Information Science, University of Otago,
PO Box 56, Dunedin 9054, New Zealand
{surangika,scranefield,mpurvis}@infoscience.otago.ac.nz

Abstract. Although expectations play an important role in designing cognitive agents, agent expectations are not explicitly being handled in most common agent programming environments. There are techniques for monitoring fulfilment and violation of agent expectations, however they are not linked with common agent programming environments so that agents can be easily programmed to respond to these circumstances. This paper investigates how expectation monitoring tools can be tightly integrated with the Jason BDI agent interpreter by extending it with built-in actions to initiate and terminate monitoring of expectations. This enables Jason agents to monitor for the fulfilment and violation of their expectations without relying on a centralised monitoring mechanism. This way, it is possible for agents to have plans that respond to the identified fulfilments and violations of their expectations.

1 Introduction

Expectations represent the anticipatory mental component of an agent, thus they resemble an important part of cognitive agents. When an agent bases its practical reasoning on the assumption that one or more of its expectations will hold, it somehow has to ensure that it is aware of when these expectations are fulfilled and/or violated.

Although much research can be found on techniques for monitoring fulfilment and violation of various types of future expectation such as those based on norms, commitments, and contracts (see [8] for a brief survey of the existing monitoring techniques), we do not see much research on providing support for these monitoring techniques in common agent programming environments. However, to successfully implement normative multiagent systems using these agent programming environments, it is important that they support techniques to monitor for the fulfilment and violation of these various types of future expectations to help in the development of socially aware multiagent systems.

In this work, we present an approach for tightly integrating expectation monitoring with the Jason [4] Belief-Desire-Intension (BDI) agent interpreter, by extending it with built-in actions to initiate and terminate monitoring of expected constraints on the future and by defining specific belief types to represent detected fulfilments and violations of expectations. With the introduction of these built-in actions, any third party monitoring tool can be "plugged in" to the Jason

environment, and in this paper we demonstrate this with an expectation monitor developed in previous research [7]. Moreover, we present extended operational semantics for Jason, which incorporates expectation handling.

Our mechanism allows agents to choose to delegate to an expectation monitor service the monitoring of rules that specify conditional constraints on the future. These rules may be based on published norms, agreed contracts, commitments created through interaction with other agents, or personally inferred regularities of behaviour, and multiple instances of the monitoring service may be active on behalf of different agents at any time. In some future time, fulfilments and violations of an agent's expectations may occur and will be detected by the monitor. These result in belief addition events for the given agent and can be handled in a flexible way by creating plans that are triggered by those events.

The benefit of using a monitoring service available within Jason rather than using an external monitoring agent is that it is easier to apply this monitoring mechanism to different applications. The only requirements for agent system developers to understand and use our approach are understanding of the abstract idea of monitoring for fulfilments and violations of future-oriented expectations and the signature of two new internal Jason actions, and to be provided with the customized (Java) logic needed to connect a given monitoring technique with Jason. This is in contrary to monitoring mechanisms based on specialised monitoring agents, such as what was presented by Meneguzzi et al. [10]. In that work, a norm monitoring tool for a specific domain was implemented as an agent and agent-level communication was used between the monitor agent and its client (a Jason agent). This can be seen as an application pattern that can be reused for different domains, but this reuse requires understanding of the function of the monitor agent, the protocols used for communication, and the Jason plans used to handle communication with the monitor agent. Furthermore, while this approach is suitable for providing an official monitor for norms and contracts defined at the institutional level, it would introduce undesirable communication overhead if used as an architecture for agents wanting their own individual expectations monitored, for use in their own personal reasoning processes.

However, it should be noted that our approach does not rule out the use of a single designated monitoring agent to monitor expectations (e.g. norms) applying to a whole society. Such a monitor agent can also make use of the techniques discussed in this paper.

The rest of the paper is organized as follows. Section 2 gives an overview of agent expectations and their significance. Section 3 gives an overview of the Jason platform, and Section 4 contains an overview of the expectation monitor used in this work. Section 5 lays out the introduced extensions to Jason, and in Section 6 we demonstrate these by means of an example. Finally, Section 7 concludes the paper.

2 Expectations of Cognitive Agents

Expectations represent the anticipatory mental component of an agent. In a theoretical perspective, expectations are "hybrid mental configurations whose components entail not only beliefs but also converging goals that those beliefs will be realized" [6]. Despite several definitions on how expectations are really formed based on beliefs and goals, it is the common agreement that beliefs and goals are the two elements that form expectations of an agent.

According to Castelfranchi [5], the belief component of an expectation is a mental anticipation of a future state or event. In addition, the expectation includes a goal (which can be any motivational mental state, such as a wish, desire or an intention[1]) to know whether the anticipated state or event occurs. This means that the agent has to monitor what is happening, and it should compare this received information with its mental representation of the belief in order to satisfy the goal to know whether the world actually is as anticipated.

Along with perceptions, expectations play a very important role in generating emotions of agents such as hope, fear, frustration, disappointment, and relief [5]. For example, if the agent had been expecting something bad, and the received punishment is less than what was expected, it generates the emotion *relief*. On the other hand, if the agent achieved a lesser result than what was expected, it leads to the emotional state *disappointment*.

Expectations also have an important role to play in the social context of multi-agent systems. They play a fundamental role in defining social norms, conventions, and commitments, and on the other hand, can arise due to these normative components [6]. Expectations are also the root course in generating social trust, where trust can be defined as a complex form of expectation [9]. Therefore properly analysing and modeling individual agent expectations is beneficial in developing more reactive and emotional agents as well as normative multi-agent societies.

3 Jason

Jason [4] is a Java-based interpreter for an extended version of the AgentSpeak agent programming language [12], which is based on the BDI model. Jason is open source, and is commonly used in research related to developing agent systems. It is a well documented and well supported platform with an active user community.

A Jason agent program consists of plans that are executed in response to the events received. These events are generated by the addition or deletion of the beliefs and goals of an agent. A belief is the result of a percept the agent

[1] In our approach, we provide a mechanism for the agent programmer to directly create an intention to monitor an expectation that is delegated to an external expectation monitoring tool. The belief component of the expectation is a declarative representation of the expectation in the language used by the monitor, e.g. a conditional rule encoded in temporal logic.

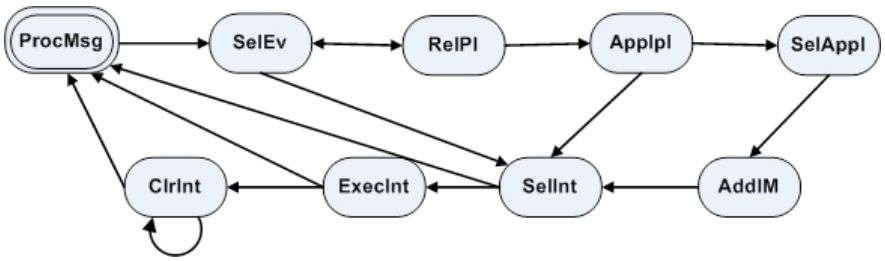


Fig. 1. Possible state transitions within one reasoning cycle [4]

retrieves from its environment or it could be based on a message received from another agent. A goal could be internally generated by the agent, or it could be a goal that was asked to be achieved by another agent. While executing a plan, an agent might generate new goals, act on its environment, create mental notes to itself or execute internal actions.

3.1 Jason Agent Reasoning Cycle

An agent is operated in a continuous cycle called the reasoning cycle, operating on a stream of belief update events produced as the agent perceives its environment. The Jason semantics define the following steps for interpreting AgentSpeak programs, as shown in Figure 1 [4]. First, the reasoning cycle checks for messages received by the agent and selects one of the messages to process further (ProcMsg). Then one event from the many pending events is selected to be processed further in that reasoning cycle (SelEv). Then is the step of selecting the set of relevant plans for the given event (RelPl). From the relevant plans, a subset of plans are again retrieved, which are currently applicable (ApplPl), meaning that their context conditions evaluate to true given the agent's current beliefs. Then, one plan from the set of applicable plans (the intended means) is selected for execution (SelAppl), and the new intended means is added to the set of intentions (AddIM). Then the reasoning cycle chooses one of the pending intentions (SelInt), executes one step of that intention (ExecInt), and clears the intended means that may have finished in the previous step (ClrInt).

Most of the aforementioned steps are directly customizable and since the Jason source is freely available, other functionalities of the Jason interpreter can also be changed as needed.

4 The Monitoring Tool

In this paper we selected an expectation monitor developed in previous research [7] to be integrated with the Jason platform. This expectation monitor aims at monitoring expectations that encode complex temporal constraints on the future. It uses a language based on hybrid temporal logic to facilitate this encoding. With the use of temporal logic, this expectation monitor is able to deal

with expectations with complex temporal aspects, as opposed to many other monitoring techniques that handle only propositions that must come true by a deadline[2].

The language that the expectation monitor accepts includes the following operators relating to conditional rules of expectation[3]:

- $\mathsf{Exp}(Condition, Expectation)$
- $\mathsf{Fulf}(Condition, Expectation)$
- $\mathsf{Viol}(Condition, Expectation)$

where *Condition* and *Expectation* are formulae in a form of linear propositional temporal logic. *Condition* expresses a condition on the past and present, and *Expectation* is a constraint that may express an expectation on the future or a check on the past (or both). Expectations come into existence when their condition evaluates to true in the current state. These expectations are then considered to be fulfilled or violated if they evaluate to true in a state.

However, this evaluation must be done without using any information from future states in the model. For example, when examining a model representing an audit trail, the expectation that a certain party will never access a certain file ($\mathsf{Exp}(\mathit{true}, \Box\neg \mathit{bob\_accessed\_file})$) should *not* be deemed to be violated before any state (s_9, say) in which that prohibited access occurs, even though $\Box\neg \mathit{bob\_accessed\_file}$ would evaluate to *false* in any state up to and including s_9. Thus the semantics for Fulf and Viol include a notion of truncating the model at the current state and using "strong" finite model semantics to evaluate the expectation [7]. In this way, the language provides declarative semantics for expectations and their fulfilment and violation can be applied to both offline monitoring such as examining audit trails, and online monitoring where states are added incrementally to the model.

If an active expectation is not fulfilled or violated in a given state, then it remains active in the following state, but in a "progressed" form. Formula progression involves partially evaluating the formula in terms of the current state and re-expressing it from the viewpoint of the next state [3], e.g. if p is true in the current state, an expectation that "p is true and q is true in the next state" will progress to the expectation that "q is true in the current state".

Although this expectation monitor supports the monitoring for the existence (i.e. activation) of an expectation, as well as its fulfilment or violation, in the integration of the monitor with Jason, we currently handle only the fulfilment and violation of an expectation. We intend to modify our extension to support monitoring for the existence of an expectation in future versions.

As an example, consider the football team play scenario "give and go" illustrated in Figure 2. This team play scenario involves two players where one player

[2] A comparison with other approaches used for monitoring expectations and related concepts such as norms and commitments is out of the scope of this paper, but can be found in reference 8.

[3] In previous work these operators were named ExistsExp, ExistsFulf and ExistsViol, but we use simplified names here.

(player 1 in the figure) passes the ball to her team mate (player 2). Player 2 then adopts an expectation that player 1 will run down to an advantageous field position (which was agreed upon according to the team tactic). The intention of player 2 is to pass the ball back to player 1, when she fulfils this expectation. On the other hand, if player 1 was unable to fulfil this expectation, player 2 has to initiate a new tactic.

As player 2 has to focus on advancing down the field while avoiding opposition players, the expectation monitor can be delegated to monitor the performance of player 1, to check whether she fulfils (or violates) the defined expectation of player 2. The fulfilment and violation of this expectation can be expressed using the following two formulae, where we assume that player 1 is supposed to advance towards the goal B in the field, *until* she reaches the penalty area in front of goal B.

$$\mathsf{Fulf}(s5, advanceToGoalB(player1)\,\mathsf{U}\,penaltyB(player1))$$
$$\mathsf{Viol}(s5, advanceToGoalB(player1)\,\mathsf{U}\,penaltyB(player1))$$

The first argument in the formulae refers to the condition that triggers the expectation, as explained earlier. As Player 2 wants to begin monitoring the expectation as soon as the ball is received, this condition is given as a *nominal* (a proposition that is true in only one state) that 'names' the current state (state 5, in this example). This ensures that the rule is fired precisely once, immediately[4].

The first formula above evaluates to true in any state in which the rule is fulfilled (i.e. player 1 reaches the penalty B area), and the second formula will be true in any state in which the rule is violated (e.g. player 1 moves in the opposite direction from goal B or stops moving before reaching the penalty area). In these cases, the monitor sends a belief addition event back to player 2 to inform her that this rule was fulfilled or violated.

5 Expectation Handling in Jason

5.1 New Internal Actions to Start and Stop Expectation Monitoring

An important feature of our monitoring mechanism is the ability to use any third-party monitoring tool in conjunction with Jason plans. Therefore the interface between Jason and an expectation monitor was designed to be more abstract than the logic described in the previous section. This helps to switch to different expectation monitor techniques without changing the actual agent logic that initializes or terminates expectation monitoring. Our intention is to provide a generic interface that would suit a range of monitoring tools.

[4] In practice, we do not require the agent to know the nominal for the current state—we allow a special keyword "#once" to be used as the condition of a rule, and this is replaced by a nominal for the current state when the rule is sent to the monitor (see Section 6).

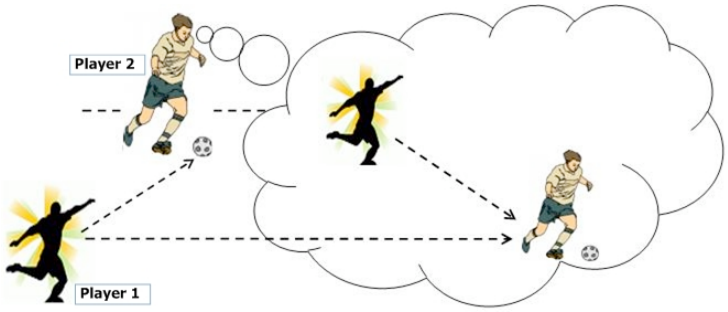


Fig. 2. The "give and go" tactic in football

Internal actions in the Jason platform help programmers to extend agent capabilities by defining them in the Java programming language. Internal actions are appropriate to use when the corresponding logic cannot be expressed in AgentSpeak language constructs (e.g. integrating Jason with an external program) or involve computations of a procedural nature that are more conveniently expressed in Java.

The new internal actions needed to extend the Jason platform are directly added to the standard internal actions library, enabling any agent program to refer to those. Thus an agent programmer does not have to know how to design these internal actions. However, if the custom logic related to an expectation monitor is included in these internal actions, the agent programmer has to change the standard Jason code each time when integrating a new expectation monitor type. Therefore we have made it possible to store this custom code in a Java class which is stored inside the same Java package as the related agent program. The internal actions expect the existence of this customized class to handle the specific logic related to a given expectation monitor. The internal actions decode the parameter values sent by an AgentSpeak program, and send these values to this customized class, to be processed according to the selected expectation monitor.

Initiating Expectation Monitoring: The internal action corresponding to the initialization of expectation monitoring is *start_monitoring*. Currently each call to start_monitoring creates a new instance of the monitor. This is due to a current limitation of the monitor implementation that it only handles one rule at a time. This introduces a limitation to the flexibility of the *start_monitoring* internal action, as a monitor started later in an agent's reasoning process may not be aware of the past states that occurred before its initialization. However in future work, we plan to have a single monitor handling multiple rules at a time, eliminating this limitation.

The *start_monitoring* internal action takes in the following parameters:

monitoring_mode: This is either "fulf" or "viol" to indicate whether the rule of expectation is to be monitored for fulfilment or violation. For example, with respect to the expectation monitor we are currently employing, the logic of the

internal action generates a Fulf formula if the parameter refers to "fulf" and a Viol formula if the parameter refers to "viol".

expectation_name: This specifies a name for the expectation, for ease of future reference.

monitor_tool: This identifies the monitoring tool that should be used to monitor this expectation.

condition: This specifies the requirements on the past and present that activate monitoring for the expectation.

expectation: This specifies the actual expectation.

context_information_list: This argument can be used to assign any other contextual information that might be useful for monitoring an expectation. For example, we can specify a specific agent or a group of agents to be monitored. This information will be added to any fulfilment belief or violation belief sent to the agent as a result of monitoring the expectation.

Terminating Expectation Monitoring: We have made it possible for an agent to stop monitoring for an expectation if the need arises to do so during its reasoning process. The internal action *stop_monitoring* stops the monitoring of the expectation. It takes the following parameters:

expectation_name: The name of the expectation

monitor_tool: The monitoring tool that is currently running the specified expectation.

5.2 Representing Expectation Fulfilments and Violations in Jason

An important design consideration is how to encode the fulfilments and violations identified by the external expectation monitor as Jason events to Jason. The Environment class in Jason acts as the interface to integrate the Jason platform with outside simulation environments. Therefore the Environment class was selected as the best option to communicate the detected fulfilments and violations to Jason agents. Just like percepts, these result in new beliefs that lead to the execution of plans that handle the detected fulfilment or violation.

We define the structure of beliefs based on the detected fulfilments and violations as follows:

$$\mathsf{fulf}(Name, StateId)[rule(Cond, Exp), rule\_triggered\_in\_state(OldStateId), context(Context)]$$

$$\mathsf{viol}(Name, StateId)[rule(Cond, Exp), rule\_triggered\_in\_state(OldStateId), context(Context)]$$

Here, *fulf* encodes the fulfilment of an expectation, while *viol* represents a violation.

The variable *Name* represents the name assigned to a particular fulfilment or violation detected, and the *StateId* represents the identifier for the state in which the actual fulfilment or violation of the expectation occurred. The notion

of a state is important because fulfilments and violations arise in a particular temporal context that is encapsulated by the state identifier. It is up to the monitor to provide an appropriate form of state identifier.

In Jason, a percept with the same content as an already existing belief will not lead to the generation of a new belief. However, we want a fulfilment or a violation detected in one iteration of the Jason reasoning cycle to be distinct from the same fulfilment or violation detected in the previous cycle (e.g. two different robberies in consecutive states are two different crimes). This requirement can be accomplished with the state number associated with the fulfilment (and violation) beliefs.

When creating beliefs in Jason, an agent programmer can add any other variable of importance using 'annotations'. These annotations can be omitted when specifying the triggering event for a plan if the context and the body of the corresponding plan do not need this information. In the above predicates, we have defined three annotations. The first annotation represents the actual rule that was fulfilled or violated. It has two parameters: the condition that triggers the expectation and the actual expectation. These can be defined in any format according to the expectation monitor in use. The second annotation is 'rule_triggered_in_state' which identifies the state in which the condition of the expectation became true. The third annotation is the list of contextual information that is related to this identified fulfilment or violation. The context information list that was generated for the related expectation when it was initiated by star_monitoring internal action is used to provide information in this annotation.

5.3 Extended Jason Semantics

In this section, we present the extended Jason semantics which includes the operation of the expectation monitor.

In Jason, the state of an agent is determined by the *belief base*, the *set of events*, the *plan library* and the *set of intentions*. With our extension, an agent can have a set of expectation monitors that are active on behalf of it, which operate external to the Jason core logic. A monitor has its own state, which is different from an agent's state. For simplicity, we only model a single expectation monitor in the semantics. Incorporating multiple monitors is a straightforward extension.

An expectation monitor can have many 'monitor tasks', distinguished by their unique name. Each monitor task is comprised of a rule (a rule resembles an expectation, and its triggering condition), and a property which states whether the rule should be monitored for its fulfilment or violation. Associated with a monitor, there is also a history component, which resembles the set of input states received by the monitor. We also define a *set of notifications*, which becomes the output of the expectation monitor. The set of notifications resembles the set of states where the expectation monitor recorded a fulfilment or violation for any of the monitor tasks that are currently being monitored. These notifications are eventually consumed by the agent.

An expectation monitor is represented by the triple $\langle H, MTs, Ns \rangle$, where:

- H is the history of the monitor. As mentioned earlier, H resembles the set of input states received by the monitor. The input states have a state identifier, and some associated information of the world in a representation specific to the expectation monitor being used.
- MTs is the set of monitor tasks associated with the expectation monitor. A monitor task MT is a 4-tuple of the form $\langle Na, Cn, Ex, Pr \rangle$. Here Na refers to the unique name assigned to the monitor task. The Cn and Ex parameters represent a rule, where Cn represents the condition specifying when an expectation becomes active and Ex refers to the actual expectation. Pr is the property which has the grammar $Pr := FULF|VIOL$, meaning that the property refers to the fulfilment or violation of a rule.
- Ns is the map of notifications generated by the expectation monitor as the output. This map associates state identifiers with sets of pairs $\langle Na, Pr \rangle$ where each pair expresses the information that in the given state, the monitor task named Na resulted in a detected event of type Pr ($FULF$ or $VIOL$).

This abstract model of a monitor can be related to the semantics of a specific monitor tool as shown by the following example rule. This shows how the model theoretic semantics (top left) of our chosen monitor [3] is related to the emission of a fulfilment notification. A similar rule can be defined to explain the emission of violation notifications.

$$\frac{H, \varnothing, |H| \models \mathsf{Fulf}(Cn, Ex) \quad \langle Na, Cn, Ex, FULF \rangle \in MTs}{\langle H, MTs, Ns \rangle \rightarrow \langle H, MTs, Ns' \rangle}$$

where

$Ns' = Ns \cup (|H| \mapsto \langle Na, FULF \rangle)$ if $|H|$ is not a key in Notifications,
or
$Ns' = map\_update(Ns, |H|, Ns[|H|] \cup \langle Na, FULF \rangle)$ otherwise.

In this rule, we assume that history states are identified by their (1-based) indices, so $|H|$ (the length of the history H) is the identifier for the final state in the history.

The rule states that when a fulfilment formula logically holds in the logic used by the monitor[5], and the corresponding rule is being monitored, a fulfilment notification is emitted for the current state (the last in the history). The notification map is updated either by adding a new mapping $|H| \mapsto \langle Na, FULF \rangle$ to the monitor notifications, or by adding $\langle Na, FULF \rangle$ to the notifications for state $|H|$ if any exist.

In the Jason semantics, the transition relation of an agent's configuration is given by a set of conditional rules that change the agent's configuration in each of the steps of the reasoning cycle. The configuration for an agent is represented by the tuple $\langle ag, C, M, T, s \rangle$ [4], where:

[5] The details of this particular monitor's semantics [3] are outside the scope of this paper.

- ag refers to the agent program, which consists of a set of beliefs and a set of plans
- C is an agent's circumstance, denoted by the tuple $\langle I, E, A\rangle$, with I being the set of intentions, E the set of events and A being the set of actions to be performed in the environment.
- M is a tuple $\langle In, Out, SI\rangle$ that registers different aspects of communicating agents. Here, In is the message inbox of an agent, Out is the out-going message box, and SI keeps track of the suspended intentions related to the communication messages that are currently being processed.
- T is a structure that stores temporary data required in various steps of the reasoning cycle. This is a tuple $\langle R, Ap, i, \epsilon, \rho\rangle$, where R represents the relevant plans, Ap represents the set of applicable plans, and i, ϵ, ρ respectively represent an intention, event and an applicable plan that are being considered along the execution of one reasoning cycle.
- s is the current step (or state) in the agent reasoning cycle shown in Figure 1, where:
  $s \in \{\mathsf{ProcMsg}, \mathsf{SelEv}, \mathsf{RelPl}, \mathsf{ApplPl}, \mathsf{SelAppl}, \mathsf{AddIM}, \mathsf{SelInt}, \mathsf{ExecInt}, \mathsf{ClrInt}\}$.

Subscripts are used to identify individual components of tuples, e.g. C_E denotes the events set within a configuration C, and the notation $i[p]$ is used to denote an intention consisting of plan p on top of intention i.

To define the semantics of our Jason extension we must address three issues: i) the effect of the new internal actions start_monitoring and stop_monitoring, and ii) how notifications emitted from the monitor are communicated to Jason as beliefs. There is a third issue that we consider out of the scope of these semantics: the process that adds states to the monitor's history. This is because the Jason agent is not responsible for sending percepts to the monitor, and our architecture does not even assume that the monitor receives state information from the Jason environment object—it may have its own separate mechanism for obtaining information from the system in which the Jason agent is situated[6].

In the rules below we define transitions on an extended system configuration comprising the Jason agent and the monitor. This is a pair $\langle AG, EM\rangle$, where $AG = \langle ag, C, M, T, s\rangle$ and EM represents the expectation monitor as defined above.

start_monitoring :

Through the start_monitoring internal action, an expectation monitor is started and is added to the set of active expectation monitors of the agent.

The start_monitoring internal action takes place when the body of an agent plan is being executed, and this internal action becomes the current intended means to be executed. This refers to the ExecInt step in the reasoning cycle in Figure 1. The internal action executes completely (i.e. without suspension, which is the normal procedure for executing internal actions) and returns.

[6] This is the case for our work on integrating this extended version of Jason with the Second Life virtual world [11]

The Jason semantics for this action is shown below.

$$\frac{T_i = i[head \leftarrow start\_monitoring(Mm, En, Mt, Cn, Ex, Cil); h]}{\langle\langle ag, C, M, T, \mathsf{ExecInt}\rangle, EM\rangle \rightarrow \langle\langle ag, C, M, T', \mathsf{ClrInt}\rangle, EM'\rangle}$$

Where:

- Parameters Mm, En, Mt, Cn, Ex and Cil respectively refer to monitoring_mode, expectation_name, monitor_tool, condition, expectation and context_information_list, as defined in Section 5.1
- Here $EM'_{MTs} = EM_{MTs} \cup \{\langle En, Cn, Ex, Mm\rangle\}$
- $T'_i = i[head \leftarrow h]$

As in the standard Jason semantics, where a transition is defined as transforming a structure S into a new version S', all components of S' are assumed to be the same as those in S except where otherwise specified.

stop_monitoring : The stop_monitoring internal action also takes place when the body of an agent plan is being executed, and this internal action becomes the current intended means to be executed. This refers to the ExecInt step in the reasoning cycle, and it moves the transition to the state ClrInt.

$$\frac{T_i = i[head \leftarrow stop\_monitoring(En, Mt); h]}{\langle\langle ag, C, M, T, \mathsf{ExecInt}\rangle, EM\rangle \rightarrow \langle\langle ag, C, M, T', \mathsf{ClrInt}\rangle, EM'\rangle}$$

where:

- Parameters En and Mt respectively refer to the expectation_name and monitor_tool as defined in Section 5.1
- $EM'_{MTs} = EM_{MTs} \setminus \{MT\}$. In other words, the stop_monitoring internal action removes the monitor task MT referenced by En (here, the expectation_name refers to the unique name of the monitor task) from the expectation monitor.
- $T'_i = i[head \leftarrow h]$

Though not included in the paper, we also modify the condition of the existing Jason semantic rule for handling internal actions to exclude it from applying the standard operational semantics in the case that the selected action a is start_monitoring or stop_monitoring.

Handling Fulfilment and Violation Notifications Whenever the monitor identifies the fulfilment or violation of a rule defined in it, it sends a notification to Jason. These notifications are treated as Jason percepts and subsequently result in new belief events. We use the function *NotBels* to denote the process that converts monitor notifications into belief events using the syntax defined in Section 5.2, and the corresponding rule for this transition can be written as:

$$\frac{EM_{Ns} \neq \varnothing}{\langle\langle ag, C, M, T, s\rangle, EM\rangle \rightarrow \langle\langle ag, C', M, T, s\rangle, EM'\rangle}$$

In this rule, $EM'_{Ns} = \varnothing$ and $C'_E = C_E \cup NotBels(EM_{Ns})$.

Here, EM'_{Ns} refers to the set of notifications belonging to all the monitor tasks active for that expectation monitor.

This rule is not executed as part of the Jason agent's reasoning cycle. Rather, it represents a separate process that consumes notifications from the monitor and adds them as new events for the agent to process. This process runs concurrently with the Jason interpreter, and we do not assume any sychronisation between the two processes (except to avoid concurrent modification of the agent's input event set C_E). Therefore this rule can be applied in any state of the agent[7].

6 Example Scenario - A Jason Agent Engaged in the Football Team Play Scenario "Give and Go"

We have integrated this Jason extension with the popular virtual world Second Life [1], with the use of a framework we have developed [11] for integrating agents with Second Life. This enables the implementation and testing of sophisticated Jason agents.

In this example, we demonstrate how the ability of a Jason agent to monitor and detect fulfilments and violations of its expectations is useful in its decision making process. We implement this example in the SecondFootball [2] virtual simulation in Second Life which enables playing virtual football. This system provides scripted stadium and ball objects that can be deployed inside Second Life, as well as a "head-up display" object that an avatar can wear to allow the user to initiate kick and tackle actions.

In this example, we implement the "give and go" team play scenario described in Section 4. Here, the Jason agent Ras_Ruby is engaged in the team play scenario with the player Su_Monday, who is controlled by a human. When Ras_Ruby receives the ball, it adopts the expectation which states that Su_Monday should run until she reaches the PenaltyB area, so that she can pass the ball back to Su_Monday for her to attempt a goal score at Goal B.

When the system starts, the Jason agent corresponding to Ras_Ruby is initialized. When the Jason agent starts executing, it first tries to log itself into Second Life. After sending its login request, the agent has to wait till it gets the confirmation of the successful login. When it receives the successful login notification, the agent adopts the new goal to walk to the area MidfieldB2. The corresponding plan for this goal addition is shown below (+! denotes a goal addition event, a context condition appears after the colon, and the arrow operator separates the head and body of the plan).

```
+!check_connected: connected  <-
  action("walk", "MidfieldB2").
```

[7] In practice, the monitor's notifications are recorded as percepts in the Jason Environment object, and the agent perceives them via Jason's belief update phase. However, Jason's operational semantics do not include a state for perceiving the environment, so here we model the connection between the monitor and the agent as a separate process that pushes fulfilment and violation beliefs to the agent.

Once in the area MidfieldB2, the agent Ras_Ruby waits for Su_Monday to kick and pass the ball to it. Once it successfully receives the ball, the agent gets the "successful_kick(su_monday, ras_ruby)" percept (which is generated by our Second Life integration framework and states that Su_Monday successfully passed the ball to Ras_Ruby through a kick), and this triggers the corresponding plan related to this belief addition, as given below.

```
+successful_kick(su_monday,ras_ruby)
   <-
   //internal actions
   .start_monitoring("fulf",
      "move_to_target",
      "expectation_monitor",
      "#once",
      "('U',
         'advanceToGoalB(su_monday)',
         'penaltyB(su_monday)')",
      []);

   .start_monitoring("viol",
       "move_to_target",
       "expectation_monitor",
       "#once",
       "('U',
         'advanceToGoalB(su_monday)',
         'penaltyB(su_monday)')",
      []).
```

This plan uses the new internal actions introduced in 5.1 for monitoring for the fulfilment and violation of the agent's expectation. Here, in the first parameter we define the type of expectation; in the first call to the internal action, it is of the type 'fulfilment' (fulf), and in the second, it is of the type 'violation' (viol). In the second parameter, the name of the expectation is given as 'move_to_target'. The third parameter is the name of the expectation monitor used. The fourth parameter is the triggering condition for the expectation, and as explained in 4, it is a keyword with a special meaning (`#once`). The BDI execution cycle only executes a single step of a plan at each iteration, and any knowledge of the current state of the world retrieved by the plan may be out of date by the time the monitor is invoked. Therefore the `#once` keyword instructs the monitor to insert a nominal for the current state of the world just before the rule begins to be monitored. The expectation formula referred by the fifth parameter resembles the formulae presented in Section 4, and it states that Su_Monday should advance towards GoalB (`'advanceToGoalB(su_monday)'`), until `('U')` she reaches PenaltyB, denoted by `'penaltyB(su_monday)'`. We do not utilize the optional sixth parameter in this example.

The fulfilment of this expectation occurs when Su_Monday advances towards GoalB until she reaches PenaltyB. Similarly, the violation of this expectation

occurs if Su_Monday stopped somewhere before reaching PenaltyB, or she moves in the opposite direction before reaching PenaltyB area[8].

If Su_Monday fulfilled Ras_Ruby's expectation, the expectation monitor detects this and reports back to the Jason agent, which results in a fulfilment belief. The following plan handles this detected fulfilment and instructs the avatar to carry out the kick action[9].

```
+fulf("move_to_target", X)
   <-
   //Calculate kick direction and force, turn, then ...
   action("animation", "kick").
```

On the other hand, if Su_Monday violated the expectation, the expectation monitor reports the violation to the Jason agent, generating a violation belief for the agent. The agent uses the first plan below to decide the agent's reaction to the detected violation, which creates a goal to choose a new tactic for execution. The second plan (responding to this new choose_and_enact_new_tactic) is then triggered, and the agent adopts the tactic of attempting to score a goal on its own by running towards the PenaltyB area with the ball.

```
+viol("move_to_target",X)
   <-
   !choose_and_enact_new_tactic.

+!choose_and_enact_new_tactic
   <-
   action("run", "penaltyB").
```

7 Conclusion

This paper addressed the importance of agents having a capability to directly monitor their expectations and detect the fulfilments and violations of these expectations, and respond accordingly.

We demonstrated a tight integration of expectation monitoring in the BDI agent model and presented an implemented mechanism to monitor expectations of individual agents in the Jason agent model. Also, we identified this as an approach to focus on monitoring at the individual agent level, as opposed to the organizational level monitoring that has received the main focus in the past research. With our approach, individual Jason agents can monitor for the fulfilment and violation of their own expectations, and can react to the identified fulfilments and violations by having plans that are triggered by those events.

[8] The conditions and expectations are defined in temporal logic and we do not wish to elaborate on them in the scope of this paper. These are written as nested Python tuples, as this is the input format for the expectation monitor written in Python.

[9] Due to technical problems the Second Life avatar cannot currently perform the actual 'kick' animation

As future work, it is interesting to investigate ways of how agents can publish their expectations to make other agents in the society aware of their personal expectations, and how agents should react to the detected fulfilments and violations of their expectations, both in a social context and with respect to their emotions. Moreover, it should also be investigated how agents can use their expectations as well as expectations of other agents in the society that they are aware of, proactively in their deliberation process.

References

1. Linden Lab. Second Life Home Page. `http://secondlife.com`, August 2010.
2. Vstex Company. Secondfootball Home Page. `http://www.secondfootball.com`, August 2010.
3. F. Bacchus and F. Kabanza. Using Temporal Logics to Express Search Control Knowledge for Planning. *Artificial Intelligence*, 116(1-2):123–191, 2000.
4. R. H. Bordini, J. F. Hubner, and M. Wooldridge. *Programming Multi-Agent Systems in AgentSpeak using Jason*. John Wiley & Sons Ltd, England, 2007.
5. C. Castelfranchi. Mind as an anticipatory device: For a theory of expectations. In M. De Gregorio, V. Di Maio, M. Frucci, and C. Musio, editors, *Brain, Vision, and Artificial Intelligence*, volume 3704 of *Lecture Notes in Computer Science*, pages 258–276. Springer Berlin / Heidelberg, 2005.
6. C. Castelfranchi, F. Giardini, E. Lorini, and L. Tummolini. The prescriptive destiny of predictive attitudes: From expectations to norms via conventions. In *Proceedings 25th Annual Meeting of the Cognitive Science Society (CogSci 2003), Boston, USA, 31 July 2 August*, 2003.
7. S. Cranefield and M. Winikoff. Verifying social expectations by model checking truncated paths. *Journal of Logic and Computation*, 2010. Advance access, doi: 10.1093/logcom/exq055.
8. S. Cranefield, M. Winikoff, and W. Vasconcelos. Modelling and monitoring interdependent expectations. Discussion Paper 2011/03, Department of Information Science, University of Otago, 2011. `http://eprints.otago.ac.nz/1094/`.
9. E. Lorini and R. Falcone. Modeling expectations in cognitive agents. In *AAAI 2005 Fall Symposium: From Reactive to Anticipatory Cognitive Embodied Systems*, 2005.
10. F. Meneguzzi, S. Miles, M. Luck, C. Holt, and M. Smith. Electronic contracting in aircraft aftercare: a case study. In *Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems: industrial track*, AAMAS '08, pages 63–70, Richland, SC, 2008. International Foundation for Autonomous Agents and Multiagent Systems.
11. S. Ranathunga, S. Cranefield, and M. Purvis. Interfacing a Cognitive Agent Platform with a Virtual World: a Case Study using Second Life. In *10th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2011)*, 2011. To appear.
12. A. S. Rao. BDI agents speak out in a logical computable language. In *Proceedings of the 7th European workshop on Modelling autonomous agents in a multi-agent world: agents breaking away*, pages 42–55. Springer-Verlag Berlin, Heidelberg, 1996.

Chapter 4

Model Checking

Abstraction for Model Checking Modular Interpreted Systems over ATL

Michael Köster and Peter Lohmann

Computational Intelligence Group, Clausthal University of Technology
Julius-Albert-Str. 4, 38678 Clausthal-Zellerfeld, Germany
mko@tu-clausthal.de

Theoretical Computer Science, Leibniz University Hannover
Appelstr. 4, 30167 Hannover, Germany
lohmann@thi.uni-hannover.de

Abstract. We present an abstraction technique for model checking multi-agent systems given as modular interpreted systems (MIS) (introduced by Jamroga and Ågotnes). MIS allow for succinct representations of compositional systems, they permit agents to be removed, added or replaced and they are modular by facilitating control over the amount of interaction. Specifications are given as arbitrary ATL formulae: We can therefore reason about strategic abilities of groups of agents.
Our technique is based on collapsing each agent's local state space with handcrafted equivalence relations, one per strategic modality. We present a model checking algorithm and prove its soundness: This makes it possible to perform model checking on abstractions (which are much smaller in size) rather than on the concrete system which is usually too complex, thereby saving space and time. We illustrate our technique with an example in a scenario of autonomous agents exchanging information.

1 Introduction

Multi-agent systems (MAS) and their logical frameworks have attracted some attention in the last decade. Agent logics have been used to reason about knowledge, time, strategic abilities, coordination and cooperation [13,10,1]. An important technique for verifying properties of a system is model checking [5], which has been refined and improved over the last years.

While an important feature of a MAS is its modularity, e.g., removing, replacing, or adding an agent, only a few of the existing compact representations are both modular, computationally grounded [22] and allow the system designer to represent knowledge and strategic ability. Among these few approaches are Modular Interpreted Systems (MIS) [17], which we modify a bit, and use it to apply our abstraction techniques. MIS are inspired by interpreted systems [12,13] but achieve a modularity and compactness property much like concurrent programs [18], i.e., they are modular, compact and computationally grounded while allowing at the same time to represent strategic abilities.

Although explicit models (and symbolic representations [21,20]) achieve the second part very well (because the semantics are defined over them) some problems arise with the first part: Usually temporal models have an exponential number of states and, in addition, they do not support modularity since there is no easy way to remove or replace an agent. Interpreted systems [12,13], however, have a modular state space. But they use a joint transition function for modelling temporal aspects of the system and are thus not modular wrt actions. In contrast, concurrent programs [18] are both modular and compact not only wrt the states but also wrt the actions. However, in the context of a MAS it is important that actions can have side effects on the states of other agents as well and this behaviour is difficult to model with concurrent programs ([17] contains a detailed comparison). Finally, our choice of using MIS to model MAS is, although motivated by the above reasons, still arbitrary to some extent and our techniques could certainly be used with other formalisms as well.

A major obstacle to model checking real systems is the state explosion problem. As model checking algorithms require a search through the state space of the system, the efficiency of any algorithm highly depends on the size of this state space. While for small problems this is still feasible, for larger state spaces it soon becomes intractable. We therefore need to eliminate irrelevant states by using appropriate abstraction techniques [4] which guarantee that the property to be verified holds in the original system if it holds for the abstract system. We present such an abstraction technique for MIS. More precisely, we reduce the local state space of each agent in a MIS. We do this by using hand crafted equivalence relations because, clearly, there cannot be a generic automatizable abstraction technique: Model checking ATL for MIS is *EXPTIME*-complete, therefore in the worst case there are instances where no abstraction technique at all is applicable.

While abstraction of reactive systems for temporal properties is a lively research area [2,3,7,19], there are only a few approaches when it comes to MAS and even fewer concerning an abstraction technique for dealing with strategic abilities. One interesting approach by Cohen et al. [6] achieves an abstraction that preserves temporal-epistemic properties. However, the abstraction is based on an interpreted system to model the MAS and therefore limits the modularity of the MAS. Several other abstraction approaches for epistemic properties (cf. [8,11]) are either not computationally grounded or use an explicit representation of the model.

Another approach by Henzinger et al. [15] shows how to use abstraction for symbolic model checking of alternating-time μ-calculus formulae over MAS given as alternating transition systems. Their technique is quite similar to ours but still more restricted in an important way. They assume that there are only two agents present and then use a single abstraction to model check the whole formula. Our approach allows for multiple agents and for many abstractions (one per strategic operator). Hence, we allow for a much finer control over what information is abstracted away but still preserve soundness of our model checking algorithm.

Note that we will assume the existence of handcrafted equivalence relations, e.g. generated from manual annotations of program code, since any automatic abstraction generation or refinement (as in [14] for two-player games) can only work in typical cases but not in the worst case. That is also the reason why we do not work out how our algorithms can be implemented fully symbolically: in the worst case it will be as bad as a non-symbolic algorithm. We are not trying to neglect the usefulness of either of those techniques but our focus lies on something else: a provable upper bound for the runtime which is exponential in the sum of the sizes of the abstract systems but linear in the size of a succinct representation of the concrete system (see Theorem 11). The exponential part of this is not as bad as it sounds because, as argued above, our technique allows for more than one abstraction and therefore each abstraction can be quite small and still much of the relevant information of the whole concrete system can be preserved for the overall model checking process.

Finally, the abstraction for MIS which we present in this paper is motivated by the idea of an IT ecosystem [9], i.e., a system composed of a large number of distributed, decentralized, autonomous, interacting, cooperating, organically grown, heterogeneous, and continually evolving subsystems. In such an ecosystem, which can be seen as a MAS, it is important to verify safety, fairness and liveness properties in order to control the stability of it. A non-trivial demonstrator (mentioned in [9]) describes one instance of such a system by introducing a fictional scenario, namely a smart airport. In that airport there exist many agents doing different things, e.g., carrying your bags, buying flight tickets or exchanging pictures about the travel destination. Among many other things some agents at the airport want to share some information with other agents. Assuming that no direct agent-to-agent connection is possible, the agents have to send the information to some middleman that forwards the message. Obviously this communication protocol raises some questions about safety, fairness and liveness properties. While examining these properties, i.e., model checking the whole system (consisting of many agents and therefore many states), is intractable, model checking a MIS allows us to concentrate on just the agents that have to communicate. Using our abstraction method it is sufficient to model-check a fairly small subset of the original system. We will use this scenario as a running example throughout the paper.

The structure of the paper is as follows: First we present the background of our work. Section 2 recalls the MIS framework and describes our modifications. We extend this section by formulating the communicating agents example as a MIS. In Section 3 we introduce the logic ATL. The main contributions of this paper are in Section 4, Section 5 and Section 6: We design an abstraction technique for MIS, then construct a model checking algorithm based on this technique and conclude with a soundness proof as well as a complexity analysis of the algorithm. Section 7 illustrates the abstraction technique by an example. Finally, Section 8 summarizes the results and discusses future work.

2 Modular Interpreted Systems

We model multi-agent systems in the framework of Modular Interpreted Systems (MIS) [17]. Each agent is described by a set of possible local states, i.e., states it can be in, and a function that calculates the available actions in a certain state. A local transition function specifies how an agent evolves from one local state to another. States are labeled with a set of propositional symbols by an associated labeling function. Finally, an agent is equipped with a function that defines the possible influences of an agent's action on its environment, i.e. the other agents, and a function for the influence of the environment on this particular agent.

Definition 1. *A* Modular Interpreted System (MIS) *is a tuple* $S = (\mathbb{A}\mathrm{gt}, Act, \mathcal{I}n)$ *where* Act *is the set of actions all agents can perform.* $\mathcal{I}n$ *is called interaction alphabet. It describes the interaction between the agent and its environment. Finally,* $\mathbb{A}\mathrm{gt} = \{a_1, \ldots, a_k\}$ *is a set of agents where an agent is a tuple* $a_i = (St_i, d_i, out_i, in_i, o_i, \Pi_i, \pi_i)$ *with*

- St_i *is the local state space. It is a non-empty set of possible local states for agent* a_i*.*
- $d_i : St_i \to \mathfrak{P}(Act)$ *defines for each state in* St_i *the available actions for agent* a_i*. With* $\mathfrak{P}$ *we denote the power set.*
- $out_i : St_i \times Act \to \mathfrak{P}(\mathcal{I}n)$ *defines the possible influences (one is then chosen non-deterministically) of agent* a_i*'s action (executed in a certain local state) on its environment.*[1] *Intuitively, this describes the external effect of an action which agent* a_i *is executing.*
- $in_i : St_i \times \mathcal{I}n^{k-1} \to \mathfrak{P}(\mathcal{I}n)$ *defines the possible influences (one is then chosen non-deterministically) of its environment on this agent.*[1] *It maps the external effects of the actions of all other agents to the influence these actions might have on the agent in a particular state.*
- $o_i : St_i \times Act \times \mathcal{I}n \to \mathfrak{P}(St_i)$ *is a local (non-deterministic) transition function.*[2]*.*
- Π_i *are the local propositions, where* Π_i *and* Π_j *are disjoint when* $i \neq j$*.*
- $\pi_i : St_i \to \mathfrak{P}(\Pi_i)$ *is a valuation (local labeling function) of these propositions.*

The global state space is defined as $St := St_1 \times \cdots \times St_k$*.*

Example 2. We consider a system with several autonomous agents which can gather information about their environment and share that information between each other if they are in communication range. We consider groups of them working together as teams.

Our example consists of six agents $\{a_1, a_2, a_3, a_4, b_1, b_2\}$ partitioned in two teams $A = \{a_1, a_2, a_3, a_4\}$ and $B = \{b_1, b_2\}$ and where the agents' locations are

[1] This is different from the original MIS definition in so far as we have a set of possible influences and the authors had one deterministic influence symbol; it is changed to cope with possible ambiguities when doing abstraction later.

[2] Non-deterministic as opposed to in the original definition; it inherits the non-determinism from in_i and adds additional non-determinism to cope with abstraction of states later.

such that each agent a_i can reach each agent b_j but no two agents from the same team can reach each other (see Figure 1). Agents of team A can send a message (if they already know it) to b_1 or b_2 or choose to do nothing. Agents of team B, however, are not allowed to send a message back to its sender agent. Once an agent a_i sent a message to an agent b_j the agent b_j is not allowed to send it to a_i in any future round. Additionally, if an agent b_j has received a message from a_i then it has to send it to some agent a_k in the following round (unless this contradicts the former rule) and if possible k has to be greater than i.

Now agent a_1 has learned something and wants to communicate its newly gathered knowledge to its team member a_4. The difficulty is that the message has to pass through an agent of the other team. But we will see that it is still possible for team A to ensure that a_4 will know the message eventually. Formally we have the following MIS

$$\begin{aligned} S := (\mathbb{A}\mathrm{gt} = \{a_1, a_2, a_3, a_4, b_1, b_2\}, Act = \{\mathrm{send_x} \mid \mathrm{x} \in \mathbb{A}\mathrm{gt}\} \cup \{\mathrm{noop}\}, \\ \mathcal{I}n = \{\mathbf{nothing}, \mathbf{m_{a_1}}, \mathbf{m_{a_2}}, \mathbf{m_{a_3}}, \mathbf{m_{a_4}}\} \\ \cup \mathfrak{P}(\{\mathbf{m_{a_i b_j}} \mid i \in \{1, \dots, 4\}, j \in \{1, 2\}\})) \end{aligned}$$

with $a_i := (St_{a_i}, d_{a_i}, out_{a_i}, in_{a_i}, o_{a_i}, \Pi_{a_i}, \pi_{a_i})$ where

- $St_{a_i} = \{k(nown), u(nknown)\}$
- $\Pi_{a_i} = \{\mathsf{known_{a_i}}, \mathsf{unknown_{a_i}}\}$
- $\pi_{a_i} : k \mapsto \{\mathsf{known_{a_i}}\}, \quad u \mapsto \{\mathsf{unknown_{a_i}}\}$
- $d_{a_i} : k \mapsto \{\mathrm{send_{b_1}}, \mathrm{send_{b_2}}, \mathrm{noop}\}, \quad u \mapsto \{\mathrm{noop}\}$
- $out_{a_i} : (k, \mathrm{send_{b_j}}) \mapsto \{\{\mathbf{m_{a_i b_j}}\}\}$
  $(k, \mathrm{noop}) \mapsto \{\mathbf{nothing}\}$
  $(u, \mathrm{noop}) \mapsto \{\mathbf{nothing}\}$
  for all $j \in \{1, 2\}$,
- $in_{a_i} : (s, \gamma_1, \dots, \gamma_5) \mapsto \begin{cases} \{\mathbf{m_{a_i}}\} & \text{if } \mathbf{m_{a_i}} \in \{\gamma_1, \dots, \gamma_5\} \\ \{\mathbf{nothing}\} & \text{else} \end{cases}$
  for all $s \in St_{a_i}$, $\gamma_1, \dots, \gamma_5 \in \mathcal{I}n$,
- $o_{a_i} : (k, \alpha, \gamma) \mapsto \{k\}$
  $(u, \alpha, \mathbf{nothing}) \mapsto \{u\}$
  $(u, \alpha, \mathbf{m_{a_i}}) \mapsto \{k\}$
  for all $\gamma \in \mathcal{I}n$ and $\alpha \in Act$.

For the agents b_j we have $b_j := (St_{b_j}, d_{b_j}, out_{b_j}, in_{b_j}, o_{b_j}, \Pi_{b_j}, \pi_{b_j})$ where

- $St_{b_j} = \mathfrak{P}(\{r_1, \dots, r_4\}) \times \mathfrak{P}(\{n_1, \dots, n_4\})$,
- $\Pi_{b_j} = \{\mathsf{known_{b_j}}, \mathsf{unknown_{b_j}}\}$,
- $\pi_{b_j} : (R, N) \mapsto \begin{cases} \{\mathsf{known_{b_j}}\} & \text{if } R \neq \emptyset \\ \{\mathsf{unknown_{b_j}}\} & \text{else} \end{cases}$,
- $d_{b_j} : (R, N) \mapsto \begin{cases} \{\mathrm{noop}\} & \text{if } R = \emptyset \\ \left\{ \mathrm{send_{a_i}} \,\middle|\, \begin{array}{l} r_i \notin R \text{ and there is no } k \geq i\text{:} \\ n_k \in N \text{ and } \exists \ell > k\text{: } r_\ell \notin R \end{array} \right\} \cup \begin{cases} \{\mathrm{noop}\} & \text{if } N = \emptyset \text{ or } R = \{r_1, \dots, r_4\} \\ \emptyset & \text{else} \end{cases} & \text{else} \end{cases}$,
- $out_{b_j} : ((R, N), \mathrm{send_{a_i}}) \mapsto \{\mathbf{m_{a_i}}\}$
  $((R, N), \mathrm{noop}) \mapsto \{\mathbf{nothing}\}$
  for all $i \in \{1, \dots, 4\}$,

– $in_{b_j} : ((R,N),\gamma_1,\ldots,\gamma_5) \mapsto$
$$\begin{cases} \{\{\mathbf{m_{a_i b_j}} \mid \{\mathbf{m_{a_i b_j}}\} \in \{\gamma_1,\ldots,\gamma_5\}\}\} \\ \quad \text{if there is } i \in \{1,\ldots,4\} \text{ with } \{\mathbf{m_{a_i b_j}}\} \in \{\gamma_1,\ldots,\gamma_5\} \\ \{\mathbf{nothing}\} \quad \text{else} \end{cases}$$
for all $\gamma_1,\ldots,\gamma_5 \in \mathcal{I}n$,
– $o_{b_j} : ((R,N),\alpha,M) \mapsto (R \cup \{r_i \mid \mathbf{m_{a_i b_j}} \in M\}, \{r_i \mid \mathbf{m_{a_i b_j}} \in M\})$
$((R,N),\alpha,\mathbf{nothing}) \mapsto (R,\emptyset)$
for all $\alpha \in Act$, $\emptyset \neq M \subseteq \{\mathbf{m_{a_1 b_j}},\ldots,\mathbf{m_{a_4 b_j}}\}$,
for all $(R,N) \in St_{b_j}$. R stands for "**r**eceived (now or some time ago)" while N means "received **n**ow".

Figure 2 shows the agent a_1. Each arrow is denoted by an action and an incoming interaction symbol. Outgoing symbols and propositions are omitted for ease of representation. Nevertheless state u should be labeled with $\mathsf{unknown}_{\mathsf{a_1}}$ and k with $\mathsf{known}_{\mathsf{a_1}}$. The agents a_i are fairly simply structured, they consist of two states representing whether the agent knows the message (k) or it does not know it (u). In the former case the agent can send the message to one of the opponents or just do nothing. In the latter case it has to wait for some agent of team B sending the message to it.

The structure of the agent b_j however is more complex since it consists of 256 states. Every state is labeled with $\mathsf{known}_{\mathsf{b_j}}$ if the state name contains at least one r_i, i.e., the agent received some time ago the message from agent a_i. Consequently, states that do not have any r_i are marked as $\mathsf{unknown}_{\mathsf{b_j}}$. Intuitively, while the agent is waiting for a message it does nothing. When it receives a message, i.e., the state contains a n_i it has to send the message to one of the opponents with a higher number than i and with the condition that this agent did not send it to b_j before. If the state contains all r_1 to r_4 the agent does nothing.

We will come back to this concrete example when presenting the logic and the model checking algorithm.

3 Specification Logic ATL

After having outlined the framework with which we model our MAS, we now have to specify a logic to talk about strategic properties of such a system. We recall the syntax of ATL, define some abbreviations and sketch the semantics of ATL for MIS.

Definition 3. *Alternating-time temporal Logic (*ATL*) [1] is a logic that enables reasoning about temporal and strategic abilities of multi-agent systems. The syntax of plain* ATL *is defined by (with $A \subseteq \mathbb{A}\mathrm{gt}$)*

$$\varphi ::= \mathsf{p} \mid \neg\varphi \mid \varphi \vee \varphi \mid \langle\langle A\rangle\rangle \mathbf{X}\varphi \mid \langle\langle A\rangle\rangle \mathbf{G}\varphi \mid \langle\langle A\rangle\rangle \varphi \mathbf{U} \varphi.$$

Informally, $\langle\langle A\rangle\rangle \mathbf{X}\varphi$ means that agents A have a collective strategy to enforce that in the next step φ holds. The operator $\mathbf{X}$ is read "in the ne**x**t state", the symbol $\mathbf{G}$ means "**g**lobally" and $\mathbf{U}$ "**u**ntil". Other Boolean operators are defined by macros in the usual way.

For the model checking algorithm presented in Section 5 we need to split a formula into subformulae.

Definition 4. *For an* ATL *formula* φ *we write* $\mathrm{qsf}(\varphi)$ *for the multiset of all subformulae of* φ *which start with a quantifier* $\langle\langle A\rangle\rangle$.

Note that identical formulae occurring in two different places inside φ occur twice in $\mathrm{qsf}(\varphi)$ and if φ itself begins with a quantifier it is in the multiset as well.

Finally, we need the following notions.

Definition 5. *For a formula* $\varphi = \langle\langle A\rangle\rangle\psi$ *we write* $[\![\varphi]\!]$ *for the set* A *of agents. For an arbitrary formula* φ *and an arbitrary* $\psi \in \mathrm{qsf}(\varphi)$ *let* $\varphi(\psi, w)$ *denote the formula resulting from* φ *by replacing* ψ *with the new proposition* w.

Example 6. Considering the communicating agents example (cf. Example 2) we can ask the following question: Is it possible for team A to ensure that a_4 will know the message eventually? Written in ATL this corresponds to the question whether

$$S, \mathrm{q} \models \langle\langle A\rangle\rangle(\top \mathbf{U} \mathsf{known}_{\mathsf{a}_4})$$

with $A = \{a_1, a_2, a_3, a_4\}$ and q is the global state where a_1 is in state k, all other agents from team A are in state u and the agents from team B are in state $(\emptyset, \emptyset)$, i.e. where only a_1 knows the message.

3.1 Semantics of ATL for MIS

Modular Interpreted Systems can be easily transformed into *concurrent game structures* (CGS, cf. [1]) as shown for deterministic CGS in [17]. The notion of a CGS is a very universal formalism to model MAS but it comes at the cost of not being modular, i.e., CGS have an unstructured global state space. Also, just as MIS, they have for every agent i a function $d_i : St \to Act$ expressing which actions are available to agent i in a certain global state. The global transition function of a CGS takes as input a global state and for every agent a permissible action and outputs a set of possible successor states (one is then chosen non-deterministically), i.e., the influences of an agent on its environment are implicitly given and there is no way to measure or limit that influence in the framework of CGS.

Definition 7. *For a* MIS $S = (\{a_1, \ldots, a_k\}, Act, \mathcal{I}n)$ *with* $a_i = (St_i, d_i, out_i, in_i, o_i, \Pi_i, \pi_i)$ $(1 \leq i \leq k)$ *the corresponding (non-deterministic) concurrent game structure* $ncgs(S) = (\mathbb{A}\mathrm{gt}', St', \Pi', \pi', Act', d', o')$ *is defined as follows:*

- $\mathbb{A}\mathrm{gt}' := \{1, \ldots, k\}$
- $St' := \prod_{i=1}^{k} St_i$
- $\Pi' := \Pi_1 \cup \cdots \cup \Pi_k$
- $\pi'(p) := \{(q_1, \ldots, q_i, \ldots, q_k) \mid q_i \in \pi_i(p)\}$
- $Act' := Act$

- $d'_i((q_1, \ldots, q_k)) := d_i(q_i)$
- $\delta'((q_1, \ldots, q_k), \alpha_1, \ldots, \alpha_k) :=$
  $\{(q'_1, \ldots, q'_k) \mid \forall 1 \leq i \leq k : q'_i \in o_i(q_i, \alpha_i, \gamma_i)$ *for a* $\gamma_i \in in_i(q_i, \gamma_1, \ldots, \gamma_{i-1},$
  $\gamma_{i+1}, \ldots, \gamma_k)$ *for some* $\gamma_1, \ldots, \gamma_{i-1}, \gamma_{i+1}, \ldots, \gamma_k$ *with* $\gamma_j \in out_j(q_j, \alpha_j)\}$

Note that our concurrent game structures extend the definition from [1] in two ways. Firstly we use labeled actions instead of plain numbers. Secondly we allow the transition relation to be non-deterministic. The semantics of a formula $\langle\langle A\rangle\rangle\varphi$ over a non-deterministic CGS is defined with the non-determinism working against the agents in A. The rationale behind this is that $\langle\langle A\rangle\rangle\varphi$ means "the agents A have a combined strategy which enforces φ". Now, to enforce φ this strategy needs to ensure that in each of the possible runs of the system – determined by the other agents' choices and the non-deterministic branching – the formula φ holds.

Model checking ATL for deterministic MIS is *EXPTIME*-complete as stated in [16] and for deterministic CGS it is *PTIME*-complete as stated in [1]. These results still hold for the non-deterministic versions of the structures. This is because in the model checking algorithm from [1, Chapter 4.1] introducing non-determinism only changes the function $\mathrm{Pre}(A, \rho)$ (which for a given set ρ of system states and a given set A of agents outputs the set of system states from which the agents A can enforce that the next state in any run will lie in ρ). And computing Pre does not get more difficult with non-determinism because even in the case of deterministic systems it is already necessary to take into account all transitions in order to compute Pre.

Having defined MIS and ATL we can now present our new abstraction technique.

4 Abstraction for MIS

In general, multi-agent systems have large associated state spaces and even if they are symbolically represented it is infeasible to verify properties by considering *all* reachable states. Nevertheless, interesting properties often only refer to parts of a system. Under this assumption it makes sense to reduce the state space by removing irrelevant states and/or by combining them. Due to the modularity of MIS, we can in a first step easily remove the obviously non-relevant parts of the global state space by removing particular agents while keeping the others. Secondly, we reduce the state space of each agent by abstraction. As in [4] and [6] we do this by partitioning the state space into equivalence classes: Each class collects all concrete states that are equivalent and forms one new abstract state. This new state is labeled by those propositions which are shared by all concrete states. We define the local transition functions of the abstract system in such a way that it behaves just as the concrete one. The set of available actions in an abstract state is decreased so that it only contains actions available in every one of the equivalent concrete states. Finally we show how to handle the interaction with an agents' environment. We start by introducing the definition of an abstraction relation.

Definition 8. *An* abstraction relation *for a* MIS *is a product* $\equiv = \equiv_1 \times \cdots \times \equiv_k$ *where each* $\equiv_i \subseteq St_i \times St_i$ *is an equivalence relation for the states* St_i *of agent* a_i.

For $q \in St_i$, *we write* $[q]_{\equiv_i}$ *for the equivalence class of the local state* q *with respect to* $\equiv_i$. *And for* $q \in St = St_1 \times \cdots \times St_k$, *we write* $[q]_{\equiv}$ *for the equivalence class of the global state* q.

An abstraction relation as in Definition 8 defines for each agent of the MIS, which local states are equivalent and therefore can be condensed to one abstract state. Note that this definition does not say anything about how to define the equivalence, because this depends on the concrete system that is model checked. Therefore these relations have to be handcrafted when modeling a system.

Using the definition of a MIS and Definition 8 we can now specify the abstraction of a system:

Definition 9. *For a* MIS $S = (\mathbb{A}\mathrm{gt}, Act, \mathcal{I}n)$, *an abstraction relation* $\equiv$ *for* S *and a set of* favored *agents* $A \subseteq \mathbb{A}\mathrm{gt}$ *we define the* abstraction *of* S *with respect to* $\equiv$ *and* A *as the* MIS

$$S^A_{\equiv} := (\mathbb{A}\mathrm{gt}', Act, \mathcal{I}n)$$

where $\mathbb{A}\mathrm{gt}' := \{a'_1, \ldots, a'_k\}$ *and* $a'_i = (St'_i, d'_i, out'_i, in'_i, o'_i, \Pi'_i, \pi'_i)$ *with*

i) $St'_i := \{[q]_{\equiv_i} \mid q \in St_i\}$

ii) $d'_i([q]_{\equiv_i}) := \begin{cases} \bigcap_{q' \in [q]_{\equiv_i}} d_i(q') & \text{for } a_i \in A \\ \bigcup_{q' \in [q]_{\equiv_i}} d_i(q') & \text{for } a_i \notin A \end{cases}$

iii) $out'_i([q]_{\equiv_i}, \alpha) := \bigcup_{q' \in [q]_{\equiv_i}} out_i(q', \alpha)$

iv) $in'_i([q]_{\equiv_i}, \gamma_1, \ldots, \gamma_{k-1}) := \bigcup_{q' \in [q]_{\equiv_i}} in_i(q', \gamma_1, \ldots, \gamma_{k-1})$

v) $o'_i([q]_{\equiv_i}, \alpha, \gamma) := \bigcup_{q' \in [q]_{\equiv_i}} \{[q'']_{\equiv_i} \mid q'' \in o_i(q', \alpha, \gamma)\}$

vi) $\Pi'_i := \Pi_i \cap \{p_i \mid \forall q \in \pi_i(p_i) : \forall q' \in [q]_{\equiv_i} : q' \in \pi_i(p_i)\}$

vii) $\pi'_i(p_i) := \{[q]_{\equiv_i} \mid q \in \pi_i(p_i)\}$

for all $q \in St_i$, $\alpha \in Act$, $\gamma, \gamma_1, \ldots, \gamma_k \in \mathcal{I}n$ *and* $p_i \in \Pi'_i$.

Note that there might be $i \in \{1, \ldots, k\}$ *and* $q \in St_i$ *such that* $d'_i([q]_{\equiv_i}) = \emptyset$. *As this would paralyse the system we will from now on assume that the abstraction relation is chosen in such a way that this does not happen.*

Formula i) defines a partition of the local state space by using the handcrafted equivalence relation of this agent. We reduce all equivalent states to just one. Function ii) then computes for this element the available actions by giving agents in A fewer choices and the opponents more choices than before. Due to this construction if a property $\langle\langle A \rangle\rangle \varphi$ (with φ propositional) holds in the abstract system it also holds in the concrete one, since we restricted the actions of the protagonists and extended the set of actions of the antagonists. The possible influences of these actions concerning the environment are calculated by the resulting function iii). It takes for the action α the union of all resulting influence symbols of all states in the equivalence class, i.e., collecting all influence symbols that

are an outcome of executing action α in each state q' of the equivalence class $[q]_{\equiv_i}$. Taking the union is motivated by the fact that executing the same action in equivalent local states results in an equivalent influence on the environment. iv) is defined the same way: We just use the union of in_i for each state in the equivalence class. Moreover, the out_i- and in_i-functions are of the same type for the protagonists as well as the antagonists because they do not introduce deliberate choices by the agents but instead introduce nondeterminism which will – as we will see in Section 5 – always work against the formula which is to be verified. The local transition function v) has to be modified to cope with equivalence classes: It gets as input an abstract state, an action and one (nondeterministically chosen) influence symbol. The output is the set of all equivalence classes that are successors. To determine this we unfold both equivalence classes and check whether there is a connection between a concrete state of the first equivalence class to another concrete state of the second equivalence class.

Finally, we have to define how to label the abstract states (cf. vi) and vii)). We do this by assigning a proposition to an abstract state if all concrete states in the equivalence class are labeled with that proposition. If a proposition only holds in some states of the class we remove it from set of propositions. This ensures that if a proposition is true in an abstract state it is also true in all concrete ones.

In the next section we describe how to evaluate whether a formula holds in a system.

5 The Model Checking Algorithm

Our algorithm takes as input a MIS S, a set *init* of global states of S (the initial states), an ATL formula φ and for each $\psi \in \mathrm{qsf}(\varphi)$ an abstraction relation $\equiv_\psi$. The algorithm either returns true or it returns unknown but it will never return false. If it returns true it is guaranteed that $S, q \models \varphi$ for all $q \in init$. But if it returns unknown we do not know whether S satisfies φ or not.

This behaviour is due to the way model checking is done here: Several abstractions of S (generated out of the abstraction relations $\equiv_\psi$) are used each to model check a part of φ. And as usual with handcrafted abstractions there can be false negatives. The important point is that there are no false positives, i.e. if the abstractions fulfill φ then so does the concrete system.

Before we can present the algorithm we need the technical notion of a pseudo-MIS which will be used in it.

Definition 10. *A* pseudo-MIS *is a* MIS *together with a set Π of* global propositions *(which is disjoint to each set of local propositions) and a global labeling function $\pi : St \to \mathfrak{P}(\Pi)$. Note that every* MIS *can be viewed as a pseudo-*MIS *with $\Pi = \emptyset$.*

The algorithm now works as follows. Details about efficiently implementing some of the steps are given in the proof of Theorem 11.

Algorithm *modelcheck*$(S, init, \varphi, (\equiv_\psi)_{\psi \in \mathrm{qsf}(\varphi)})$:
Let $\varphi = \lambda(\theta_1, \ldots, \theta_n, \ell_1, \ldots, \ell_m)$ where
- λ is a monotone Boolean formula, i.e. λ is composed of conjunctions and disjunctions only,
- $\theta_1, \ldots, \theta_n$ are arbitrary ATL formulae each beginning with a quantifier or a negation directly followed by a quantifier, i.e. each θ_i is of the form $\langle\langle B\rangle\rangle\theta_i'$ or $\neg\langle\langle B\rangle\rangle\theta_i'$ (in the latter case we will still write $\equiv_{\theta_i}$ and $[\![\theta_i]\!]$ instead of $\equiv_{\langle\langle B\rangle\rangle\theta_i'}$ and $[\![\langle\langle B\rangle\rangle\theta_i']\!]$), and
- $\ell_1, \ldots, \ell_m$ are literals, i.e. atomic propositions or negations of atomic propositions.

1) For all $i \in \{1, \ldots, n\}$ do:
   i) Set $W_i := label(\theta_i, \equiv_{\theta_i})$.
   ii) Set $S := S(w_i, W_i)$, i.e. S is from now on viewed as a pseudo-MIS, a new global proposition w_i is introduced in S and it is labeled exactly in the states in W_i.
2) If $S, s \models \lambda(w_1, \ldots, w_n, \ell_1, \ldots, \ell_m)$ for all $s \in init$ then return true. Otherwise return unknown.
   Note that for this step the algorithm only has to locally check the labeling of the states $s \in init$ as $\lambda(w_1, \ldots, w_n, \ell_1, \ldots, \ell_m)$ is an entirely propositional formula.

Algorithm *label*$(\psi, \equiv)$:
Let $\psi = \neg^\psi\langle\langle A\rangle\rangle\mathbf{Y}\lambda(\theta_1, \ldots, \theta_n, \ell_1, \ldots, \ell_m)$ where
- $\neg^\psi$ is $\neg$ if ψ begins with a negation and it is the empty string elsewise,
- $\mathbf{Y} \in \{\mathbf{X}, \mathbf{G}, \mathbf{U}\}$,
- λ is a monotone Boolean formula,
- $\theta_1, \ldots, \theta_n$ are arbitrary ATL formulae each beginning with a quantifier or a negation directly followed by a quantifier, and
- $\ell_1, \ldots, \ell_m$ are literals.

1) Construct the abstraction

$$S' := \begin{cases} S^{[\![\psi]\!]}_{\equiv} & \text{if } \psi \text{ does not begin with a negation} \\ S^{\mathbb{A}\mathrm{gt}\setminus[\![\psi]\!]}_{\equiv} & \text{if } \psi \text{ does begin with a negation} \end{cases}$$

   We will view S' as a pseudo-MIS in the following steps.
2) For all $i \in \{1, \ldots, n\}$ do:
   i) Set $W_i := \{[q]_\equiv \mid \forall q' \in [q]_\equiv : q' \in label(\theta_i, \equiv_{\theta_i})\}$.
   ii) Set $S' := S'(w_i, W_i)$.
3) Compute the set W' of global states of S' (note that these are global states of the system abstracted with $\equiv$) satisfying ψ, i.e. $W' :=$

$$\{[q]_\equiv \mid S', [q]_\equiv \models \neg^\psi\langle\langle A\rangle\rangle\mathbf{Y}\lambda(w_1, \ldots, w_n, \ell_1, \ldots, \ell_m)\},$$

   by translating S' to a non-deterministic CGS and then using the ATL model checking algorithm from [1, Chapter 4.1]. As already pointed out in Section

3.1 their algorithm is only given for deterministic CGS but can be easily adapted to also handle non-deterministic systems.
There is, however, a caveat here. Because additional non-determinism might be introduced by abstracting the system we have to make sure that the non-determinism works "against the formula" because we want to avoid false positive outputs of our algorithm. This is the reason why we have to interpret the non-determinism as working *for* the agents in A if $\neg^\psi = \neg$ and working *against* them otherwise. If we always had it working against them (which seems natural as argued in Section 3.1) then in the former case it could happen that the algorithm comes to the conclusion that $S', [q]_\equiv \models \psi$ although $S, q \not\models \psi$ – a false positive. The reason for this would be non-determinism present in S' and absent in S that would presumably prevent agents A to have a winning strategy in S' although they do have one in S.

4) Return $W := \{q \in St \mid [q]_\equiv \in W'\}$.

6 Complexity and Soundness of the Algorithm

The following theorem shows that our model checking algorithm runs in time linear in the size of a succinct representation of the concrete system as well as linear in the length of the formula and exponential in the sum of the sizes of the abstract systems. Now, since there is a special abstraction for each modality, the abstractions should be very small and therefore this should be a huge improvement over the *EXPTIME*-completeness of model checking MIS without abstractions.

Theorem 11. *Algorithm modelcheck*$(S, init, \varphi, (\equiv_\psi)_{\psi \in \mathrm{qsf}(\varphi)})$ *runs in time*

$$O\left(|init| + |S| \cdot |\varphi|\right) \cdot 2^{O\left(\sum_{\psi \in \mathrm{qsf}(\varphi)} \left|S^{[\![\psi]\!]}_{\equiv_\psi}\right|\right)}$$

where $|S|$ *denotes the size of the* MIS S *in a compact representation. The cardinality of the global state space of* S *may then be upto* $2^{\Theta(|S|)}$.

For a proof see Theorem 13 in the appendix.

The following theorem shows that our algorithm is sound. It is, however, not complete because, as usual for abstraction techniques, the capability of the algorithm to show the truth of a formula depends on choosing a suitable abstraction. It should, anyhow, be possible to find good abstractions since it is possible to define a specific abstraction for each strategic operator. Of course that problem could be overcome by an automatic abstraction refinement technique but this, on the other hand, would make a provable upper bound on the runtime in the form of Theorem 11 impossible.

Theorem 12. *Algorithm modelcheck is sound, i.e. if modelcheck*$(S, init, \varphi, (\equiv_\psi)_{\psi \in \mathrm{qsf}(\varphi)})$ *outputs* `true` *then* $S, q \models \varphi$ *for all* $q \in init$.

For a proof see Theorem 14 in the appendix.

7 Communicating Agents Example

Consider the example of the six agents again (Example 2 and Example 6). We will now apply the model checking algorithm to the example using the formula

$$S, \mathrm{q} \models \langle\langle A \rangle\rangle (\top \mathbf{U} \mathsf{known}_{\mathsf{a}_4})$$

with $A = \{a_1, a_2, a_3, a_4\}$ and q is the global state in which only a_1 knows the message (cf. Example 6). The formula φ describes the following question: "Is it possible for team A to always ensure that a_4 will know the message eventually?" The algorithm takes the formula φ and constructs for all quantifier subformulae an abstract system by using the specific abstraction relation for that quantifier. The multiset $\mathrm{qsf}(\varphi)$ of quantified subformulae consists just of the formula φ. Therefore, we have to define only one abstraction for φ.

Before we give the abstraction relation we note that b_2 is not necessary for the property we want to verify and therefore we can temporarily delete it from the system. As abstraction for φ we do not abstract the agents a_i at all and for agent b_1 we use the equivalence relation given by the following partition of its local state space:

$$\begin{aligned} S_i &:= \{(R, N) \mid \emptyset \neq R \subseteq \{r_1, \ldots, r_i\}, n_i \in N\} \setminus \textstyle\bigcup_{j=1}^{i-1} S_j \quad \text{for } i = 1, \ldots, 3 \\ S_{\mathrm{rest}} &:= \{(R, N) \mid (R, N) \notin S_1 \cup \cdots \cup S_3\} \end{aligned}$$

Now, the agents a_i remain unchanged and the abstracted agent b_1 looks like the following:

$$b_1' = (St_{b_1}', d_{b_1}', out_{b_1}', in_{b_1}, o_{b_1}', \Pi_{b_1}, \pi_{b_1}')$$

where

- $St_{b_1}' = \{S_1, S_2, S_3, S_{\mathrm{rest}}\}$
- $\begin{aligned} \pi_{b_1}' : S_i &\mapsto \{\mathsf{known}_{\mathsf{b}_1}\} \quad \text{for } i = 1, \ldots, 3 \\ S_{\mathrm{rest}} &\mapsto \{\mathsf{known}_{\mathsf{b}_1}, \mathsf{unknown}_{\mathsf{b}_1}\} \end{aligned}$
- $\begin{aligned} d_{b_1}'(S_i) &= \{\mathrm{send}_{\mathrm{x}} \mid \mathrm{x} \in \{a_{i+1}, \ldots, a_4\}\} \quad \text{for } i = 1, \ldots, 3 \\ d_{b_1}'(S_{\mathrm{rest}}) &= \{\mathrm{send}_{\mathrm{x}} \mid \mathrm{x} \in \{a_1, \ldots, a_4\}\} \cup \{\mathrm{noop}\} \end{aligned}$
- $\begin{aligned} out_{b_1}' : (S_{\mathrm{rest}}, \mathrm{noop}) &\mapsto \{\mathbf{nothing}\} \\ (s, \mathrm{send}_{\mathrm{a}_1}) &\mapsto \{\mathbf{m_{b_1 a_1}}\} \\ (s, \mathrm{send}_{\mathrm{a}_2}) &\mapsto \{\mathbf{m_{b_1 a_2}}\} \\ (s, \mathrm{send}_{\mathrm{a}_3}) &\mapsto \{\mathbf{m_{b_1 a_3}}\} \\ (s, \mathrm{send}_{\mathrm{a}_4}) &\mapsto \{\mathbf{m_{b_1 a_4}}\} \end{aligned}$
  for all $s \in St_{b_1}'$,
- $\begin{aligned} o_{b_1}' : (S_{\mathrm{rest}}, \alpha, \mathbf{nothing}) &\mapsto \{S_{\mathrm{rest}}\} \\ (S_{\mathrm{rest}}, \alpha, \{\mathbf{m_{a_j b_1}}\}) &\mapsto \{S_{\mathrm{rest}}, S_j\} \\ (S_i, \mathrm{send}_{\mathrm{a_j}}, \mathbf{nothing}) &\mapsto \{S_{\mathrm{rest}}\} \\ (S_i, \mathrm{send}_{\mathrm{a_j}}, \{\mathbf{m_{a_{j'} b_1}}\}) &\mapsto \{S_{j'}\} \\ (S_i, \mathrm{send}_{\mathrm{a_j}}, \{\mathbf{m_{a_4 b_1}}\}) &\mapsto \{S_{\mathrm{rest}}\} \end{aligned}$
  for all $\alpha \in Act,\ \gamma \in \mathcal{I}n,\ i, j' \in \{1, 2, 3\}$ and $j \in \{1, 2, 3, 4\}$.

Now, everything is specified so that the algorithm $\mathit{modelcheck}(S, \mathit{init}, \varphi, (\equiv_\psi)_{\psi \in \mathrm{qsf}(\varphi)})$ can be started. S is the MIS mentioned in Section 2, $\mathit{init} = \{q\}$, φ as above and $(\equiv_\psi)_{\psi \in \mathrm{qsf}(\varphi)}$ contains all abstraction relations for the quantified subformulae. Since the formula φ only consists of one quantifier, by invoking the algorithm *modelcheck*() we get the quantifier subformulae $\theta_1 := \varphi$. The algorithm takes then φ and computes W_1 by executing the labeling algorithm $\mathit{label}(\varphi, \equiv_\varphi)$. Now, we construct the pseudo-MIS $S'_\varphi := S^{[\![\varphi]\!]}_{\equiv_\varphi}$ for the favored agents a_1, a_2, a_3, a_4. Step 2i) of the labeling algorithm is skipped since there is no further quantified subformula for φ. This is the moment when the recursion stops and we start to label the states in a bottom-up order. $W_1 := \{[q]_{\equiv_\varphi} \mid S'_\varphi, [q]_{\equiv_\varphi} \models \varphi\}$ is computed by creating the non-deterministic CGS and apply the model checking algorithm for ATL.

Now we are almost finished. In the *modelcheck*() algorithm we set $S' := S'(w_1, W_1)$. The last step is to evaluate whether $S, s \models \varphi$ holds and we therefore answer with true.

8 Conclusion and Future Work

While in the MAS community model checking agent systems already has attracted some attention there has not been much work on abstraction techniques for reducing the state space. In this paper, we presented a technique to cope with the state explosion problem which opens the way to reduce the state space of a MAS so that model checking might get tractable. Clearly, there cannot be a generic automatizable abstraction technique: Model checking ATL for MIS is *EXPTIME*-complete, therefore in the worst case, there are instances where no abstraction technique at all is applicable.

Consequently we focused on handcrafted abstraction relations and proved that the presented model checking algorithm is sound, i.e., if the algorithm claims that a property holds then it really does. Of course, using abstraction always leads to losing completeness. However, abstraction still has its benefits because without reducing the state space many problems could not be model checked at all. Defining different abstraction relations for each quantifier allows to shrink the state space as needed for each subformula. Usually a MAS consists of more than two teams and the agents are more complex than in our example which increases our speedup factor significantly. In fact, we believe that most real problems carry with them a rich structure which allows the abstraction technique to be successfully applied, especially when using the possibility to use more than one abstraction for a single formula.

We decided to take MIS as the modelling framework and argued that for any framework the modularity is important not only because of the nature of MAS but also due to the ability of reducing the state space by replacing or removing agents that are not necessary when checking a certain property. We therefore introduced a modified version of a MIS and defined an abstraction over it.

The need to have a compact, modular and ground representation was motivated by the idea of an IT ecosystem, i.e., a system composed of a large number

of distributed, decentralized, autonomous, interacting, cooperating, organically grown, heterogeneous, and continually evolving subsystems. An example for such a system is a smart city that contains agents for cars, traffic lights, cameras, etc. In such an IT ecosystem, new agents are introduced, other agents are removed and others again are modified. If we nevertheless want to ensure some safety, liveness or fairness properties we need a framework that on the one hand enables theoretical analysis and on the other hand supports modularity.

An IT ecosystem in general is the topic of a large research project consisting of 17 professors and 33 scientists in total collecting knowledge in different research areas: multi-agent systems, organic computing, ambient intelligence, software engineering and embedded systems. Together we try to solve the contradiction of having a continually evolving and highly heterogeneous system on the one side and still controlling this system by ensuring some properties on the other side.

We are currently implementing our abstraction technique in a first prototype and will use it for a concrete, non-trivial demonstrator scenario [9]. The application will run on a smartphone and will send properties to be checked to a server that will then model check it. Users will get feedback if the formula holds or if it is unknown and can then decide whether they want to take part in the IT ecosystem. For this implementation it will, of course, be very useful to develop some heuristics and automatic refinement methods to generate abstractions.

For the future, we plan to put some effort in developing parallel model checking methods for this system and using a logic that facilitates the use of probabilities.

9 Acknowledgments

This work was funded by the NTH Focused Research School for IT Ecosystems. NTH (Niedersächsische Technische Hochschule) is a joint university consisting of Technische Universität Braunschweig, Technische Universität Clausthal, and Leibniz Universität Hannover.

References

1. Alur, R., Henzinger, T.A., Kupferman, O.: Alternating-time temporal logic. J. ACM 49(5), 672–713 (2002)
2. Ball, T., Rajamani, S.: Boolean programs: A model and process for software analysis. Tech. Rep. 2010-14 (2000)
3. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. J. ACM 50(5), 752–794 (2003)
4. Clarke, E.M., Grumberg, O., Long, D.E.: Model checking and abstraction. ACM Trans. Program. Lang. Syst. 16(5), 1512–1542 (1994)
5. Clarke, E., Grumberg, O., Peled, D.: Model checking. Springer (1999)
6. Cohen, M., Dam, M., Lomuscio, A., Russo, F.: Abstraction in model checking multi-agent systems. In: AAMAS (2). pp. 945–952 (2009)

7. Das, S., Dill, D.: Successive approximation of abstract transition relations. In: Logic in Computer Science, 2001. Proceedings. 16th Annual IEEE Symposium on. pp. 51–58. IEEE (2002)
8. Dechesne, F., Orzan, S., Wang, Y.: Refinement of kripke models for dynamics. In: Fitzgerald, J., Haxthausen, A., Yenigun, H. (eds.) Theoretical Aspects of Computing - ICTAC 2008, Lecture Notes in Computer Science, vol. 5160, pp. 111–125. Springer Berlin / Heidelberg (2008), http://dx.doi.org/10.1007/978-3-540-85762-4_8
9. Deiters, C., Köster, M., Lange, S., Lützel, S., Mokbel, B., Mumme, C., (eds.), D.N.: DemSy - A Scenario for an Integrated Demonstrator in a SmartCity. Tech. Rep. 2010/01, NTH Focused Research School for IT Ecosystems, Clausthal University of Technology (May 2010), http://www.gbv.de/dms/clausthal/H_BIB/IfI/NTH_CompSciRep/2010-01.pdf
10. Emerson, E., Halpern, J.: "Sometimes" and "not never" revisited: on branching versus linear time temporal logic. Journal of the ACM (JACM) 33(1), 151–178 (1986)
11. Enea, C., Dima, C.: Abstractions of multi-agent systems. In: Burkhard, H.D., Lindemann, G., Verbrugge, R., Varga, L. (eds.) Multi-Agent Systems and Applications V, Lecture Notes in Computer Science, vol. 4696, pp. 11–21. Springer Berlin / Heidelberg (2007), http://dx.doi.org/10.1007/978-3-540-75254-7_2
12. Halpern, J., Fagin, R.: Modelling knowledge and action in distributed systems. Distributed computing 3(4), 159–177 (1989)
13. Halpern, J., Fagin, R., Moses, Y., Vardi, M.: Reasoning about knowledge. Handbook of Logic in Artificial Intelligence and Logic Programming 4 (1995)
14. Henzinger, T.A., Jhala, R., Majumdar, R.: Counterexample-guided control. In: ICALP. Lecture Notes in Computer Science, vol. 2719, pp. 886–902. Springer-Verlag (2003)
15. Henzinger, T.A., Majumdar, R., Mang, F.Y.C., Raskin, J.F.: Abstract interpretation of game properties. In: Proceedings of the 7th International Symposium on Static Analysis. pp. 220–239. SAS '00, Springer-Verlag, London, UK (2000), http://portal.acm.org/citation.cfm?id=647169.718154
16. Jamroga, W., Ågotnes, T.: Modular interpreted systems: A preliminary report. Tech. Rep. IfI-06-15, Clausthal University of Technology (2006)
17. Jamroga, W., Ågotnes, T.: Modular interpreted systems. In: Durfee, E.H., Yokoo, M., Huhns, M.N., Shehory, O. (eds.) AAMAS. p. 131. IFAAMAS (2007)
18. Kupferman, O., Vardi, M.Y.: An automata-theoretic approach to modular model checking. ACM Trans. Program. Lang. Syst. 22, 87–128 (January 2000), http://doi.acm.org/10.1145/345099.345104
19. Kurshan, R.: Computer-aided verification of coordinating processes: the automata-theoretic approach. Princeton Univ Press (1994)
20. McMillan, K.: Applying sat methods in unbounded symbolic model checking. In: Brinksma, E., Larsen, K. (eds.) Computer Aided Verification, Lecture Notes in Computer Science, vol. 2404, pp. 303–323. Springer Berlin / Heidelberg (2002), http://dx.doi.org/10.1007/3-540-45657-0_19
21. McMillan, K.: Symbolic model checking: An Approach to the State Explosion Problem. Kluwer Academic Publishers Norwell, MA, USA (1993)
22. Wooldridge, M.: Computationally grounded theories of agency. Multi-Agent Systems, International Conference on 0, 0013 (2000)

10 Appendix

10.1 Figures

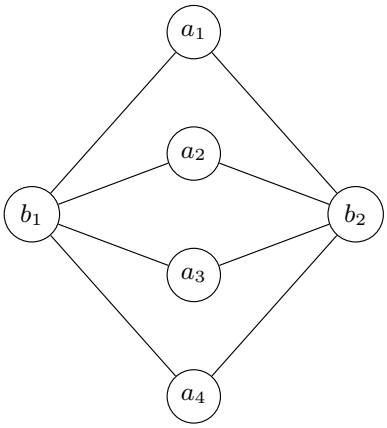


Fig. 1. Communication graph of the six agents

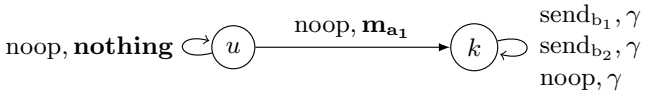


Fig. 2. Graph of agent a_1

10.2 Proofs

Theorem 13. *Algorithm modelcheck*$(S, init, \varphi, (\equiv_\psi)_{\psi \in \text{qsf}(\varphi)})$ *runs in time*

$$O\left(|init| + |S| \cdot |\varphi|\right) \cdot 2^{O\left(\sum_{\psi \in \text{qsf}(\varphi)} \left| S_{\equiv_\psi}^{[\![\psi]\!]} \right|\right)}$$

where $|S|$ *denotes the size of the* MIS S *in a compact representation. The cardinality of the global state space of* S *may then be upto* $2^{\Theta(|S|)}$.

Proof. The crucial implementation detail is that it is not possible to explicitly enumerate the set returned in step 4 of *label*() because the set may be as large as the global state space St of S. Instead, both *label*() and *modelcheck*() have to save and pass on computed sets of global states in a symbolic way, i.e. by refering to the modular structure of S and to the abstraction relation's equivalence

classes. This technique is needed in step 1 of *modelcheck*() and in steps 2i and 4 of *label*().

Also, in step 2i of *label*() it is not possible to check the condition for each $[q]_{\equiv} \in St'$ by enumerating through all $q' \in [q]_{\equiv}$ because a single equivalence class may already be as large as the global state space. Hence, the algorithm has to construct the equivalence relation $\equiv' := \equiv \cap \equiv_{\theta_i}$ (which is a refinement of both $\equiv$ and $\equiv_{\theta_i}$) and compute W_i as the set $\{[q]_{\equiv} \mid \forall [q']_{\equiv'} \subseteq [q]_{\equiv} : [q']_{\equiv'} \subseteq \mathit{label}(\theta_i, \equiv_{\theta_i})\}$. This can be done in time $O(|S_{\equiv}^{[\![\psi]\!]}| \cdot |S_{\equiv_{\theta_i}}^{[\![\theta_i]\!]}|)$.

Step 3 of *label*() runs in time $2^{O\left(|S_{\equiv}^{[\![\psi]\!]}|\right)} \cdot O(|\psi|)$ because the translation to a CGS may involve an exponential blow-up in the system size. All other steps are easy to implement – when keeping in mind the symbolic handling of state sets.

Finally, *label*() is executed at most $|\varphi|$ times. Altogether this gives the claimed upper bound on the runtime.

Theorem 14. *Algorithm modelcheck is sound, i.e. if* $\mathit{modelcheck}(S, \mathit{init}, \varphi, (\equiv_\psi)_{\psi \in \mathrm{qsf}(\varphi)})$ *outputs* true *then* $S, q \models \varphi$ *for all* $q \in \mathit{init}$.

Proof. (Sketch) First note that if we skipped step 1 of the *label*() algorithm and simply ran the algorithm without constructing any abstractions we would exactly run the bottom-up, subformula labeling, model checking algorithm from [1, Chapter 4.1].

Hence we only have to argue why the abstractions do not lead the algorithm to produce more positive answers than without them. The crucial observation is that for each modality $\langle\langle A \rangle\rangle \mathbf{Y}$ the aspects which are of an existentially quantifying nature, i.e. the actions available to agents A, can only be restricted by an abstraction but never extended and for the aspects of universally quantifying nature, i.e. the actions available to agents $\mathbb{A}\mathrm{gt} \setminus A$ as well as the non-deterministic branching of the system, it is the other way around. Thus it is ensured that if a formula $\langle\langle A \rangle\rangle \mathbf{Y} \varphi$ is true in an abstraction it is also true in the original system.

Furthermore, for formulae of the form $\neg\langle\langle A \rangle\rangle \mathbf{Y} \varphi$ the abstraction is constructed the other way around, i.e. extended choices for A and restricted choices for $\mathbb{A}\mathrm{gt} \setminus A$, to ensure that if the agents A do not have a winning strategy in the abstraction then neither do they have one in the original system.

The non-determinism, however, is extended even in this case. As already discussed in step 3 of the algorithm we therefore have to change the meaning of the non-determinism to be of existential rather than of universal nature. The sacrifice we have to make is that if the original system is already non-deterministic and this non-determinism ensures some property $\neg\langle\langle A \rangle\rangle \mathbf{Y} \varphi$ then the algorithm will return unknown.

MAS: Qualitative and Quantitative Reasoning

Ammar Mohammed and Ulrich Furbach

Universität Koblenz-Landau, Artificial Intelligence Research Group,
D-56070 Koblenz, {ammar,uli}@uni-koblenz.de

Abstract. In a former work, we have presented/implemented a framework for modeling and verifying multi-agent systems, using hybrid automata. To specify properties of those systems, one needs a specification language that brings, at the same level of specification, both the qualitative and quantitative requirements. For this aim, there have been proposed several temporal logics with either event or state based approach. Both approaches have their pros and cons which should not be played off against each other. This paper contributes to present a variant of temporal logics which combines the expressiveness of both approaches. Using this proposed logic, we are able reason about many properties in a concise and intuitive manner. In particular, we concentrate on those types of properties that can be verified using reachability analysis. Hence these properties can be verified directly within our implemented framework.

1 Introduction

A great deal of research has made the concept of Multi-Agent Systems (MAS) more precise by means of logical systems, particularly modal logics [33], which has contributed to develop several programming and verification tools to reason about MAS. Temporal logic, LTL or CTL [30, 9], is as a subclass of model logics which is able to reason about the evolving of systems in time. An important advantage of the use of temporal logics is that they verify systems by means of model checking. The latter is one of the approaches that is recently used in automated planning [28, 16]. Several work has integrated temporal logic on the top of certain modal logic to be able to reason about actions with temporal properties (cf. [22, 7]).

Temporal logics basically allow us to express and reason about those qualitative properties of systems which focus on the temporal order of the occurrence of events. An example of these properties is to specify that a certain property of interest may eventually occur, or in other words the formula is reached in the model. Temporal logics, however, are insufficient to specify those quantitative temporal requirements which put timing deadlines on the behaviors of specified systems. For example, temporal logic can specify that the $action_1$ is always followed by $action_2$, but it can not reveal how long the period between the two action takes place. Because of their inability to specify such quantitative properties, temporal logics have to incorporate explicitly the notation of time. For this aim, there have been proposed several extensions to temporal logics that

bind the notation of time to formulas (see [4, 8] for a survey). The underlying models of these logics are represented as state transition graphs annotated with time constraints, using either *event* or *state* based approach. The former approach uses the discrete time model of the occurrence of events to reason about systems, while the latter approach uses continuous time model that records the state changes of systems at each point of time. Each approach has advantages over the other. The main advantage of event based approach together with its underlying discrete time model, is its simplicity to express quantitative properties by abstracting lots of details within a model of a system. Intuitively most of the quantitative requirements often occur at the discrete changes of the behavior of systems and hence these requirements can reason about agents carrying out actions in time. This approach, however, can not reason about quantitative properties, which may occur within a particular time interval. For example, it might be desirable to reason about the satisfaction of a certain property of interest within an interval of time, say $10 \leq t \leq 20$. This can not be expressed with events unless the time interval coincides with events on the boundaries of the interval. This limitation can be coped with using the expressive power of state based approach. The latter approach, however, can not directly reason about events, which are used to reason about actions of MAS. Converting from the state based to the event based representation often leads to a significant enlargement of the state space. To specify and hence verify a property based on the occurrence of events, it should be converted into an appropriate state base representation before it is checked by state based quantitative temporal logics tools [34, 10]. For example, to specify and verify that it is always the case that $event_1$ is followed by $event_2$ within t time unit— this property is called a bounded response property— a traditional solution to verify this within a model M of a multi-agent system is to translate this specification to a testing transition model A, and then check whether the parallel model of A and M can reach a designated state of A.

Usually any quantitative temporal logic needs an interpretation models of the form labeled timed graph or even more general structure like timed automata [2]. A general model of timed automata is hybrid automata [18] in which one can reason not only about quantitative time requirements, but also about quantitative behaviors of systems raised from evolution of continuous actions.

The main contribution of this paper is to propose a novel variant of CTL called Region Computation Tree Logic(RCTL) that extends CTL by incorporating time on states and events in order to reason about both qualitative and quantitative requirements of systems particularly MAS. RCTL encompasses, in the same framework, the expressive power of event and state based approach.The formulas of RCTL are interpreted on tree of regions generated from the transition system of hybrid automata presented in a former work [27]. For the purpose of model checking, we concentrate on those quantitative properties that can be verified using reachability analysis. Hence, we will be able to use a former constraint logic programming approach presented in [27, 32]. In this approach, a model of MAS described as hybrid automata is converted to an equivalent model

of constraint logic program. Then requirements are converted to suitable queries that are checked within the constraint logic program.

The rest of this paper is organized as follows. Sec. 2 introduces hybrid automata, which constitute the interpretation model of RCTL. Then syntax and semantics of RCTL are discussed in Sec. 3. Specifications of properties that can be verified by means of reachability analysis are discussed in Sec. 4, before we end up with the conclusion and related work in Sec. 5.

2 Hybrid Automata:Perliminaries

In this section we briefly review our labeled hybrid automata, in which we admit the existance of events on transitions (see [27] for details with an example). But first we need to define those constraints which may appear as guards on transitions and invariants of hybrid automata. Then, we define the constraints that define the possible dynamics in our model.

Definition 1 (Linear Constraints) *Let $\mathbb{X}$ be set of n real variables and $\omega = \sum_{i=1}^{n} a_i \cdot x_i$, with $x_i \in \mathbb{X}$, be a linear combination of variables from $\mathbb{X}$, where $1 \leq i \leq n, a_i \in \mathbb{R}$. A set $\Phi(\mathbb{X})$ of linear constraints over $\mathbb{X}$, with a typical elements φ, is defined by the following syntax:*

$$\varphi ::= \ \omega \sim b \mid \varphi_1 \wedge \varphi_2 \mid true$$

where $b \in \mathbb{R}$, $\sim \in \{<, \leq, =, >, \geq\}$, and φ_1, $\varphi_2 \in \Phi(\mathbb{X})$.

The continuous behaviors of MAS show how physical quantities, e.g. position, temperature and humidity, evolve with respect to time. Those behaviors are usually described by differential equations whose solutions can be described as continuous functions in time. In the following, we define the basic constraints that constitute the continuous dynamics of the variables.

Definition 2 (Dynamical Constraints) *Let $\mathbb{X}$ be a set of n real variables, with a typical element $x \in \mathbb{X}$, and $\dot{\mathbb{X}}$ be set of first derivatives of the variables of $\mathbb{X}$ with a typical element $\dot{x} \in \dot{\mathbb{X}}$. A set $\mathbb{D}(\mathbb{X} \cup \dot{\mathbb{X}})$ of dynamical constraints over $\mathbb{X} \cup \dot{\mathbb{X}}$ with typical element d, is defined inductively by the following syntax:*

$$d ::= \ \dot{x} \sim c \mid \dot{x} + a \cdot x = c \mid d_1 \wedge d_2 \mid true$$

where $a \neq 0, c \in \mathbb{R}$, $\sim \in \{=, \leq, \geq\}$, d_1, $d_2 \in \mathbb{D}(\mathbb{X} \cup \dot{\mathbb{X}})$.

Having defined the linear and dynamical constraints, we are ready to introduce a hybrid automaton[1].

[1] Each automaton represents an agent

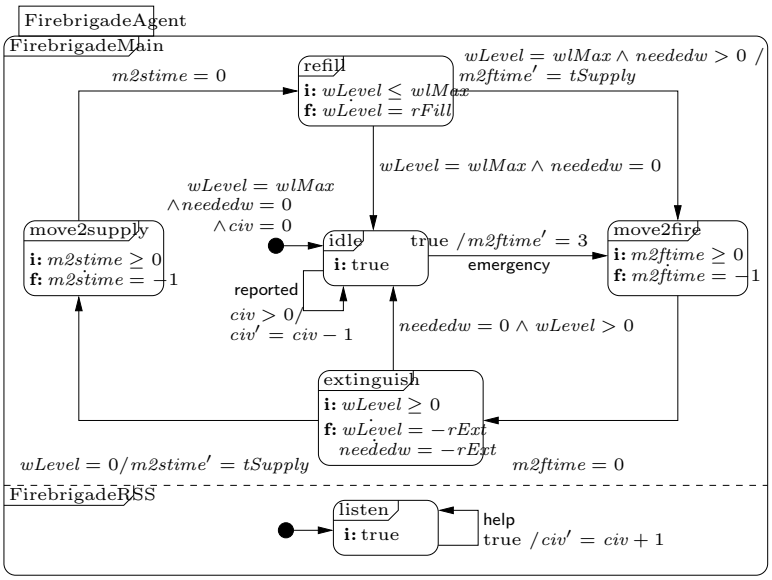


Fig. 1. A simple fire brigade agent [15].

2.1 Syntax

Definition 3 (basic components) *A hybrid automaton is a tuple* $H = (Q, \mathbb{X}, Inv, Flow, E, Jump, Reset, Event, Event_H, q_0, v_0)$ *where:*

- Q *is a finite set of control locations.*
- $\mathbb{X}$ *is an ordered set of* n *real variables.*
- $Inv : Q \rightarrow \Phi(\mathbb{X})$ *is a function that assigns a linear constraint* $Inv(q)$ *to each location* $q \in Q$*.*
- $Flow : Q \rightarrow \mathbb{D}(\mathbb{X} \cup \dot{\mathbb{X}})$ *is a function that assigns a dynamical constraints* $Flow(q)$ *to each control location* $q \in Q$*.*
- $E \subseteq Q \times Q$ *is a finite set of transitions among control locations.*
- $Jump : E \rightarrow \Phi(\mathbb{X})$ *is a function that assigns to each transition* $e \in E$ *a constraints* $jump(e)$*, which must hold to fire* e*.*
- $Reset : E \times \mathbb{X} \rightarrow \mathbb{R}$ *is a mapping, which assigns a real value to each variable on each transition* $e \in E$*. A reset of a variable* $x \in \mathbb{X}$ *on a transition* $e \in E$ *is denoted as* $x' = Reset(e, x)$*. Conveniently, we write* $Reset(e, \mathbb{X})$ *to denote the reset of all variables.*
- $Event_H$ *is a finite set of events.*
- $Event : E \rightarrow Event_H$ *is a function that assigns an event to each transition* $e \in E$ *from a set of events* $Event_H$*.*
- $q_0 \in Q$ *defines the initial location of the automaton.*
- v_0 *defines the initial values of the variables* $\mathbb{X}$*.*

An example of a MAS described as hybrid automata is shown in Fig. 1, which describes a fire brigade agents in a Robobcup rescue scenario (cf. [15] for details)

2.2 Semantics

A hybrid automaton can exactly be in one of its control locations at each stage of its computation. But knowing the present control location is not enough to determine which of the outgoing transitions can be taken next, at all. A snapshot of the current state of the computation should also keep in mind the present valuation of the continuous variables. To begin formalizing the semantics of a hybrid automaton, we need to define the concept of a *state* and to show how control evolves from one state to another. But first we need to define how continuous variables evolve.

Definition 4 (Evaluation of Linear Constraints) *Let $\varphi \in \phi(\mathbb{X})$ be a constraints and $v \in \mathbb{R}^n$ be the valuation of the variables $\mathbb{X}$, then we write*

$$v \models \varphi,$$

if v satisfies the constraint φ, which is defined inductively as

$$\begin{array}{lll} \varphi = true. & & \\ \varphi = \sum_{i=1}^{n} a_i \cdot x_i \sim c & \textit{iff} & \sum_{i=1}^{n} a_i \cdot v_i \sim c \ \textit{holds.} \\ \varphi_1 \wedge \varphi_2 & \textit{iff} & v \models \varphi_1 \textit{ and } v \models \varphi_2. \end{array}$$

where v_i is the valuation of the ith components of v

Definition 5 (Evaluation of Dynamical Constraints) *Let $d \in \mathbb{D}(\mathbb{X} \cup \dot{\mathbb{X}})$ be a dynamical constriants and $f : \mathbb{R}^{\geq 0} \to \mathbb{R}^n$ be a differentiable function, then we write*

$$f \vDash_* d$$

if f satisfies the dynamical constraint d, which is defined inductively as

$$\begin{array}{lll} d = true. & & \\ d = \dot{x} \sim c & \textit{iff} & f'(t) \sim c \ \textit{holds.} \\ d = \dot{x} + a \cdot x = c & \textit{iff} & f'(t) + a \cdot f(t) \sim c \ \textit{holds.} \\ d = d_1 \wedge d_2 & \textit{iff} & f \vDash_* d_1 \textit{ and } f \vDash_* d_2. \end{array}$$

where $f'(t)$ is the differentiation of the function f for $t \in \mathbb{R}^{\geq 0}$.

Definition 6 (State) *At any instant of time $t \in \mathbb{R}^{\geq 0}$, a state of a hybrid automaton is given by $\sigma_i = \langle q_i, v, t \rangle$, where $q_i \in Q$ is a control location, v is the valuation of the real variables. A state $\sigma_i = \langle q_i, v, t \rangle$ is admissible iff $v \models Inv(q_i)$.*

The semantics of a hybrid automaton is defined in terms of a labeled transition system between states. Transitions between states are generally categorized into two kinds of transitions: continuous transitions, capturing the continuous evolution of states, and discrete transitions, capturing the changes of location. We will define the semantics of hybrid automaton more formally.

Definition 7 (Operational Semantics) *A transition rule between two admissible states $\sigma_1 = \langle q_1, v_1, t_1 \rangle$ and $\sigma_2 = \langle q_2, v_2, t_2 \rangle$ is*

Discrete transition *iff* $e = (q_1, q_2) \in E$, $t_1 = t_2$ *and* $v_1 \models Jump(e)$, *and* $v_2 \models Inv(q_2)$, *such that* v_2 *is the valuations coming from* $Reset(e, \mathbb{X})$. *In this case an event* $a \in Event_H$ *occurs. Conventionally, we write this as* $\sigma_1 \xrightarrow[t_1]{a} \sigma_2$.

Continuous(Delay) transition *iff* $q_1 = q_2$, $(t_2 - t_1) > 0$ *is the duration of time passed at location* q_1, *there exists a differentiable function* f *with* $f \vDash_* Flow(q_1)$ *and* $f(t_1) = v_1$ *and* $f(t_2) = v_2$, *and for all* $t \in [t_1, t_2]$, $f(t) \models Inv(q1)$.

An execution of a hybrid automaton corresponds to a sequence of transitions from one state to another. For this purpose, we define the valid run as follows:

Definition 8 (Run: micro level) *A path* $\rho = \sigma_1\sigma_2\sigma_3, \ldots,$ *of a hybrid automaton* H *is a finite or infinite sequence of admissible states, where the transition from a state* σ_i *to a state* σ_{i+1}, *for all* $i \geq 1$, *is related either by a discrete or continuous transition. A set of all possible paths of* A *is denoted as* $\Pi(H)$. *A run of* H *is a path* ρ *starting with the initial state* σ_0.

It should be noted that the continuous change of states in a path ρ generates an infinite number of reachable states. Therefore, state-space exploration techniques require a symbolic representation way for representing these infinite states appropriately. One way to do so is to use mathematical intervals. We call this symbolic mathematical interval *region*, which is defined as follows:

Definition 9 (Region) *Given a path* $\rho \in \Pi(H)$, *a sub-sequence of admissible states* $\Gamma = (\sigma_{i+1} \cdots \sigma_{i+m}) \subseteq \rho$ *is called a region, if for all states* σ_{i+j} *with* $1 \leq j \leq m$, *it holds* $q_{i+j} = q$ *and for the states* σ_i *and* σ_{i+m+1} *with respective locations* q_i *and* q_{i+m+1}, *then it must hold* $q_i \neq q$ *and* $q_{i+m+1} \neq q$. *Conventionally, a region* Γ *is written as* $\Gamma = \langle q, V, T \rangle$, *where* $t_{i+1} \leq T \leq t_{i+m}$ *is the interval of continuous time, and* V *is the tuple of intervals valuations of the variables during the time interval* T.

A run of a hybrid automaton can be re-phrased in terms of reached regions, where the change from one region to another is fired by using a discrete step.

Definition 10 (Run: macro level) *A run of hybrid automaton* H *is* $\rho_H = \Gamma_0, a_1, \Gamma_1, a_2, ...,$ *a sequence of (possibly infinite) regions, where a transition from a region* Γ_i *to a region* Γ_{i+1}, *written as* $\Gamma_i \xrightarrow[t_{i+1}]{a_{i+1}} \Gamma_{i+1}$, *is enabled, if there is* $\sigma_i \xrightarrow[t_{i+1}]{a_{i+1}} \sigma_{i+1}$, *where* $\sigma_i \in \Gamma_i$, $\sigma_{i+1} \in \Gamma_{i+1}$ *and* $a_{i+1} \in Event$ *is the generated event before the control goes to the region* Γ_{i+1}. Γ_0 *is the initial region obtained from a start state* σ_0 *by means of continuous transitions.*

The operational semantics are the basis for verification of a hybrid automaton. In particular, model checking of a hybrid automaton is defined as the reachability analysis of its underlying transition system. The most useful question to ask about hybrid automata is the reachability of a given state. We define the reachability of a region and state as follows.

Definition 11 (Reachability) *A region Γ_i is called reachable in a run ρ_H, if $\Gamma_i \in \rho_H$. Consequently, a state σ_j is called reachable, if there is a reached region Γ_i such that $\sigma_j \in \Gamma_i$*

2.3 Parallel Composition

The parallel composition of hybrid automata can be used to specify larger systems (MAS), where a hybrid automaton is given for each part of the system and communication between the different parts may occur via shared variables and synchronization labels. The transitions from the different automata are interleaved, unless they share the same synchronization label. In this case, they are synchronized on transitions. As a result of the parallel composition, a new automaton called composed automaton is created which captures the behavior of the entire system. The composed automaton is, in turn, given to a model checker that checks the reachability of a certain state. In [27], we showed how to construct the composition *on-the-fly*— i.e., during the verification phase—, in which the composition of hybrid automata H_1 and H_2 can be defined in terms of synchronized or interleaved regions of the regions produced from run of both H_1 and H_2. As a result of the composition procedure, compound regions are constructed, which consist of a conjunction of a region $\Gamma_1 = \langle q_1, V_1, T \rangle$ from H_1 and another region $\Gamma_2 = \langle q_2, V_2, T \rangle$ from H_2. Therefore, each compound region takes the form $\Lambda = \langle (q_1, V_1), (q_2, V_2), T \rangle$ (shortly written as $\Lambda = \langle \Gamma_1, \Gamma_2, T \rangle$), which represents the reached region at both control locations q_1 and q_2 the during a time interval T.

Definition 12 (Composed Run) *A run of composed automata is the sequence $\sum_{H_1 \circ H_2} = \Lambda_0, a_1, \Lambda_1, a_2, ...$ of compound regions, where a transition between compound regions $\Lambda_1 = \langle \Gamma_1, \gamma_1, T_1 \rangle$ and $\Lambda_2 = \langle \Gamma_2, \gamma_2, T_2 \rangle$ (written as $\Lambda_1 \xrightarrow[t]{a} \Lambda_2$) is enabled, if one of the following holds:*

- *$a \in Event_{H_1} \cap Event_{H_2}$ is a joint event, $\Gamma_1 \xrightarrow[t]{a} \Gamma_2$, and $\gamma_1 \xrightarrow[t]{a} \gamma_2$. In this case , we say that the region Γ_1 is synchronized with the region γ_1.*
- *$a \in Event_{H_1} \setminus Event_{H_2}$ (respectively $a \in Event_{H_2} \setminus Event_{H_1}$), $\Gamma_1 \xrightarrow[t]{a} \Gamma_2$ and $\gamma_1 \rightarrow \gamma_2$, such that both γ_1 and γ_2 have the same control location—i.e. they relate to each other using a continuous transition.*

3 Region Computation Tree Logic (RCTL)

This section primarily focuses on the definition of the region computation tree logic (RCTL), which extends the qualitative temporal logic of CTL with time on states, events, and constraints of variables. RCTL combines, in the same level of specifications, qualitative together with quantitative requirements. The formulas of RCTL are interpreted over the possible regions obtained from the run of hybrid automata. As described previously, a region can be seen as a sequence of states separated by transition points. Each transition point marks

the instantaneous exit from region Γ_{i-1} and the entrance into region Γ_i, and corresponds to the occurrence of a particular event. Therefore, we see regions constituting the essence of RCTL, such that RCTL can be viewed as a state based quantitative temporal logics in a sense that regions capture the changes of states, and as event based quantitative temporal logics in a sense that events mark the instantaneous exist from region to another. Thus, RCTL brings together, in the same framework, the advantages of both approaches. In the following we show the syntax and semantics of RCTL, but first we define timed variables and its valuation function.

Definition 13 (Timed-variables) *Let $\mathbb{T}$ be a set of non-negative real variables called* timed-variables*, and $\Phi(\mathbb{T})$ be a set of linear constraints over $\mathbb{T}$. The valuation ξ of the timed-variables $\mathbb{T}$ is a function $\xi : \mathbb{T} \rightarrow \mathbb{R}^{\geq 0}$. Given $\pi \in \Phi(\mathbb{T})$, we write $\xi \models \pi$, if ξ satisfies the constraint π.*

3.1 Syntax of RCTL

Let $\mathbb{X}$ be a set of real variables, $\mathbb{T}$ be a set of non-negative real variables disjoint from $\mathbb{X}$, $\Phi(\mathbb{X})$ and $\Phi(\mathbb{T})$ be two sets of linear constraints with free variables from $\mathbb{X}$ and $\mathbb{T}$ respectively, L be a set of atomic propositions denoting the locations, and *Event* be a set of atomic propositions denoting events disjoint from L.

Definition 14 (Formulas of RCTL) *The formula Ψ of RCTL are inductively defined as*

$$\Psi ::= p \mid a \mid \phi \mid y.\Psi \mid \pi \mid \neg\Psi \mid \Psi_1 \wedge \Psi_2 \mid \exists(\Psi_1 U \Psi_2) \mid \forall(\Psi_1 U \Psi_2)$$

for $y \in \mathbb{T}$, $p \in L$, $a \in Event$, $\phi \in \Phi(\mathbb{X})$, $\pi \in \Phi(\mathbb{T})$, and Ψ_1, Ψ_2 are RCTL formulas.

Before giving the semantics of RCTL, we introduce some common notations. $\exists\Diamond\Psi$ is equivalent to $\exists(true\ U\Psi)$, $\forall\Diamond\Psi$ is eqivalent to $\forall(true\ U\Psi)$, $\exists\Diamond\Psi$ is equivalent to $\exists(true\, U\Psi)$, and $\forall\Diamond\Psi$ is equivalent to $\forall(true\, U\Psi)$

3.2 Semantics of RCTL

We will interpret the formulas of RCTL over the set of all possible regions generated from possible runs of hybrid automata. Let a region Γ take the form $\Gamma = (q, V, T)$, with $\delta(\Gamma) = q$ is its location, and V and T are the interval of valuations and time respectively, in which the region is admissible. If there is a transition from a region Γ_1 to a region Γ_2, then an event a occurs at some timing point t, written as $\Gamma_1 \xrightarrow[t]{a} \Gamma_2$. A sub-region $\beta \subseteq \Gamma$, with $\beta \neq \emptyset$ means that $\beta = (q, V', T')$ with $T' \subseteq T$ and $V' \subseteq V$. A state $\sigma \in \Gamma$ means that $\sigma = (q, v, t)$, with $v \in V$ and $t \in T$. σ satisfies a constraint $\phi \in \Phi(\mathbb{X})$, written as $\sigma \models \varphi$, iff $v \models \varphi$. In the following, we show the semantics of RCTL formulas on the set of all possible runs Π_H.

Definition 15 (Semantics) *Let Ψ is a RCTL formula, H be a hybrid automaton, Π_H be the possible runs of H with a region $\Gamma = (q, V, T) \in \Pi_H$, and ξ is a valuation function of timed-variables. The satisfaction relation $\langle \Pi_H, \Gamma \rangle \overset{T}{\underset{\xi}{\models}} \Psi$, which means that Ψ is satisfied in the region Γ within the time interval (duration) T for some valuation function ξ, is defined inductively as follows:*

- $\langle \Pi_H, \Gamma \rangle \overset{T}{\underset{\xi}{\models}} p$ iff $p = \delta(\Gamma)$.
- $\langle \Pi_H, \Gamma \rangle \overset{T}{\underset{\xi}{\models}} a$ iff *there is* $t' \in T$ *with* $\Gamma \xrightarrow[t']{a} \Gamma'$.
- $\langle \Pi_H, \Gamma \rangle \overset{T}{\underset{\xi}{\models}} \phi$ iff *there is* $\beta \subseteq \Gamma$, *for each* $\sigma_k \in \beta, \sigma_k \models \phi$.
- $\langle \Pi_H, \Gamma \rangle \overset{T}{\underset{\xi}{\models}} y.\Psi$ iff there is $t \in T$ such that $\xi(y) = t$ *and* $\langle \Pi_H, \Gamma \rangle \overset{T:=t}{\underset{\xi}{\models}} \Psi$.
- $\langle \Pi_H, \Gamma \rangle \overset{T}{\underset{\xi}{\models}} \pi$ iff $\xi \models \pi$.
- $\langle \Pi_H, \Gamma \rangle \overset{T}{\underset{\xi}{\models}} \neg\Psi$ iff $\langle \Pi_H, \Gamma \rangle \overset{T}{\underset{\xi}{\not\models}} \Psi$.
- $\langle \Pi_H, \Gamma \rangle \overset{T}{\underset{\xi}{\models}} \Psi_1 \wedge \Psi_2$ iff $\langle \Pi_H, \Gamma \rangle \overset{T}{\underset{\xi}{\models}} \Psi_1$ and $(\Pi_H, \Gamma) \overset{T}{\underset{\xi}{\models}} \Psi_2$.
- $\langle \Pi_H, \Gamma \rangle \overset{T}{\underset{\xi}{\models}} \exists(\Psi_1 U \Psi_2)$ iff there is a run $\Pi \in \Pi_H, \Pi = \Gamma_0, \Gamma_1, \cdots$, with $\Gamma = \Gamma_0$, for some $j \geq 0$, $\langle \Pi_H, \Gamma_j \rangle \overset{T_j}{\underset{\xi}{\models}} \Psi_2$, and $\langle \Pi_H, \Gamma_k \rangle \overset{T_k}{\underset{\xi}{\models}} \Psi_1$ for $0 \leq k < j$.
- $\langle \Pi_H, \Gamma \rangle \overset{T}{\underset{\xi}{\models}} \forall(\Psi_1 U \Psi_2)$ iff for every run $\Pi \in \Pi_H, \Pi = \Gamma_0, \Gamma_1, \cdots$, with $\Gamma = \Gamma_0$, for some $j \geq 0$, $\langle \Pi_H, \Gamma_j \rangle \overset{T_j}{\underset{\xi}{\models}} \Psi_2$, and $\langle \Pi_H, \Gamma_k \rangle \overset{T_k}{\underset{\xi}{\models}} \Psi_1$ for $0 \leq k < j$.

The quantifiers $\forall$, and $\exists$, in the previous semantics, are called paths quantifiers. The variable y in the formula $y.\Psi$ holds the time at which Ψ is satisfied. $y := t$ means that the variable y is set to the value t. $\langle \Pi_H, \Gamma \rangle \overset{T:=t}{\underset{\xi}{\models}} \Psi$ means that the formula Ψ is satisfied in the region Γ when the time T is restricted to the time point t. In case Ψ represents an atomic proposition from the set $Events$, then $y.\Psi$ binds the time at which the event has occurred. This can be used to specify various quantitative properties, such as time bound response properties as we will see in what follows. However, if Ψ represents a constraint formula, then $y.\Psi$ evaluates the time interval at which the constraint Ψ is satisfied. This allows to specify quantitative properties, which could not be specified using events.

Definition 16 (Satisfiability) *Let H be hybrid automaton with initial state init and Π_H as its possible runs. We say that H satisfies the RCTL formula Ψ from init, written as $(H, init) \models \Psi$, iff $(\Pi_H, \Gamma_0) \models \Psi$, where Γ_0 is the initial region of Π_H.*

4 Model Checking as Reachability

For the purpose of verification by means of model checking, we need to describe the properties. Generally, the qualitative properties are often classified into reachability, safety and liveness properties. However, when the time becomes a critical factor to react in the environment, then the concept of safety and liveness properties should be refined. In what follows these types of properties will be reviewed with their specifications by means of RCTL. For the purpose of model checking, these properties will be encoded into suitable queries in Constraint logic program (CLP), which follow the outline of CLP model presented in [27]. However, in order to put model checking within our framework, we will concentrate only the reachability requirements. Indeed, many properties of interest can be specified as a form of reachability, as we will see in the sequel.

4.1 Reachability Requirements

The reachability of a property Ψ means that there is a possibility to reach a state where Ψ holds. In other words, the reachability of the property Ψ asserts that starting from an initial state, is there a region along a run in which Ψ is satisfiable. This can be specified in RCTL as follows $init \rightarrow \exists\Diamond\Psi$, where *init* is the predicate characterizing the set of initial states and is defined as conjunctions of atomic propositions from L and constraints from $\Phi(\mathbb{X})$. It is worth mentioning that checking reachability for hybrid automata is generally undecidable. However, under various constraints of hybrid automata the reachability is decidable. In particular, the decidability result has been proven for for certain classes of hybrid automata including timed and initialized rectangular automata [20].

In terms of the CLP, the reachability of a certain region that satisfies the formula Ψ is done by performing forward reachability analysis from the system's initial state, and then checking whether the conjunction of Ψ with the possible reached regions is satisfied. Assuming for example *init* has been assigned to the set of initial states, the following is the CLP query to check the safety requirements (see [27] for a concrete example).

```
?- reachable(init,Reached),
      member(Ψ1,Reached),ϕ.
```

In the previous query, the formula Ψ is rewritten as a conjunction of two formulas Ψ_1 and ϕ, where $\phi \in (\Phi(\mathbb{X}) \cup \Phi(\mathbb{T}))$ is an atomic the constraint appearing in the formula Ψ. Indeed, any RCTL formula can be rewritten as $\Psi = \Psi_1 \wedge \phi$, if necessary ϕ can be set to true making that conjuct trivial.

A safety property states that *something bad must never happen*. The bad thing represents a critical property that should never occur. Let Ψ represent this critical property, then the safety property is specified as $init :\rightarrow \forall\Box\neg\Psi$. A safety property can be reduced to a reachability property, which can be specified as

$init :\rightarrow \neg\exists\Diamond\Psi$. The previous specification asserts that after executing the initial state $init$, the requirement characterized by Ψ will not be reached.

$$init :\rightarrow \neg\exists\Diamond\Psi.$$

It is often that in certain cases we may be interested in the reachability of a certain property either before or after a time deadline has expired called *Time bounded reachability*. For example, the possibility of a formula Ψ to be reached within the bounded time α is specified in RCTL as $init \rightarrow \exists\Diamond\ (t.\Psi \wedge t \leq \alpha)$.

4.2 Quantitative Requirements

As it is known that a safety property asserts what may or may not occur, but do not require that anything ever does happen. In the train gate example described in [27], closing the gate permanently can maintain the safety of the system, but it is unacceptable for the waiting cars or pedestrians in front of the gate. For this reason, the liveness property is needed to specify such requirements, which asserts that some property of interest will always occur. It should be noted that these type of properties can not be falsified in bounded time. Since the occurrence of some state does not say how long it will take for this state to occur, we can not sure that the liveness property is violated. For this reason, these types of properties are not strong enough in the context of quantitative time properties. Here one would like to see a time bound when the good state occurs. This leads to the next kind of properties.

Bounded Response Properties A bounded response property is one of the most important classes of quantitative requirements used to specify many important applications. It asserts that something will happen within a certain limit of time. A typical application of bounded response property is the specification of worst case performance; that is the specification of an upper bound α on the termination of a system S: if started at time t, then S is guaranteed to reach a final state no later than $\alpha + t$ unit time. For example, specifying that every request will be acknowledged within 3 seconds in communication protocols.

A bounded response property between two events $event_1$ and $event_2$ is specified in RCTL as the formula $init \rightarrow \forall\Box(t_1.event_1 \rightarrow \forall\Diamond(t_2.event_2 \wedge t_2 \leq \alpha + t_1))$. This formula states that whenever there is a request $event_1$ occurs at time t_1, then it is followed by a response $event_2$, at time t_2, such that t_2 is at most $\alpha + t_1$.

It should be mentioned that this property can be falsified within time bound. Therefore this property can be specified as a kind of safety requirement represented as reachability. For this reason, proving the previous property means proving that it is not possible to reach a state where $event_2$ is not reached from $event_1$ within $t_2 \leq \alpha + t_1$. In other words, starting from $event_1$, finding a reachable state satisfies $event_2$, within α time bound, is sufficient to check the reachability of the property. In terms of the CLP, the previous property can be encoded into the following steps: First, we get all possible reachable states from $event_1$ within $t_1 + \alpha$ as L. Second, we check that reachability of $event_2$ has not

been occurred. A positive answer of the reachability indicates a negative answer to the original problem, and vice versa. The following is a CLP query encoding the previous specification:

```
?- reachable(Ψ0,Reached),
   reached_from(L,event1,Reached),
   reached_within(Target, α,L),
   \+ member((_,..,_,event2),Target)
```

We should say that the traditional way to verify this kind of properties using any quantitative time model- checkers—like UPPAAL [10] and Hytech [21]–is to translate that property to what is called a testing automata A, and then check whether the parallel composition of the underlying model together with A can reach a designated violation state. As we said earlier, the reason behind this translation is that there is no direct use of events in the model. The use of events is limited to construct only the parallel composition of automata. In contrast to our adopted approach, the direct use of events with the model allows us to avoid this translation process.

Specifying quantitative properties by means of time of events are not satisfactory in some cases. Suppose for example that one needs to specify that a part of a certain region can be reached in a particular time bound interval. To do so, we present the bounded invariance properties.

Bounded invariance Properties Like the bounded response property, bounded invariance property is one of the most important classes of quantitative timing requirements. It asserts that once an event has been triggered, a certain condition will continuously hold for a certain amount of time. It is often used to specify that something will not happen for a certain period of time. In RCTL this can be specified formally as $init \rightarrow \forall\Box(t_1.event \rightarrow \forall\Box(t_2.\Psi \wedge t_2 \leq \alpha + t_1))$, where α is the duration at which the formula Ψ must be continuously held.

The bounded invariance property can be checked as a safety property. Starting from the time t_1 of the occurence of event , finding a non-reachable violating state for the formula Ψ, within α time bound is sufficient to check the reachability of the property. This can be encoded into CLP as the following

```
?- reachable(Ψ0,Reached),
   reached_from(L,event,Reached),
   reached_within(Target, α,L),
   member((_,..,X,_,Target), X$≤100.
```

A satisfactory solution to the previous query violates the original property.

The way used to specify the bounded invariance properties can be used to specify what is the so-called *minimal event separation* [19] too, i.e no $event_2$ can occur earlier than α time units after an occurrence of $event_1$. This property can be specified as

$$init \rightarrow \forall\Box(t_1.event_1 \rightarrow \forall\Box(t_2 < t_1 + \alpha \rightarrow \neg t_2.event_2)).$$

5 Conclusion and Related Work

This paper introduced the quantitative temporal logic RCTL that extends the well known temporal logic CTL in order to reason about those qualitative and quantitative properties of MAS that occurs as a result of performing continuous actions over time. We used hybrid automata as interpretation model of RCTL. The formulas of RCTL are interpreted on the possible regions produced form the run of hybrid automata. With regions, RCTL combines the expressive power of both state based and event based quantitative temporal logics, which have been proposed already to extend the qualitative temporal logics. The paper showed how to specify and reason about important properties that can be automatically verified by means of reachability analysis. Furthermore, the paper showed how to encode these properties into suitable queries implemented with constraints logic programming.

Reasoning about MAS by means of hybrid automata or timed automata have been approached by several works, for example [12, 13, 15, 26, 23]. These works, however, provide no mean to reason about the quantitative properties in terms of any quantitative temporal logic.

There exist several quantitative temporal logics that can be used to reason about MAS. One can distinguish those temporal logics based on various parameters including the type of computational models; that is linear or computational view of time, the type of accessibility of time; that is whether the time is implicit or explicit in the temporal logics. We classify those works into event or based approach. Examples, of those works, which follow the event based approach, are are *Timed Propositional Temporal Logic* (TPTL) [5], and *Explicit Clock Temporal Logic* [17, 31, 29] for linear time logics, and *Real-Time Computation Tree Logic* (RTCTL) [14] for computation tree time logic. Examples of state based logics are *Metric Temporal logic* (MTL) [25] and *Metric Interval Temporal Logic* (MITL)[3] for linear time with dense semantics, and *Timed Computation Tree Logic* (TCTL) [1], and *Integrator Computation Tree Logic* (ICTL) [6] for computation tree time. The proposed RCTL in this paper tries to combine the expressiveness of both approaches

The idea of combining event based and state based approach is certainly not new. Several works, like [11, 24], motivated their approach by arguing, as we do, that pure state-based or event-based formalisms lack expressiveness in important respects. These works however do not take in consideration the quantitative aspects systems.

As ongoing works, we are currently investigating the complexity of RCTL. We will study the undecidability result of RCTL and try to provide fragments of RCTL which are decidable.

References

1. R. Alur, C. Courcoubetis, and D. Dill. Model-checking in dense real-time. *Inf. Comput.*, 104(1):2–34, 1993.

2. R. Alur and D. Dill. A Theory of Timed Automata. *Theoretical Computer Science*, 126(2):183–235, 1994.
3. R. Alur, T. Feder, and T. A. Henzinger. The benefits of relaxing punctuality. *J. ACM*, 43(1):116–146, 1996.
4. R. Alur and T. Henzinger. Logics and models of real time: A survey. *Real Time: Theory in Practice, Lecture Notes in Computer Science*, 600:74–106, 1992.
5. R. Alur and T. Henzinger. A really temporal logic. *Journal of the ACM (JACM)*, 41(1):203, 1994.
6. R. Alur, T. A. Henzinger, and P.-H. Ho. Automatic symbolic verification of embedded systems. *IEEE Transactions on Software Engineering*, 22(3):181–201, 1996.
7. F. Bacchus and F. Kabanza. Using temporal logics to express search control knowledge for planning. *Artificial Intelligence*, 116(1-2):123 – 191, 2000.
8. P. Bellini, R. Mattolini, and P. Nesi. Temporal logics for real-time system specification. *ACM Comput. Surv.*, 32(1):12–42, 2000.
9. M. Ben-Ari, A. Pnueli, and Z. Manna. The temporal logic of branching time. *Acta Informatica*, 20(3):207–226, 1983.
10. J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson, and W. Yi. Uppaal—a tool suite for automatic verification of real-time systems. In *Proceedings of the DIMACS/SYCON workshop on Hybrid systems III : verification and control*, pages 232–243, Secaucus, NJ, USA, 1996. Springer-Verlag New York, Inc.
11. S. Chaki, E. Clarke, J. Ouaknine, N. Sharygina, and N. Sinha. Concurrent software verification with states, events, and deadlocks. *Formal Aspects of Computing*, 17:461–483, 2005. 10.1007/s00165-005-0071-z.
12. M. Egerstedt. Behavior Based Robotics Using Hybrid Automata. *LECTURE NOTES IN COMPUTER SCIENCE*, pages 103–116, 2000.
13. A. El Fallah-Seghrouchni, I. Degirmenciyan-Cartault, and F. Marc. Framework for Multi-agent Planning Based on Hybrid Automata. *LECTURE NOTES IN COMPUTER SCIENCE*, pages 226–235, 2003.
14. E. A. Emerson, A. K. Mok, A. P. Sistla, and J. Srinivasan. Quantitative temporal reasoning. *Real-Time Syst.*, 4(4):331–352, 1992.
15. U. Furbach, J. Murray, F. Schmidsberger, and F. Stolzenburg. Hybrid multiagent systems with timed synchronization – specification and model checking. In M. Dastani, A. El Fallah Seghrouchni, A. Ricci, and M. Winikoff, editors, *Post-Proceedings of 5th International Workshop on Programming Multi-Agent Systems at 6th International Joint Conference on Autonomous Agents & Multi-Agent Systems*, LNAI 4908, pages 205–220. Springer, 2008.
16. F. Giunchiglia and P. Traverso. Planning as Model Checking. *LECTURE NOTES IN COMPUTER SCIENCE*, pages 1–20, 2000.
17. E. Harel, O. Lichtenstein, and A. Pnueli. Explicit clock temporal logic. In *Proceedings, Fifth Annual IEEE Symposium on Logic in Computer Science, 4-7 June 1990, Philadelphia, Pennsylvania, USA*, pages 402–413. IEEE Computer Society, 1990.
18. T. Henzinger. The theory of hybrid automata. In *Proceedings of the 11th Annual Symposium on Logic in Computer Science*, pages 278–292, New Brunswick, NJ, 1996. IEEE Computer Society Press.
19. T. Henzinger, P.-H. Ho, and H. Wong-Toi. A user guide to HyTech. In *Proceedings of International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, LNCS 1019, pages 41–71. Springer, Berlin, Heidelberg, New York, 1995.
20. T. Henzinger, P. Kopke, A. Puri, and P. Varaiya. What's Decidable about Hybrid Automata? *Journal of Computer and System Sciences*, 57(1):94–124, 1998.

21. T. A. Henzinger, P.-H. Ho, and H. Wong-Toi. Hytech: A model checker for hybrid systems. In *CAV '97: Proceedings of the 9th International Conference on Computer Aided Verification*, pages 460–463, London, UK, 1997. Springer-Verlag.
22. K. V. Hindriks, W. van der Hoek, and M. B. van Riemsdijk. Agent programming with temporally extended goals. In *Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems - Volume 1*, AAMAS '09, pages 137–144, Richland, SC, 2009. International Foundation for Autonomous Agents and Multiagent Systems.
23. G. Hutzler, H. Klaudel, and D. Y. Wang. Towards timed automata and multi-agent systems. In *Formal Approaches to Agent-Based Systems, Third InternationalWorkshop, FAABS 2004, Greenbelt, MD, USA, April 26-27, 2004, Revised Selected Papers*, volume 3228 of *Lecture Notes in Computer Science*, pages 161–172. Springer, 2005.
24. E. Kindler and T. Vesper. Estl: A temporal logic for events and states. In *Proceedings of the 19th International Conference on Application and Theory of Petri Nets*, ICATPN '98, pages 365–384, London, UK, 1998. Springer-Verlag.
25. R. Koymans. Specifying real-time properties with metric temporal logic. *Real-Time Systems*, 2(4):255–299, 1990.
26. A. Mohammed and U. Furbach. Modeling multi-agent logistic process system using hybrid automata. In U. Ultes-Nitsche, D. Moldt, and J. C. Augusto, editors, *In Proceedings of the 7th International Workshop on Modelling, Simulation, Verification and Validation of Enterprise Information Systems, MSVVEIS 2008*, pages 141–149, Barcelona, Spain, 2008. INSTICC PRESS. Held in conjunction with 10th International Conference on Enterprise Information Systems (ICEIS 2008).
27. A. Mohammed and U. Furbach. Multi-agent systems: Modeling and verification using hybrid automata. In J.-P. B. Lars Braubach and J. Thangarajah, editors, *Programming Multi-Agent Systems:7th International Workshop,ProMAS2009, Budapest, Hungary, May 2009, Revised Selected Papers*, LNAI 5919, pages 49–66. Springer, Berlin, Heidelberg, 2010.
28. D. Nau, M. Ghallab, and P. Traverso. *Automated Planning: Theory & Practice*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.
29. J. Ostroff and W. Wonham. A framework for real-time discrete event control. *IEEE Transactions on Automatic Control*, 35(4):386–397, 1990.
30. A. Pnueli. The temporal logic of programs. In *Foundations of Computer Science, 1977., 18th Annual Symposium on*, pages 46–57, 1977.
31. A. Pnueli and E. Harel. Applications of temporal logic to the specification of real-time systems. In *Systems, Proceedings of a Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 84–98, London, UK, 1988. Springer-Verlag.
32. C. Schwarz, A. Mohammed, and F. Stolzenburg. A tool environment for specifying and verifying multi-agent systems. In J. Filipe, A. Fred, and B. Sharp, editors, *Proceedings of the 2nd International Conference on Agents and Artificial Intelligence*, volume 2, pages 323–326. INSTICC Press, 2010.
33. J. Van Benthem and A. ter Meulen, editors. *Handbook of Logic and language*. Elsevier, 1997.
34. S. Yovine. Kronos: A verification tool for real-time systems. *International Journal on Software Tools for Technology Transfer (STTT)*, 1(1):123–133, 1997.

State Space Reduction for Model Checking Agent Programs

Sung-Shik T.Q. Jongmans[1], Koen V. Hindriks[2], and M. Birna van Riemsdijk[2]

[1] Centrum Wiskunde & Informatica, Amsterdam, the Netherlands
[2] Delft University of Technology, Delft, the Netherlands

Abstract. State space reduction techniques have been developed to increase the efficiency of model checking in the context of imperative programming languages. Unfortunately, these techniques cannot straightforwardly be applied to agents: the nature of states in the two programming paradigms differs too much for this to be possible. To resolve this, we adapt core definitions on which existing reduction algorithms are based to agents. Moreover, the framework that we introduce is such that different reduction algorithms can be defined in terms of the same relations. This is beneficial because it enables the reuse of code and reduces computation time when different techniques are used simultaneously. Specifically, we adapt and combine two known techniques: property-based slicing and partial order reduction. We exemplify our work with the GOAL agent programming language, and implement the theory that we present for GOAL. Several experiments with this implementation show that performance is in line with known results from traditional model checking.

1 Introduction

Model checking techniques for the verification of programs have traditionally been developed in the context of *imperative* programming languages (IPL). Ideally, for model checking programs written in *agent* programming languages (APL), one would take the technology and tools developed for IPLs, and apply them to agent programs without too much alteration. Unfortunately, this is sometimes an INEFFICIENT solution, and sometimes even IMPOSSIBLE:

INEFFICIENT — In [11], we show that it can be beneficial to develop new model checkers tailored to the verification of an APL rather than reusing existing tools for agent verification. The reason is that APL-tailored model checkers can reuse the APL's standard interpreter for fast generation of states. Consequently, there is no need to encode the agent program to lower-level code serving as input to an existing tool, which typically blows up the state space.

IMPOSSIBLE — In this paper, we argue that *state space reduction techniques*,[3] henceforth simply *reduction techniques*, known from traditional model check-

[3] State space reduction techniques combat the *state space explosion problem* (common to both IPL and APL model checking). This is the problem that systems to be verified are typically huge in terms of their *state space*, rendering model checking such systems often beyond our reach: it takes too many resources to finish verification.

ing cannot be applied directly in an agent context. The reason is that (dependencies between) states and transitions in the transition system of an imperative program differ fundamentally from those of an agent program.

Our main contribution is the redefinition, for agents, of concepts at the heart of existing reduction algorithms with a novel framework that brings together different techniques in a unifying way: we show that both *property-based slicing* (PBS) and *partial order reduction* (POR) can be defined in terms of the same relations using our framework. This enables a shared code base and runtime synergy: computations carried out for one algorithm can be reused by the other. We use the GOAL agent language [6] as running example throughout the paper.

The remainder is organised as follows. Section 2 provides background on model checking and GOAL. In Sect. 3, we argue why existing reduction techniques cannot straightforwardly be applied to agents, and introduce our framework. In Sect. 4, we define PBS and POR algorithms in terms of this framework. Section 5 discusses our implementation. Finally, Sect. 6 discusses related work with respect to reduction techniques in agent verification, and concludes the paper.

2 Preliminaries

Model checking Model checking [4] is a technique for automatically establishing whether a program P satisfies a property φ. Usually, φ is expressed in a *temporal logic*, a formalism for describing change over time. In this paper, we consider *linear temporal logic* (LTL) [4]. An LTL formula, denoted by ϕ or φ (if φ is a property to be model checked), is built from a set of propositions $\mathcal{P}$, the boolean connectives, and the temporal operators $\bigcirc$ (next), $\mathcal{U}$ (weak until), and $\mathcal{R}$ (strong release). We denote the set of all LTL formulas by $\mathcal{L}$. An LTL formula is interpreted over an infinite sequence of states, which we call a *computation*, denoted by $\boldsymbol{\pi}$. Let $i \geq 0$ be an index of $\boldsymbol{\pi}$, and let $\models$ be LTL's entailment relation. Purely propositional (sub)formulas are interpreted with respect to the i-th state on $\boldsymbol{\pi}$, denoted by $\boldsymbol{\pi}_i$, using a *valuation function* $\mathcal{V}$. Such a function maps a state to the set of propositions in $\mathcal{P}$ that are true in it. Temporal (sub)formulas are interpreted with respect to the (infinite) postfix of $\boldsymbol{\pi}$ starting in the i-th state:

$$
\begin{array}{ll}
\boldsymbol{\pi}, i \models \bigcirc \phi & \text{iff } \boldsymbol{\pi}, i+1 \models \phi \\
\boldsymbol{\pi}, i \models \phi \,\mathcal{U}\, \phi' & \text{iff } \exists_{k \geq i}(\boldsymbol{\pi}, k \models \phi' \text{ and } \forall_{i \leq j < k}(\boldsymbol{\pi}, j \models \phi)) \\
\boldsymbol{\pi}, i \models \phi \,\mathcal{R}\, \phi' & \text{iff } \boldsymbol{\pi}, i \models \neg(\neg\phi \,\mathcal{U}\, \neg\phi') \qquad \text{(note that } \mathcal{R} \text{ is the dual of } \mathcal{U}\text{)}
\end{array}
$$

In model checking, the program P is represented by its *transition system* $\mathcal{T} = \langle M, \mu_0, \longrightarrow \rangle$ in which M is a finite *set of states*, $\mu_0 \in M$ is the *initial state*, and $\longrightarrow \subseteq M \times M$ is a *transition relation* connecting states. A *path* π through $\mathcal{T}$ is an infinite sequence of states $\pi_0 \pi_1 \cdots$ such that for all $i \geq 0$: $\pi_i, \pi_{i+1} \in M$ and $\pi_i \longrightarrow \pi_{i+1}$. A computation $\boldsymbol{\pi}$ of P is a path through its transition system that starts in μ_0, i.e. $\boldsymbol{\pi}_0 = \mu_0$. We denote the set of all computations of P by $\boldsymbol{\Pi}$. The model checking problem for P and φ, given a valuation function $\mathcal{V}$, can now be formulated more formally as follows: determine for all $\boldsymbol{\pi} \in \boldsymbol{\Pi}$ whether $\boldsymbol{\pi}, 0 \models \varphi$.

In that case, we say that P *satisfies* φ. Otherwise, if there exists a $\boldsymbol{\pi} \in \boldsymbol{\Pi}$ such that $\boldsymbol{\pi}, 0 \models \neg\varphi$, P is said to *violate* φ, and $\boldsymbol{\pi}$ is called a *counterexample*.

Various approaches to model checking exist. In this paper, we assume *NDFS explicit-state automata-theoretic LTL model checking* [4], because the implementation we discuss in Sect. 5 extends [11] in which this approach is also taken.[4] In this approach, every $\boldsymbol{\pi} \in \boldsymbol{\Pi}$ is checked for satisfaction of $\neg\varphi$ in *negation normal form* (NNF). If such a computation is found, the model checker immediately halts, and reports it as a counterexample. Otherwise, the model checker terminates after investigating all computations, reporting that $\neg\varphi$ is not satisfied by any computation, i.e. φ is satisfied by all computations. Thus, instead of determining if all computations satisfy φ, in fact one determines whether there exists a counterexample. Henceforth, we assume all LTL formulas in NNF.

An important optimisation that the sketched approach allows for is *on-the-fly exploration*: the transition system of the program under investigation is generated *during* execution of the model checking algorithm instead of *before* it. Consequently, if a counterexample is quickly found and the model checker terminates, no resources have been spent on the generation of parts of the transition system whose inspection has turned out unnecessary. Importantly, the reduction algorithms discussed next are compatible with on-the-fly model checking.

GOAL The Goal agent programming language [6] facilitates programming of *rational agents* (i.e. agents that pursue their goals) at the cognitive level: agents choose their actions by reasoning about their *beliefs* and *goals*, which are expressed in some knowledge representation language $\mathcal{L}_X$ (e.g. Prolog). The beliefs that a Goal agent has at some point in time are stored in its *belief base*, denoted by Σ. Similarly, the goals of a Goal agent are stored in its *goal base*, denoted by Γ. Goals are *declarative*: they specify *what* the desired state of the world is instead of *how* this state may be brought about. Together, the belief and goal base of an agent constitute its *mental state*, denoted by $\mu = \langle \Sigma, \Gamma \rangle$.[5]

A Goal agent derives its choice of action from its mental state, hence it needs a mechanism to inspect it. To this end, agents evaluate *mental state conditions* (MSC). An MSC, denoted by ψ, is a boolean expression about the beliefs and goals of an agent, according to the following syntax:

$$\begin{aligned} \chi &::= \text{ any well-formed formula from } \mathcal{L}_X \\ \psi &::= \mathbf{bel}(\chi) \mid \mathbf{goal}(\chi) \mid \neg\psi \mid \psi \wedge \psi \end{aligned}$$

The semantics of MSCs is defined by the entailment relation $\models_{\text{MS}}$ [6]. Informally, if μ is a mental state then $\mu \models_{\text{MS}} \mathbf{bel}(\chi)$ is true if χ is believed by the agent;

[4] Another well-known approach is *symbolic model checking* using *binary decision diagrams* (BDD) [4, 13]. This approach is based on an abstraction technique different from the techniques discussed here and is out of scope of this work.

[5] Although we do not discuss knowledge, our implementation is able to deal with this; in contrast, modules [6], percepts, and beliefs about dynamic environments that evolve independently of the agent's acting are at present beyond our scope.

similarly, $\mu \models_{\text{MS}} \mathbf{goal}(\chi)$ is true if χ is a goal of the agent. The set of all MSCs, denoted by $\mathcal{L}_{\text{MS}}$, is called the *language of mental state conditions*.

MSCs are used in the definition of *action rules*. An action rule, denoted by ρ, is a statement of the form **if** ψ **then** α in which α is an *action*. An action rule may be read as "if ψ is true, then the agent may consider performing α". In that case, the action rule is said to be *applicable*. The effects that performance of an action have on the mental state of an agent are formalised by the *mental state transformer*, denoted by $\mathcal{M}$. The mental state transformer maps an action and a mental state to a *successor mental state*. $\mathcal{M}$ need not be defined for all mental state–action pairs $\langle \mu, \alpha \rangle$: if $\mathcal{M}$ is undefined for μ and α, this means that α cannot be performed in μ. A precise definition of $\mathcal{M}$ is given in [6].

Let μ be a mental state, and let $\rho =$ **if** ψ **then** α be an action rule. If ρ is applicable in μ (i.e. $\mu \models_{\text{MS}} \psi$) and $\mathcal{M}(\alpha, \mu)$ is defined, then α is called an *option* in μ. During each reasoning cycle, a GOAL agent determines its options given its current mental state and set of action rules, and chooses and performs one of them non-deterministically. This is formalised by an *operational semantics*. Let **if** ψ **then** α be an action rule, and let μ be a mental state. Then, the transition relation $\longrightarrow$ is the smallest relation induced by the following transition rule:

$$\frac{\mu \models_{\text{MS}} \psi \qquad \mathcal{M}(\alpha, \mu) \text{ is defined}}{\mu \longrightarrow \mathcal{M}(\alpha, \mu)}$$

The transition relation $\longrightarrow$ is subsequently used to define the transition system $\mathcal{T} = \langle M, \mu_0, \longrightarrow \rangle$ of a GOAL agent, in which we assume that M is a finite[6] set of mental states and that μ_0 is the initial mental state of the agent.

Example 1. The source code and transition system of a simple example GOAL agent, whose task is to put on two socks, appears in Fig. 1. We use this agent, called `socksAgent`, as a running example throughout this paper.

For model checking GOAL agents, we instantiate the set of LTL propositions $\mathcal{P}$ with the language of mental state conditions $\mathcal{L}_{\text{MS}}$. The valuation function $\mathcal{V}$ in this case maps every mental state μ to the MSCs that are true in it, i.e. $\mathcal{V}(\mu) = \{ \psi \in \mathcal{L}_{\text{MS}} \mid \mu \models_{\text{MS}} \psi \}$. This allows us to formulate and verify properties about the evolution of beliefs and goals of a GOAL agent during its execution.

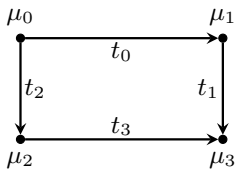
A final remark on terminology. Although we illustrate our techniques with GOAL, they can be applied to other agent languages as well. Therefore, when we write "mental state" in what follows, we do not refer exclusively to a state of a GOAL agent, but rather to a state of an agent written in some BDI-based APL.

3 Operations on Mental States

The aim of reduction techniques is to remove sets of transitions from the transition system that do not affect the truth value of the property under investigation. In our framework, we identify such sets of transitions by classifying them

[6] Finiteness is not imposed by GOAL, but a model checking termination requirement.

```
 1. main: socksAgent{
 2.   beliefs{
 3.     bothSocksOn :- wearing(sock,left),
 4.                    wearing(sock,right).
 5.   }
 6.   goals{
 7.     wearing(sock,left).
 8.     wearing(sock,right).
 9.   }
10.   program{
11.     if   goal(wearing(sock,left))
12.     then putOn(sock,left).
13.     %
14.     if   goal(wearing(sock,right))
15.     then putOn(sock,right).
16.   }
17.   action-specs{
18.     putOn(S,X){
19.       pre{ not(wearing(S,X)) }
20.       post{ wearing(S,X) }
21. } } }
```



$$\mu_0 = \left\langle \emptyset \,,\, \left\{ \begin{array}{l} \texttt{wearing(sock,left)}, \\ \texttt{wearing(sock,right)} \end{array} \right\} \right\rangle$$

$$\mu_1 = \left\langle \begin{array}{l} \{\texttt{wearing(sock,left)}\}, \\ \{\texttt{wearing(sock,right)}\} \end{array} \right\rangle$$

$$\mu_2 = \left\langle \begin{array}{l} \{\texttt{wearing(sock,right)}\}, \\ \{\texttt{wearing(sock,left)}\} \end{array} \right\rangle$$

$$\mu_0 = \left\langle \left\{ \begin{array}{l} \texttt{wearing(sock,left)}, \\ \texttt{wearing(sock,right)} \end{array} \right\} , \emptyset \right\rangle$$

Fig. 1. Example agent. On the left, its source code; on the right, its transition system.

in terms of *operations*. Informally, we may think of an operation, denoted by τ, as a function that transforms states μ to other states μ'. In that case, τ is said to be *applied* to μ. More specifically, we characterise an operation in terms of the CHANGES that it brings about, and the STATEMENT in the source code from which it can be induced. Below, let $\mathcal{T} = \langle M, \mu_0, \longrightarrow \rangle$ be the transition system of some agent program P, and let $t = \langle \mu, \mu' \rangle \in \longrightarrow$ be a transition.

CHANGES — Grouping individual transitions in $\mathcal{T}$ according to the changes that they bring about enables us to express that the order in which two operations can be applied is without consequence (relevant in POR). To formalise this notion, let $\mathsf{Ch}(t)$ denote the change between μ and μ'.

STATEMENT — Characterising operations by statements allows us to remove sets of transitions from $\mathcal{T}$ by deleting statements from P's source code. This enables us, for instance, to reduce $\mathcal{T}$ by performing static analysis of the program text alone (relevant in PBS). To formalise this notion, let $\mathsf{St}(t)$ denote the set of statements in P's source code from which t can be induced.

Example 2. In case of GOAL, $\mathsf{Ch}(t)$ denotes the beliefs and goals to be added and deleted to get from μ to μ', and $\mathsf{St}(t)$ denotes the action rules that induce t. Applied, for instance, to transition $t_0 = \langle \mu_0, \mu_1 \rangle$ of `socksAgent` in Fig. 1 yields: $\mathsf{Ch}(t_0) = \langle \Sigma + \{\,\texttt{wearing(sock,left)}\,\} - \emptyset, \Gamma + \emptyset - \{\,\texttt{wearing(sock,left))}\,\} \rangle$ and $\mathsf{St}(t_0) = \{\,\texttt{if goal(wearing(sock,left)) then putOn(sock,left)}\,\}$.

We now define an operation τ formally.

Definition 1. *An operation is a pair $\tau = \langle T, s \rangle$ in which s is a statement and $T \subseteq \longrightarrow$ is the largest set such that for all $t, t' \in T$: $\mathsf{Ch}(t) = \mathsf{Ch}(t')$ and $s \in \mathsf{St}(t)$.*

Example 3. We identify the following operations of `socksAgent` in Fig. 1:

$\tau_0 = \langle \{ t_0, t_3 \}, \texttt{if goal(wearing(sock,left)) then putOn(sock,left)} \rangle$
$\tau_1 = \langle \{ t_1, t_2 \}, \texttt{if goal(wearing(sock,right)) then putOn(sock,right)} \rangle$

We use the following notation and definitions. The set of all possible operations is denoted by Ω_τ. If $\tau = \langle T, s \rangle$ is an operation, then we use $\mathsf{Tran}(\tau)$ and $\mathsf{Stat}(\tau)$ as a shorthand for, respectively, T and s. We call $\mathsf{Stat}(\tau)$ the statement that *induces* τ, and say that τ is *enabled* in a state μ if there exists a μ' such that $\langle \mu, \mu' \rangle \in \mathsf{Tran}(\tau)$; we write $\tau(\mu)$ as a shorthand for μ'. The set of all enabled operations in μ is denoted by $\mathsf{En}(\mu)$, i.e. $\mathsf{En}(\mu) = \{ \tau \in \Omega_\tau \mid \tau \text{ is enabled in } \mu \}$. The set of all operations $\mathsf{Ops}(s)$ that a statement s can induce is called its *operation class*, defined as $\mathsf{Ops}(s) = \{ \tau \in \Omega_\tau \mid \mathsf{Stat}(\tau) = s \}$. Finally, for brevity, we write $\mathsf{Ch}(\tau)$ to denote the change brought about by any $t \in \mathsf{Tran}(\tau)$, and write $\mathsf{Ch}(s)$ to denote the set at least having $\bigcup_{\tau \in \mathsf{Ops}(s)} \mathsf{Ch}(\tau)$ as a subset.

3.1 Variable Assignments versus Mental States

State space reduction techniques have originally been developed for use with transition systems whose states are characterised by variables and their values, henceforth called *variable assignment*. By carefully analysing which variables change by applying operations on states (i.e. when moving from one state to the next), relations on operations essential to the application of reduction algorithms can be computed. For instance, one can determine whether enabledness of an operation τ' is affected by the application of an operation τ, by comparing the variables that τ mutates and τ' accesses. We call the sets of variables an operation τ accesses and mutates its *read set*, denoted by $\mathsf{Read}(\tau)$, and its *write set*, denoted by $\mathsf{Write}(\tau)$, respectively. These sets are not used only for determining whether enabledness of operations depends on the application of (other) operations, but also to determine if the application of an operation influences the truth value of LTL formulas. Importantly, analyses based on read and write sets can be done by inspection of the source code alone: the read and write set of an operation τ can be determined straightforwardly by inspecting the variables occurring in the statement that induces τ, i.e. $\mathsf{Stat}(\tau)$. This is of great value, because it allows for *off-line computation* of (most of the) reduction algorithms, meaning that their computation does not depend on information that is available only during model checking, and reducing their overhead at runtime to a minimum as a result.

Example 4. Suppose two operations $\tau, \tau' \in \Omega_\tau$ such that $\mathsf{Stat}(\tau) = [x := x + 1]$ and $\mathsf{Stat}(\tau') = [y := z + 42]$ are enabled simultaneously in some variable assignment ν, e.g. because they belong to different concurrent processes (and x, y, z are shared variables). Then: $\mathsf{Read}(\tau) = \mathsf{Write}(\tau) = \{x\}$ and $\mathsf{Read}(\tau') = \{z\}$ and $\mathsf{Write}(\tau') = \{y\}$. Because $\mathsf{Read}(\tau) \cap \mathsf{Write}(\tau') = \mathsf{Write}(\tau) \cap \mathsf{Read}(\tau') = \emptyset$, application of τ cannot cause τ' to become disabled and vice versa.

When model checking agent programs, however, states are *not* characterised by variable–value pairs, but by mental attitudes, which are very different: how and

which mental attitudes change over time is not stated explicitly in the program text, e.g. due to underspecification. We elaborate on this in Sect. 3.2. Hence, in agent verification, we cannot use directly the analysis techniques known from traditional model checking to compute the relations essential to the application of reduction algorithms: the gap between variable assignments and mental states need be bridged. Specifically, to be able to reuse existing reduction algorithms for agents, we need to answer (in the next subsection) the following questions:

1. What are the elements constituting read and write sets when dealing with mental states of agents, which are not composed of variable–value pairs?
2. Given a definition of read and write sets for mental states of agents, can we still compute them off-line?

3.2 Read Sets and Write Sets for Mental States

Ad 1. We aim at a definition of read and write sets for mental states that is sufficiently generic in the sense that these definitions should accommodate multiple APLs. This is nontrivial, because mental states look different in each APL, i.e. the mental attitudes constituting a mental state vary between different languages. To this end, we introduce the notion of an APL-specific *condition language*, denoted by $\mathcal{L}_K$, whose elements are *conditions*, denoted by κ. Informally, the idea is that the read set of an operation τ contains those conditions that *must* be true for τ to be enabled, while τ's write set contains those conditions whose truth value changes due to application of τ. Thus, $\mathsf{Read}(\tau) \subseteq \mathcal{L}_K$ and $\mathsf{Write}(\tau) \subseteq \mathcal{L}_K$. The only requirement that $\mathcal{L}_K$ must satisfy is that it should have the set of propositions $\mathcal{P}$ as a subset, i.e. $\mathcal{P} \subseteq \mathcal{L}_K$: this allows us to determine, by means of write set analysis, whether a transition can affect the truth value of a property. Apart from that, $\mathcal{L}_K$ can be tailored completely to the needs of the APL.

Example 5. In the context of GOAL, the condition language equals the language of MSCs, i.e. $\mathcal{L}_K = \mathcal{L}_{\text{MS}}$ (recall that $\mathcal{P} = \mathcal{L}_{\text{MS}}$ for GOAL).

Next, to accommodate formal definitions, we assume an entailment relation $\models_K$, relating (mental) states to conditions that are true in them, and a function $\mathcal{I}$ mapping a mental state μ to the subset of $\mathcal{L}_K$ that is true in μ, i.e. $\mathcal{I}(\mu) = \{\kappa \in \mathcal{L}_K \mid \mu \models_K \kappa\}$. Read and write sets are then defined formally as follows.

Definition 2. *Let τ be an operation. Then:*

$$
\begin{aligned}
\mathsf{Read}(\tau) &= \{\kappa \in \mathcal{L}_K \mid \textit{there exist states } \mu, \mu' \textit{ s.t. } \tau \in \mathsf{En}(\mu), \tau \notin \mathsf{En}(\mu') \\
&\qquad\qquad \textit{and } \kappa \in \mathcal{I}(\mu) \textit{ and } \mathcal{I}(\mu') = \mathcal{I}(\mu) \setminus \{\kappa\} \qquad\qquad \} \\
\mathsf{Write}^{+}(\tau) &= \textstyle\bigcup_{\langle \mu, \mu' \rangle \in \mathsf{Tran}(\tau)} \mathcal{I}(\mu') \setminus \mathcal{I}(\mu) \\
\mathsf{Write}^{-}(\tau) &= \textstyle\bigcup_{\langle \mu, \mu' \rangle \in \mathsf{Tran}(\tau)} \mathcal{I}(\mu) \setminus \mathcal{I}(\mu') \\
\mathsf{Write}(\tau) &= \mathsf{Write}^{+}(\tau) \cup \mathsf{Write}^{-}(\tau)
\end{aligned}
$$

We call $\mathsf{Write}^{+}(\tau)$ and $\mathsf{Write}^{-}(\tau)$ the positive and negative write sets of τ; $\mathsf{Write}(\tau)$ is sometimes referred to as τ's total write set.

We use the distinction between positive and negative write sets in Sect. 3.3. The distinction is important, because it allows us, for instance, to state that some transition τ can enable a transition τ': in that case, the *positive* write set of τ coincides with the read set of τ'. Conversely, if τ's *negative* write set does *not* coincide with the read set of τ', τ cannot disable τ'. Note that "not disabling" is different from "enabling", and in general, Write^+ and Write^- are not each other's complement: $\mathcal{L}_K \setminus \mathsf{Write}^+(\tau) \neq \mathsf{Write}^-(\tau)$ and $\mathcal{L}_K \setminus \mathsf{Write}^-(\tau) \neq \mathsf{Write}^+(\tau)$.

Example 6. Consider operation τ_0 of `socksAgent`, defined in Ex. 3. For convenience, we restrict this example to the MSC set $\{$`goal(wearing(sock,left))`, `goal(wearing(sock,right))`, `bel(bothSocksOn)`$\} \subset \mathcal{L}_{\text{MS}}$. Now, the positive write set of τ_0 equals $\{$`bel(bothSocksOn)`$\}$, while both its read set and negative write set equal $\{$`goal(wearing(sock,left))`$\}$. From this, we can deduce that τ_0 disables itself, while it has no effect on enabledness or disabledness of τ_1.

Ad 2. As outlined in Sect. 3.1, off-line computation of read and write sets is important, because it reduces the resource consumption of reduction algorithms at runtime. For imperative programming languages, as shown in Ex. 4, this can be done easily. Unfortunately, in case of agent programs, the situation is more complex: conditions from $\mathcal{L}_K$ often do not occur explicitly in the agent's source code, and cannot be simply extracted from it without further analysis.

Example 7. Consider the read and write sets of operation τ_0 of `socksAgent` given in Ex. 6. While τ_0's read set can be determined straightforwardly from the action rule `if goal(wearing(sock,left)) then putOn(sock,left)`, this is not the case for its write set for two reasons. First, the removal of the goal `wearing(sock,left)` occurs automatically due to GOAL's semantics, and is not specified explicitly in the program text. Second, the derivation of `bothSocksOn` using the Prolog rule in the belief base (see Fig. 1) cannot be detected by inspection of this action rule alone.

Switching to a more general perspective, we must deal with two issues when computing read and write sets for GOAL agents. First, not all beliefs and goals that an operation adds or deletes can be derived from the source code of a GOAL agent, making it difficult to determine which MSCs incur a change of truth value. Second, as changing the belief base by an operation also changes the consequences that can be derived from Prolog rules, we need an algorithm to approximate these. The issue is that this algorithm must run on only the source code and that the content of the belief base at runtime is unknown.

Thus, we may need to derive read and write sets with more complex analysis techniques. Unfortunately, it may be impossible to compute *precise* read and write sets using the source code alone due to underspecification of the agent or the occurrences of uninstantiated variables combined with Prolog-style reasoning as sketched in the previous example. There are two ways to resolve these issues: acquire sufficient information by generating the entire transition system, or use *approximation techniques*. We prefer the latter, because the former is incompatible with on-the-fly model checking. We stress that approximation is unnecessary

Table 1. Formal definition of relations on operations and statements.

Relation	Precise (for operations τ, τ')	Approximate (for statements s, s')
Visibility	$\mathsf{Vis}(\tau, \phi)$ iff $Props(\phi) \cap \mathsf{Write}(\tau) \neq \emptyset$	$\mathfrak{Vis}(s, \phi)$ iff $Props(\phi) \cap \mathfrak{Write}(s) \neq \emptyset$
Enables	$\mathsf{Enables}(\tau, \tau')$ iff $\mathsf{Read}(\tau') \cap \mathsf{Write}^+(\tau) \neq \emptyset$	$\mathfrak{Enables}(s, s')$ iff $\mathfrak{Read}(s') \cap \mathfrak{Write}^+(s) \neq \emptyset$
Independence	$\mathsf{Indep}(\tau, \tau')$ iff $H^{en}_{\mathsf{Indep}}(\tau, \tau')$ and $H^{comm}_{\mathsf{Indep}}(\tau, \tau')$	$\mathfrak{Indep}(s, s')$ iff $H^{en}_{\mathfrak{Indep}}(s, s')$ and $H^{comm}_{\mathfrak{Indep}}(s, s')$

in an IPL context, because there, read and write sets can be obtained with straightforward source code inspection.

The key property any approximation technique for read and write sets must satisfy is that of *over*-approximation: to ensure that model checking with reduction algorithms yields the same results as without, approximate read and write sets (denoted here in $\mathfrak{font}$) need to over-approximate the precise sets. Formally:

Property 1. Let s be a statement. For all $\tau \in \mathsf{Ops}(s)$: $\mathsf{Read}(\tau) \subseteq \mathfrak{Read}(s)$ and $\mathsf{Write}^+(\tau) \subseteq \mathfrak{Write}^+(s)$ and $\mathsf{Write}^-(\tau) \subseteq \mathfrak{Write}^-(s)$ and $\mathsf{Write}(\tau) \subseteq \mathfrak{Write}(s)$.

Intuitively, over-approximation of read and write sets is required because these sets are used to determine dependencies between operations: the less dependencies present, the more reduction can be obtained. Thus, if all operations depend on each other, no reduction is gained. By over-approximating, dependencies that actually do not exist are nevertheless assumed. Although this may cause reduction algorithms to be less effective, correctness is assured. Henceforth, we assume all sets $\mathfrak{Read}$ and $\mathfrak{Write}$ to satisfy Property 1 (e.g. in the proof of Lemma 1).

3.3 Relations on Operations

Next, we use read and write sets to define relations on operations known from existing literature [4] on reduction techniques (see Table 1), and used by the algorithms in Sect. 4. Our contribution is that we define each relation not only in terms of precise read and write sets, but also in terms of their approximate counterparts. The resulting *approximate relations* can be computed before the transition system is generated (instead of during its generation), i.e. off-line. This reduces computational overhead of reduction algorithms at runtime to a minimum, and ensures compatibility with on-the-fly model checking. We prove lemmas to show how the precise and approximate relations relate to each other.

The first relation we discuss is the *visibility relation* Vis. Let τ be an operation, and let ϕ be an LTL formula. Then, $\mathsf{Vis}(\tau, \phi)$ states that application of τ can affect the truth value of ϕ; the formal definition can be found in Table 1. Because Vis is defined in terms of precise write sets, which typically cannot be computed off-line (see Sect. 3.2), we introduce the *approximate visibility relation* $\mathfrak{Vis}$, which is an approximation of Vis defined in terms of approximate write sets (see Table 1). Relations Vis and $\mathfrak{Vis}$ are related by the following lemma.

Table 2. Independence conditions, definitions, and heuristics.

Condition	Heuristic	Approximate heuristic
ENABLEDNESS : $\tau \in \mathsf{En}(\tau'(\mu))$	$H^{en}_{\mathsf{Indep}}(\tau, \tau')$: $\mathsf{Read}(\tau') \cap \mathsf{Write}^{-}(\tau) = \emptyset$	$H^{en}_{\mathfrak{Indep}}(s, s')$: $\mathfrak{Read}(s') \cap \mathfrak{Write}^{-}(s) = \emptyset$
COMMUTATIVITY : $\tau(\tau'(\mu)) = \tau'(\tau(\mu))$	$H^{comm}_{\mathsf{Indep}}(\tau, \tau')$: $\mathsf{Ch}(\tau) \cap \mathsf{Ch}(\tau') = \emptyset$	$H^{comm}_{\mathfrak{Indep}}(s, s')$: $\mathsf{Ch}(s) \cap \mathsf{Ch}(s') = \emptyset$

Lemma 1. *Let s be statement, let τ be an operation such that $\mathsf{Stat}(\tau) = s$, and let ϕ be an LTL formula. If $\mathsf{Vis}(\tau, \phi)$, then $\mathfrak{Vis}(s, \phi)$.*

Proof. By definition of Vis *in Table 1, $Props(\phi) \cap \mathsf{Write}(\tau) \neq \emptyset$. Also, because* $\mathfrak{Write}$ *satisfies Property 1, $\mathfrak{Write}(s) \subseteq \mathsf{Write}(\tau)$. Hence, $Props(\phi) \cap \mathfrak{Write}(s) \neq \emptyset$. The lemma then follows from the definition of* $\mathfrak{Vis}$ *in Table 1.* □

Thus, $\mathfrak{Vis}(s, \phi)$ is true if s induces an operation τ whose application affects the truth value of ϕ, as such over-approximating the relation Vis.

The second relation we discuss is the *enables relation* Enables. Let τ, τ' be operations. Then, $\mathsf{Enables}(\tau, \tau')$ states that application of τ to some state μ can cause τ' to become enabled, i.e. τ is enabled in μ while τ' is not, but in the state that results from applying τ to μ, τ' *is* enabled. The formal definition (in terms of precise read and write sets) occurs in Table 1, together with the definition of the *approximate enables relation* $\mathfrak{Enables}$ (in terms of approximate read and write sets). Relations Enables and $\mathfrak{Enables}$ are related by the following lemma, whose proof is analogous to that of Lemma 1 (omitted for reasons of space).

Lemma 2. *Let s, s' be statements, and let τ, τ' be operations such that $\mathsf{Stat}(\tau) = s$ and $\mathsf{Stat}(\tau') = s'$. If $\mathsf{Enables}(\tau, \tau')$, then $\mathfrak{Enables}(s, s')$.*

Thus, $\mathfrak{Enables}(s, s')$ is true if s induces an operation τ whose application can enable an operation τ' induced by s', over-approximating the relation Enables.

The third relation we discuss is the *independence relation* Indep. Let τ, τ' be operations. Then, $\mathsf{Indep}(\tau, \tau')$ is true if the *independence conditions* in the left column of Table 2 hold for each state μ of the transition system: ENABLEDNESS states that independent operations cannot *disable* each other, while COMMUTATIVITY states that applying independent operations in either order results in the same state. In practice, checking the independence conditions in each state would be too much a computational burden. Therefore, as usual [4], Indep is defined heuristically (see Table 1 and the middle column of Table 2).

We approximate ENABLEDNESS with condition H^{en}_{Indep} given in Table 2, which is guaranteed to be true if ENABLEDNESS is true. The intuition behind it is that if an operation τ does not disable an operation τ', then τ cannot make a condition κ on which enabledness of τ' depends (i.e. $\kappa \in \mathsf{Read}(\tau')$) false. Similarly, COMMUTATIVITY is approximated with $H^{comm}_{\mathsf{Indep}}$ in Table 2. The intuition behind $H^{comm}_{\mathsf{Indep}}$ is that if the orders in which operations τ and τ' can be applied both lead to the same state, the changes that they bring about are disjoint, i.e. τ does not (partially) undo changes brought about by τ' and vice versa.

Definitions of H^{en}_{Indep} and $H^{comm}_{\mathsf{Indep}}$ (similar to those in [4]) are in terms of operations instead of statements: to be able to compute independences before actual model checking, we require the latter. Therefore, as before, we introduce the *approximate independence relation* $\mathfrak{Indep}$, in whose definition (see Table 1) the precise heuristics have been replaced by their approximate counterparts $H^{en}_{\mathfrak{Indep}}$ and $H^{comm}_{\mathfrak{Indep}}$ (see the right column of Table 2). Relations Indep and $\mathfrak{Indep}$ are related by the following lemma; its proof is analogous to that of Lemma 1.

Lemma 3. *Let s, s' be statements, and let τ, τ' be operations such that $\mathsf{Stat}(\tau) = s$ and $\mathsf{Stat}(\tau') = s'$. If $\mathfrak{Indep}(s, s')$, then $\mathsf{Indep}(\tau, \tau')$.*

We use $\mathsf{Dep}(\tau, \tau')$ (and $\mathfrak{Dep}(s, s')$) as a shorthand for "$\mathsf{Indep}(\tau, \tau')$ is false" (and "$\mathfrak{Indep}(s, s')$ is false"), and call τ, τ' (and s, s') *dependent*.

4 State Space Reduction

In a nutshell, the idea of state space reduction is as follows. Let $\mathcal{T} = \langle M, \mu_0, \longrightarrow \rangle$ be the *complete* transition system. The aim of reduction techniques is to find a *reduced* transition system $\widehat{\mathcal{T}} = \langle \widehat{M}, \mu_0, \widehat{\longrightarrow} \rangle$ such that $\widehat{M} \subseteq M$ and $\widehat{\longrightarrow} \subseteq \longrightarrow$. The idea is that $\widehat{M}$ and $\widehat{\longrightarrow}$ may be *significantly smaller* than M and $\longrightarrow$, and that investigating $\widehat{\mathcal{T}}$ will require less resources (time and memory) than inspection of $\mathcal{T}$ would. To ensure that model checking $\widehat{\mathcal{T}}$ for φ yields the same results as model checking $\mathcal{T}$, henceforth referred to as *correctness*, $\widehat{\mathcal{T}}$ should be both SOUND and COMPLETE with respect to $\mathcal{T}$ and φ [7]. Let $\boldsymbol{\Pi}$ be the set of computations in $\mathcal{T}$, and let $\widehat{\boldsymbol{\Pi}}$ be the set of computations in $\widehat{\mathcal{T}}$. Then:

SOUND — If $\boldsymbol{\pi} \in \boldsymbol{\Pi}$ s.t. $\boldsymbol{\pi} \models \neg\varphi$, then there exists a $\widehat{\boldsymbol{\pi}} \in \widehat{\boldsymbol{\Pi}}$ s.t. $\widehat{\boldsymbol{\pi}} \models \neg\varphi$.
COMPLETE — If $\widehat{\boldsymbol{\pi}} \in \widehat{\boldsymbol{\Pi}}$ s.t. $\widehat{\boldsymbol{\pi}} \models \neg\varphi$, then there exists a $\boldsymbol{\pi} \in \boldsymbol{\Pi}$ s.t. $\boldsymbol{\pi} \models \neg\varphi$.

In the remainder, we describe and define two reduction techniques, PBS and POR, in terms of the relations given in Sect. 3.3. We stress that these techniques by themselves and the ideas behind them are not new: both have extensively been studied in the context of imperative languages. Their coherent definition for agents in terms of the same relations, however, is a contribution of ours. This requires the following efforts. With respect to PBS, we redefine data structures used in traditional PBS in terms of relations given in Sect. 3.3. With respect to POR, we can straightforwardly apply the existing *ample set method*, which is already defined in terms of relations similar to those of Sect. 3.3; a novelty, however, is the introduction of a heuristic that generalises SPIN's [4].

4.1 Property-Based Slicing

The aim of *property-based slicing* (PBS) is to remove statements from the source code of the system to be verified that do not *influence* the (negated) property $\neg\varphi$. Removal of such statements may cause certain states and transitions to be eliminated from the transition system, thus yielding a reduction. A PBS

algorithm is run *before* generation of the transition system commences (and *without* the need for generation of the complete transition system). The challenge of PBS is to remove as much code as possible while preserving correctness.

PBS algorithms represent the source code of the system under verification as a graph [15]. Such a graph makes explicit how execution of one statement can influence the execution of other statements as well as the property to be checked. Moreover, it enables the formulation of the PBS problem as a graph reachability problem. In our PBS algorithm, we use *influence graphs*. Informally, the influence graph with respect to a set of statements S (by which some program P is defined) and a (negated) property $\neg\varphi$ is a graph whose vertices are statements and $\neg\varphi$, and whose edges are elements of the visibility and enables relation.

Definition 3. *Let S be the set of statements by which some program P is defined, and let $\neg\varphi$ be a negated property. The influence graph $\mathcal{G}(S, \neg\varphi) = \langle \mathcal{N}, \mathcal{E} \rangle$ is a digraph with $\mathcal{N} = S \cup \{\neg\varphi\}$ and $\mathcal{E} = \{ \langle s, \neg\varphi \rangle \in S \times \{\neg\varphi\} \mid \mathfrak{Vis}(s, \neg\varphi) \} \cup \{ \langle s, s' \rangle \in S \times S \mid \mathfrak{Enables}(s, s') \}$.*

The first line of the definition of $\mathcal{E}$ represents the notion of *direct influence* on $\neg\varphi$: every edge $\langle s, \neg\varphi \rangle$ indicates that there exists an operation $\tau \in \mathsf{Ops}(s)$ that can influence the truth value of a proposition in $\neg\varphi$. The second line of $\mathcal{E}$'s definition represents the notion of *indirect influence* on $\neg\varphi$: every edge $\langle s, s' \rangle$ indicates that there exist operations $\tau \in \mathsf{Ops}(s)$ and $\tau' \in \mathsf{Ops}(s')$ such that τ can enable τ'. If s' influences $\neg\varphi$ (directly or indirectly), s influences $\neg\varphi$ indirectly.

Closely related to influence is the notion of *routes*. A route through an influence graph is a finite sequence of vertices $s_0 \cdots s_n \neg\varphi$, abbreviated $s_0 \rightsquigarrow \neg\varphi$, such that every statement occurs only once on a route, i.e. if $i \neq j$ then $s_i \neq s_j$ for all $0 \leq i, j \leq n$, and every route ends in $\neg\varphi$. The set of all routes through an influence graph $\mathcal{G}(S, \neg\varphi)$ is denoted by $\mathit{Routes}(\mathcal{G}(S, \neg\varphi))$. The idea central to our PBS algorithm is that every statement that is *not* on any route through the influence graph $\mathcal{G}(S, \neg\varphi)$ can safely be removed from the source code: these statements have no influence on the truth value of $\neg\varphi$. The algorithm takes a set of statements S as input, and computes a reduced set of statements $\widehat{S}$ by constructing an influence graph and computing routes. To determine if a route exists, the algorithm starts at a vertex s, and explores the influence graph until the vertex $\neg\varphi$ is reached, or no more reachable yet unexplored vertices are left.[7]

Existing PBS algorithms work in roughly the same way: the program is represented as a graph, reducing the PBS problem to graph reachability analysis. A key difference is that in our approach, the connection between operations and statements is made very explicit,[8] allowing for a rigid proof of correctness. We have not found similar explicit connections in the existing literature on PBS.

[7] Several optimisations may be implemented. For instance, if a depth-first exploration strategy is applied, all vertices on the depth-first stack at the moment $\neg\varphi$ is reached also have a route to $\neg\varphi$, making additional searches for these statements unnecessary.

[8] The visibility and enables relations ($\mathfrak{Vis}$ and $\mathfrak{Enables}$) are defined in terms of read and write sets on statements ($\mathfrak{Read}$ and $\mathfrak{Write}$), which are related to read and write sets on operations (Read and Write) by Property 1, which are defined in terms of individual transitions of the transition system.

Theorem 1. *Our PBS algorithm preserves soundness and completeness.*

Proof (Sketch). We adopt the premise that if a computation $\boldsymbol{\pi}$ satisfies $\neg\varphi$, i.e. $\boldsymbol{\pi} \models \neg\varphi$, then an operation that influences $\neg\varphi$ is applied during $\boldsymbol{\pi}$'s generation.

SOUNDNESS *If $\boldsymbol{\pi} \models \neg\varphi$ and by our premise, there exists a computation $\boldsymbol{\pi}'$ such that $\boldsymbol{\pi}' \models \neg\varphi$ and that is generated exclusively by applying influential operations. Hence, as the algorithm retains all statements that can induce influential operations, $\boldsymbol{\pi}'$ is also a computation in the reduced transition system.*

COMPLETENESS *Because the algorithm does not introduce new statements to the set S, no new transitions are introduced either.* □

We note that the adopted premise in the previous proof is *false* if $\neg\varphi$ (in NNF) contains $\bigcirc$ or $\mathcal{R}$ operators: $\bigcirc\,\phi$ can be true without application of an influential operation if ϕ is already true in the current state, while $\phi\,\mathcal{R}\,\phi'$ can be true if ϕ' is true from the current state onwards without an influential operation ever being applied (i.e. ϕ never becomes true). Thus, the PBS algorithm is only applicable if $\neg\varphi$ is in the $\{\bigcirc, \mathcal{R}\}$-free fragment of LTL.

4.2 Partial Order Reduction

Next, we present a *partial order reduction* (POR) algorithm in terms of the relations of Sect. 3.3. POR algorithms try to exploit the observation that the various orders in which certain events can take place are irrelevant with respect to a certain property. Once such a situation is identified, a POR algorithm forces the model checker to choose only one *representative* order and to disregard all the others. While a PBS algorithm is applied prior to the generation of the reduced transition system, a POR algorithm is run *during* its generation (and *without* the need for generation of the complete transition system first).

There are various approaches to POR. Here, we focus on the *ample set method* [4] as it fits the relations of Sect. 3.3 seamlessly. The idea is to construct a reduced transition system by selecting only a subset of all enabled operations in each state (and disregarding the other enabled operations). To preserve correctness, such a subset, called an *ample set* and denoted by $\mathsf{Ample}(\mu)$, must satisfy the following:

C0 (Emptiness) $\mathsf{Ample}(\mu) = \emptyset$ iff $\mathsf{En}(\mu) = \emptyset$.
C1 (Ample Decomposition) In the complete transition system, on any path starting from some state μ, an operation dependent on an operation from $\mathsf{Ample}(\mu)$ cannot appear before some operation from $\mathsf{Ample}(\mu)$ is executed.
C2 (Invisibility) If $\mathsf{En}(\mu) \neq \mathsf{Ample}(\mu)$, all operations in $\mathsf{Ample}(\mu)$ are visible.
C3 (Cycle Closing) If a cycle contains a state in which an operation τ is enabled, then it also contains a state μ such that $\tau \in \mathsf{Ample}(\mu)$.

Details about these conditions are given in [4].

Let μ be a state. A naive implementation of the ample set method would be to check for all subsets of $\mathsf{En}(\mu)$ whether the four conditions are satisfied, and then pick one such subset as ample set. The problem with such an implementation, however, is that checking **C1** is computationally just as hard as the model

checking problem for the complete transition system [4]. Therefore, in practice, rather than checking **C1** for an arbitrary subset of enabled operations, a heuristic approach that finds a set of operations that is *guaranteed* to satisfy **C1** is used. We call such a set a *candidate set*. Such an approach does not always lead to an ample set that yields the greatest reduction possible, but can be effective nevertheless. Once candidate sets are chosen, they need only be checked for **C0**, **C2**, and **C3**, which are easy to compute. Our idea for choosing candidate sets is to first select a subset of S, denoted by $\widehat{S}$, which satisfies the following:

Property 2. Let S be the set of statements defining a program. Then, for all $s' \in \widehat{S}$, there does not exist a $s \in S \setminus \widehat{S}$ s.t. (i) $\mathfrak{Dep}(s, s')$ and (ii) $\mathfrak{Enables}(s, s')$.

Once a set $\widehat{S}$ satisfying Property 2 is found, the set of all enabled operations in a state μ that can be induced by a statement $s \in \widehat{S}$ is selected as a candidate set C, i.e. $C = \mathsf{En}(\mu) \cap \bigcup_{s \in \widehat{S}} \mathsf{Ops}(s)$. It is guaranteed that C satisfies **C1**.

Lemma 4. *If $\widehat{S}$ satisfies Property 2, $C = \mathsf{En}(\mu) \cap \bigcup_{s \in \widehat{S}} \mathsf{Ops}(s)$ satisfies* **C1**.

Proof (Sketch). There are two situations in which **C1** *may be violated, which differ by whether τ is induced by a statement s' outside $\widehat{S}$ or in it. In the former case, if $s' \notin \widehat{S}$, there exists a statement in $\widehat{S}$ on which s' depends (because τ is dependent on an operation in C). This situation is covered by condition (i) of Property 2. In the latter case, if $s' \in \widehat{S}$, then τ is not enabled in the current state (because $\tau \notin C$). Hence, there exists another statement s that enables s'. If $s \notin \widehat{S}$, then $\mathfrak{Enables}(s, s')$, hence this situation is covered by condition (ii) of Property 2. Otherwise, if $s \in \widehat{S}$, the previous argument can be applied inductively.* □

In practice, the challenge is finding suitable sets $\widehat{S}$ as efficiently as possible. A straightforward approach is iterating over all elements in the power set of S, and checking Property 2 for each $\widehat{S} \in 2^S$. However, as this requires time exponential in the number of statements, this is not a good idea. Instead, we let the search for sets $\widehat{S}$ be guided by the definition of $\mathfrak{Dep}$: we search for sets $\widehat{S}$ that are guaranteed to satisfy (i) of Property 2. This search can be done in time linear in the number of statements $|S|$ and the size of $\mathfrak{Dep}$, and yields at most $|S|$ sets $\widehat{S}$ instead of $2^{|S|}$ for which (ii) of Property 2 need be checked. The idea is to regard the relation $\mathfrak{Dep}$ as a graph whose vertices are statements and whose edges are elements of the relation. Because every edge is an element of $\mathfrak{Dep}$, each statement belonging to a set $\widehat{S}$ cannot have edges to statements outside $\widehat{S}$: a set $\widehat{S}$ satisfying (i) of Property 2 corresponds to a *connected component* in the graph, which can be found with a depth-first search [9]. Such a search runs in time linear in the number of vertices and edges. As there cannot be more connected components than vertices, this approach yields at most $|S|$ sets $\widehat{S}$. The previous comprises the key difference with SPIN's POR implementation: in SPIN, sets $\widehat{S}$ satisfying Property 2 are always singletons. We have generalised this with an approach that reduces the problem to finding connected components. Note that our approach's applicability is not limited to agents, but extends to, for instance, SPIN as well.

The POR algorithm is run each time successors of a state μ are required during model checking. It first computes sets of operations satisfying **C1** as

outlined above and then performs simple checks for **C0**, **C2** (using $\mathfrak{Vis}$), and **C3**. If no set satisfying all conditions can be found, all successors in μ are returned. Like all POR algorithms, the algorithm described is applicable only if the property under investigation is in the *stuttering invariant* subset of LTL: it may not contain $\bigcirc$ operators. Also, it is compatible with on-the-fly model checking, provided the remarks made in [8] are taken into account.

Theorem 2. *Our POR algorithm preserves soundness and completeness.*

Proof (Sketch). The algorithm is, essentially, the algorithm in [4] with a different approach to generating **C1**. *Soundness and completeness thus follow from the ample set method's correctness as proven in Sect. 10.6 of [4] and Lemma 4.* □

5 Implementation & Experience

We have implemented the algorithms discussed in the previous section as extensions to the *interpreter-based* GOAL model checker introduced in [11]. The idea of the interpreter-based approach to agent verification is to implement model checking algorithms on top of an existing agent interpreter. An alternative approach is to encode the semantics of the agent language in a format that an existing model checker can process and to use this existing model checker for actual verification. Interpreter-based model checking, however, has been shown to consume less resources and offers immediate language support without the need for complex translations [11].

With respect to the implementation of reduction techniques, the interpreter-based approach has another benefit: the model checking algorithms implemented on top of the existing agent interpreter can easily be extended with implementations of reduction algorithms. In contrast, if existing model checkers are used for agent verification, such extensions are likely to be less straightforward to implement. As a result, one is bound to use reduction techniques that ship with the existing model checker, but that are not tailored to the agent language that the agent program is written in. It has been shown [2] that generic reduction algorithms may not work well on translated agent programs.

The PBS and POR algorithm discussed are defined in terms of the same relations on operations. From a software engineering point of view, the implementation of these techniques benefits from this in two ways: SHARED-CODE-BASE and RUNTIME-SYNERGY.[9]

SHARED-CODE-BASE — We implemented a library for analysis of action rules and computation of the visibility, enables, and (in)dependence relation. The implementations of the PBS and POR algorithms both use this library.

RUNTIME-SYNERGY — Computation of the visibility, enables, and dependence relation occurs at most once each verification run. Subsequently, the PBS and POR implementations can both use the results of these computations; no duplicate calculations are performed.

[9] Note we address the recommendation of [14] that research in state space reduction should not only focus on new techniques, but also on combining existing ones.

To investigate whether our PBS and POR algorithms are able to significantly reduce resource consumption, we have carried out several small experiments involving non-deterministic single-agent systems. In what we call the *blender experiments*, we have investigated an agent whose task is to put bananas and oranges into a blender to make juice. In the *blocks counter experiments*, the subject of verification is an agent that breaks down towers of blocks, while counting to some natural number. Finally, in the *wumpus experiments*, we have model checked agents that must navigate through an unknown maze in search of a heap of gold, while avoiding bottomless pits and a vicious cave animal: the wumpus. With these experiments, we aim at investigating whether PBS and POR algorithms for agent languages like GOAL have the same potential as in traditional model checking. Below, we give a synopsis; details appear in [10].

With respect to PBS, the blender and blocks counter experiments show that the reduction can be significant: the measured decrease of the state space ranged from 75% to 99%, the reduction in runtime (including PBS computation) ranged from 43% to 97%, and the measured reduction in memory consumption (including PBS computation) ranged from 25% to 88%. However, in the wumpus experiment, a reduction in resource consumption was not achieved: in fact, the entire verification procedure took longer to finish with PBS enabled than without PBS, although the difference was less than three seconds for the most complex wumpus agent. The reason is that a wumpus agent's tasks (exploring the cave, grabbing the gold, hunting the wumpus) all influence each other, i.e. all action rules are on a route in the influence graph. Consequently, no action rules are removed by the PBS algorithm, hence no reduction is obtained, despite the spending of resources on its computation. A prerequisite for the PBS algorithm to yield a reduction is, thus, that the property under investigation concerns a task of the agent that is not influenced by its other tasks. This prerequisite is satisfied by the agents in the other two experiments: putting bananas in a blender does not influence putting oranges in a blender (and vice versa), and deconstructing a tower does not influence counting (and vice versa).

Similar to the PBS results, our blender and blocks counter experiments with POR show that this technique can yield significant reductions, particularly if the agent under consideration is (i) *loosely coupled*, meaning that there are few dependencies between the different tasks that it needs to carry out (the case in the blocks counter experiments),[10] or (ii) significantly underspecified (the case in the blender experiments). While the former has already been pointed out in existing POR literature, the latter seems specific to the application of POR to agents, as underspecification in imperative languages is rare. Using POR, the measured reduction of the state space ranged from 59% to over 99%, the reduction in runtime (including POR computation) ranged from 34% to 98%, and the measured reduction in memory consumption (including POR computation) ranged from 8% to 50%. As the agents in the wumpus experiments are neither loosely coupled

[10] This is a stronger requirement than the PBS prerequisite regarding influence, because influence is a directed relation (e.g. A can influence B, while B does not influence A), while dependence is undirected (e.g. A depends on B iff B depends on A).

nor underspecified, no reduction is obtained using POR. We speculate that nondeterministic agent programs are, in general, tighter coupled than concurrent imperative systems. Therefore, POR may be less often applicable in an agent context than in traditional model checking. Further investigations are, however, necessary to confirm or disprove this conjecture.

6 Related Work & Conclusion

Related work Both PBS and POR have extensively been studied in traditional model checking. An extensive survey with many references is given in [14]. Here, we focus on state space reduction techniques for agent model checking.

To the best of our knowledge, PBS has been studied in an agent context only by Bordini et al. [2, 3], who have designed a PBS algorithm for AgentSpeak systems. Their algorithm is based on earlier work on slicing logic programs [16], because *plans* in AgentSpeak are similar to guarded clauses in logic programming. The algorithm of Bordini et al. slices AgentSpeak programs by removing such plans from agents, and is, like other PBS algorithms, based on a graph representation of the program. An important difference between Bordini et al. and our work is that we have defined our PBS algorithm generically, i.e. not tailored to any specific APL. We do not, however, think of our effort as a generalisation of Bordini et al., because we have not based our PBS algorithm on [16] or [2, 3]. Instead, we see our work as a second and independent attempt to applying PBS to agents; it would be interesting to instantiate our framework for AgentSpeak, and compare the performance of the algorithm of Bordini et al. to ours.

To the best of our knowledge, POR has only been studied in an agent context by Lomuscio et al. [12]. While both our work and the work of Lomuscio et al. are based on the ample set method and applied in a context in which a depth-first strategy is used for the generation of the transition system, our approach differs in a number of ways. Most notably, [12] focuses on the verification of *models* of agent-based systems, while we consider verification of actual agent *programs*. Other work in the latter direction is the AIL framework [5] and its model checker AJPF [1]; a comparison between the aforementioned interpreter-based model checker for Goal and AJPF appears in [11].

Conclusion We have introduced a framework, based on operations on mental states of agents, that facilitates the definition and implementation of the existing PBS and POR techniques in a unifying way. We have argued that existing state space reduction algorithms do not fit agent programs seamlessly due to the different nature of mental states (compared to variable assignments), and proposed a solution. The resulting definition of read and write sets for agents is the heart of our framework. With these and the relations defined in terms of them, in principle, we can readily reuse existing reduction algorithms. Nevertheless, we have also advanced the theory of PBS and POR to some extent: with respect to PBS, we have a very explicit connection between the algorithm and the transition system (absent in previous contributions), while with respect

to POR, we have introduced an alternative heuristic to be used for ample set computation (Property 2). Finally, by defining two different techniques in terms of the same relations, we gain implementation benefits: shared code-base and runtime synergy.

We identify three directions for future work: (i) expanding our experience with both techniques to gain a better understanding of when their application can be beneficial and to what extent, (ii) instantiating the framework for multi-agent systems, and (iii) extending the framework to open systems.

References

1. R. Bordini, L. Dennis, B. Farwer, and M. Fisher. Automated verification of multi-agent programs. In *Proc. of ASE*, pages 69–78, 2008.
2. R. Bordini, M. Fisher, W. Visser, and M. Wooldridge. State-space reduction techniques in agent verification. In *Proc. of AAMAS*, pages 896–903, 2004.
3. R. Bordini, M. Fisher, M. Wooldridge, and W. Visser. Property-based slicing for agent verification. *Journal of Logic and Computation*, 19(6):1385–1425, 2009.
4. E. Clarke, O. Grumberg, and D. Peled. *Model checking*. The MIT Press, 2000.
5. L. Dennis, B. Farwer, R. Bordini, M. Fisher, and M. Wooldridge. A common semantic basis for BDI languages. In M. Dastani, A. E. F. Seghrouchni, A. Ricci, and M. Winikoff, editors, *Programming Multi-Agent Systems*, volume 4908/2008 of *LNCS*, pages 124–139. 2008.
6. K. Hindriks. Programming rational agents in Goal. In A. Seghrouchni, J. Dix, M. Dastani, and R. Bordini, editors, *Multi-Agent Programming*, chapter 4, pages 119–157. 2009.
7. G. Holzmann. *The SPIN model checker*. Addison-Wesley, September 2003.
8. G. Holzmann, D. Peled, and M. Yannakakis. On nested depth first search. In *The SPIN Verification Systems*, volume 32 of *DIMACS*, pages 23–31. 1997.
9. J. Hopcroft and R. Tarjan. Efficient algorithms for graph manipulation. Technical Report STAN-CS-71-207, Stanford University, March 1971.
10. S.-S. Jongmans. Model checking Goal agents. Master's thesis, Delft University of Technology, August 2010. Available at `http://repository.tudelft.nl`.
11. S.-S. Jongmans, K. Hindriks, and M. van Riemsdijk. Model checking agent programs by using the program interpreter. In J. Dix, J. Leite, G. Governatori, and W. Jamroga, editors, *CLIMA*, volume 6245/2010 of *LNCS*, pages 219–237. 2010.
12. A. Lomuscio, W. Penczek, and H. Qu. Partial order reductions for model checking temporal epistemic logics over interleaved multi-agent systems. *Fundamenta Informaticae*, 101(1-2):71–90, 2010.
13. A. Lomuscio and F. Raimondi. Mcmas: A model checker for multi-agent systems. In H. Hermanns and J. Palsberg, editors, *TACAS*, volume 3920/2006 of *LNCS*, pages 450–454. 2006.
14. R. Pelanek. Fighting state space explosion: review and evaluation. In D. Cofer and A. Fantechi, editors, *Formal Methods for Industrial Critical Systems*, volume 5596/2009 of *LNCS*, pages 37–52. 2009.
15. F. Tip. A survey of program slicing techniques. Technical Report CS-R9438 1994, CWI, 1994.
16. J. Zhao, J. Cheng, and K. Ushjima. Literal dependence net and its use in concurrent logic programming environment. In *Proc. of the Workshop on Parallel Logic Programming*, pages 127–141, 1994.

Index of Authors