Model Checking Agent Programming Languages*

Louise A. Dennis[α] Michael Fisher[α] Matthew P. Webster[β] Rafael H. Bordini[γ]

α : Department of Computer Science, University of Liverpool, Liverpool L69 3BX, United Kingdom.
l.a.dennis@liverpool.ac.uk, mfisher@liverpool.ac.uk

β : Virtual Engineering Centre, Daresbury Laboratory, Warrington WA4 4AD, United Kingdom.
matt@liverpool.ac.uk

γ : Institute of Informatics, Federal University of Rio Grande do Sul,
PO Box 15064, 91501-970 Porto Alegre, RS - Brazil.
R.Bordini@inf.ufrgs.br

Published as *Automated Software Engineering Journal 19(1):3-63*, Mar. 2012

Abstract

In this paper we describe a verification system for multi-agent programs. This is the first *comprehensive* approach to the verification of programs developed using programming languages based on the BDI (belief-desire-intention) model of agency. In particular, we have developed a specific layer of abstraction, sitting between the underlying verification system and the agent programming language, that maps the semantics of agent programs into the relevant model-checking framework. Crucially, this abstraction layer is both flexible and extensible; not only can a variety of different agent programming languages be implemented and verified, but even *heterogeneous* multi-agent programs can be captured semantically. In addition to describing this layer, and the semantic mapping inherent within it, we describe how the underlying model-checker is driven and how agent properties are checked. We also present several examples showing how the system can be used. As this is the first system of its kind, it is relatively slow, so we also indicate further work that needs to be tackled to improve performance.

1 Introduction

Since the introduction of the *agent-based systems* concept in the 1980s [2, 14, 26, 19, 32, 69] the field has seen significant growth and increasing research maturity. This is true not only within academia but also for industrial applications, where the agent metaphor has been shown to be useful in capturing many practical situations, particularly those involving complex systems comprising flexible, autonomous, and distributed components [53].

The large number of agent platforms now available [3] has meant that the *industrial* uptake of this technology is continually growing. In software development, we have seen significant commercialisation of multi-agent systems technology, e.g., in the form of a Java-based ontology development and multi-agent toolkit [42]. Similarly, building effective and user-friendly transportation systems is increasingly tackled using AI methodologies and multi-agent technology in particular [53]. Other application areas have seen the emergence of agent frameworks, financed by industry, and designed to cope specifically with industrial requirements [45], and such technology has been successfully introduced in many companies (e.g., goal-oriented, autonomic process navigation [67]). An early survey of applications of agent technology in industrial systems control can be found in [48]. Other areas in which agents are actively used include air-traffic control [55], autonomous spacecraft control [58], and health care [57]. Clearly, many of these are areas for which we must demand software *dependability* and *security*.

*The authors would like to thank EPSRC for its support of this work through research projects EP/D052548, EP/DO54788, and EP/F037201, and CNPq for its support through grant 307924/2009-2.

As agent-based solutions are used in increasingly complex and critical areas, there is clearly a need to analyse, comprehensively, the behaviour of such systems. Not surprisingly, therefore, the area of formal verification techniques tailored specifically for agent-based systems is now attracting a great deal of attention [36]. While program verification is well advanced, for example Java verification using Java PathFinder [74, 50], the verification of agent-oriented programs poses new challenges that have not yet been adequately addressed, particularly within the context of practical model-checking tools. Tackling this deficiency is our long-term aim in this work.

We here consider the formal verification, via model checking, of programs written in Agent Programming Languages (APLs). As will be described below, such languages are significantly different from traditional programming languages and, hence, their verification requires extended techniques. The predominant model for agent systems is the BDI (belief-desire-intention) model [64, 63] within which agents are viewed as being *rational* and acting in accordance with their beliefs and goals. As such it is important to be able to verify properties expressed in terms of such concepts. Thus, in agent verification, we have to verify not only what the agent does, but *why* it chose that course of action, *what* it believed that made it choose to act in this way, and *what* its intentions were that led to this. Our previous work in this area has concentrated on one particular language, AgentSpeak [62], whereas we here extend to *any* agent language that is based on the BDI paradigm [63] and for which a formal operational semantics can be provided within our framework. In addition, we now go beyond previous work in verifying more complex multi-agent systems. Though the performance is still not high when a larger number of agents are considered, we show that our approach is very flexible and, with the speed improvements envisaged, can potentially provide an effective verification system for much larger multi-agent systems.

In essence, our framework consists of two components. The *Agent Infrastructure Layer* (AIL) is a set of Java classes designed to act as a toolkit for creating interpreters for BDI Agent Programming languages. This toolkit is designed to make the construction of such interpreters quick and easy once an operational semantics is provided. The second component is *Agent JPF* (AJPF), a version of the Java Pathfinder (JPF) model checker [74, 50] which has been extended with a property specification language appropriate for agent programs and some Java interfaces suitable for encapsulating multi-agent systems in an efficient fashion.

Our aim has been to define a sufficiently general intermediate layer on which model checking could be performed. AIL and AJPF are designed to work efficiently together: AIL is designed to optimise model-checking time rather than execution time and to implement the interfaces specified in AJPF[1]. We are thus able to model check the properties of any (potentially heterogeneous) multi-agent system implemented in languages that have AIL-based interpreters. By optimising the AIL's state representation for use with AJPF, our customised version of the JPF model checker, we avoid the intricacies of carrying out such optimisation for each language separately, thereby reducing the probability of introducing errors at this stage.

The work described in this paper builds on results from a number of workshop [28, 76] and conference [5] papers. It develops and updates these results and includes unpublished applications [76] developed to assess the practicality of the software. This paper constitutes the first complete description of the final system, which is now available on Sourceforge, at `http://mcapl.sourceforge.net`.

2 Background

2.1 Rational Agents and Multi-Agent Systems

The agent abstraction captures truly *autonomous* behaviour. As we do not consider arbitrary, 'random' autonomy, then we assert that an autonomous system should have some *motivation* for acting in the way it does. This aspect is captured by the concept of a *rational agent* [13, 19, 64]. The key aspect of a rational agent is that the decisions it makes, based on dynamic motivations, should be both "reasonable" and "justifiable". In detail, a BDI agent comprises *beliefs* that the agent has about itself and its environment, *desires* (or *goals*) representing its long-term aims, and *intentions* describing the agent's immediate goals (the ones it is currently trying to achieve through acting on the environment where it is situated). Thus,

[1] Note that, if required, AIL can be used without AJPF and *vice versa*.

such an agent analyses data about itself and its environment and generates, or updates, its beliefs. The agent endeavours to tackle its long-term aims within this context, which leads to a set of immediate goals (*intentions*), which are tackled through the agent's plans (determining appropriate courses of action to achieve goals). When several possible (conflicting) goals exist, the agent must undertake some deliberation and decide which intention to realise in practice as well as reasoning about how to do so.

A multi-agent system (MAS) is a system consisting of a number of rational agents interacting with each other. Problem solving using multi-agent systems is now an established area of software engineering. The cooperation aspect helps solve problems that are difficult to solve by individual agents or traditional, centralised, computer systems. Areas in which multi-agent systems have been successful include online trading, space missions, air traffic control, disaster response, and the modelling of social structures [53, 45, 67, 57], to name just a few.

2.2 BDI Agent Programming Languages

There are *many* different agent programming languages and agent platforms based, at least in part, on the BDI approach. Particular languages developed for programming *rational* agents include AgentSpeak [62], *Jason* [9, 10], 3APL [43, 24], DRIBBLE [66], Jadex [60], GOAL [44, 1], CAN [79], SAAPL [78], GWENDOLEN [27], and METATEM [34, 35]. Rather than providing a framework in which the complex logical properties of systems using just *one* particular agent approach can be verified, we have developed a flexible and uniform framework allowing the verification of a wide range of agent-based programs, produced using several different high-level agent programming languages. Specifically, our focus has been on BDI languages, i.e., languages that generally follow the *beliefs*, *desires*, and *intentions* paradigm [63] of agent-oriented programming. Consequently, the architecture presented in this paper is based upon our study of common concepts and structures appearing in the operational semantics of various BDI programming languages [28]. Agents programmed in these languages commonly contain a set of *beliefs*, a set of *goals*, and a set of *plans*. Plans determine how an agent acts based on its beliefs and goals; the sophisticated form or plan execution is the basis for *practical reasoning* (i.e., reasoning about actions) in such agents. As a result of executing a plan, the beliefs and goals of an agent may change as the agent performs actions in its environment.

In [28], we tackled issues in the treatment of beliefs, goals, plans as well as events, intentions, and other components central to the design of these languages. We started by extracting the major concepts of 3APL [24] and the variant of AgentSpeak [62] encapsulated by *Jason* [10] for use in our framework. However, our approach does not exclude other languages, in some cases even those based on completely different agent architectures. It was our aim to include languages that have practical relevance. Thus, we did not want to restrict ourselves to (abstract) programming languages that could not be considered for serious software development projects.

2.3 Agent Verification

The main approaches to detailed systems analysis can be broadly categorised as *testing* [41], *model checking* [18], and *theorem proving* [59].

Testing is fast, but generally non-exhaustive, and thus cannot guarantee that a property holds throughout a given system. However, the latter two techniques are exhaustive and will, in principle, give a definite answer to the question of whether a property holds for all program executions. Theorem proving requires a deep knowledge of mathematical structures and techniques, is usually only partly automatable, and is, in practice, often very costly, since the task can usually only be carried out by experts. Model checking, on the other hand, relies on a complete, but automated, inspection of the system's state space. This makes it computationally costly (in terms of time and space), but does not usually require a specialist as in the case of theorem proving. In order to reduce the computational problems, state-of-the-art model checkers employ a number of reduction techniques that make it possible to handle even large state spaces relatively efficiently. Since we require an automatic/definite answer and are looking into practical applications of verification in the design of multi-agent programs, model checking is the most promising of the approaches.

Model checking is a technique whereby a finite description of a system is analysed with respect to a property in order to ascertain whether *all* possible executions of the system satisfy this property. Formally,

the property is typically described using a temporal logic formula [33], while the model checking process essentially involves trying to find a model within the system description for the *negation* of this property. If any such model is found then it describes a 'run' of the system that does *not* satisfy the required property [18]. In agent-based systems, it is vital to verify not only the behaviour that the system has, but also to verify *why* the agents are undertaking certain courses of action within the multi-agent system. Thus, in our approach the temporal basis of model-checking, capturing the *dynamic* nature of agent computation, is extended with *modal operators* [38] capturing the *informational* ('beliefs'), *motivational* ('desires') and *deliberative* ('intentions') aspects of rational agents.

With the increasing use of multi-agent systems and the sophistication of model-checking technology, it is not surprising that the concept of automated agent verification has attracted significant interest. Pioneering work on *model checking* techniques for the verification of agent-based systems has appeared, for example, in [6, 51, 7, 61]. Our previous work [6, 7] has concentrated on model checking techniques for agent-based systems written in the logic-based agent-programming language AgentSpeak [62]. This required a specific encoding of the AgentSpeak agent system's states in the input language used by the model checker. Applying similar techniques to other agent programming languages would require manually encoding the state representation of that language. This is both tedious and error-prone. This was a key motivation for the work presented here. Our intention was to lift away the effort involved in developing a model checking framework for a given language from encoding the language directly in a model checker, moving up to the level of constructing a (model-checker backed) interpreter for the language given suitable support.

3 AIL: A Framework for Creating BDI Interpreters

The *Agent Infrastructure Layer* (AIL) is an intermediate layer that encompasses key concepts from a wide range of BDI programming languages as data structures in Java and enables the implementation of their operational semantics within a clear framework. These data structures can then be used both in creating Java interpreters for the agent programming languages and for interfacing with the underlying agent model checker.

An agent originally programmed in some agent programming language (APL) and running in an AIL-based interpreter uses the AIL data structures to store its internal state comprising, typically: a belief base, a plan library, a set of intentions, and other temporary state information. We also assume that the APL defines a *reasoning cycle* which is expressed using an operational semantics. The rules in the cycle define how the agent's practical reasoning progresses, depending on its current internal representation and the current stage of the reasoning cycle. The AIL provides support for constructing reasoning cycles along with a number of rules that commonly appear in the operational semantics of agent programming languages. The operational semantic rules within AIL are given in Appendix A. In short, the AIL toolkit collects together Java classes that:

1. facilitate the implementation of interpreters for various agent programming languages;
2. contain adaptable, clear, operational semantics; and
3. can be verified through AJPF, an extended version of the open source Java model checker JPF [74].

If an AIL-based interpreter is run in conjunction with the AJPF model checker, then the system will notify the model checker each time a new state is reached that is relevant to the verification. It is left to the designers of interpreters to decide where, in the reasoning cycle, such points should fall.

As a natural consequence of this, AIL also makes it easier to develop an interpreter for a programming language using the AIL classes than it would be to build an interpreter "from scratch" in Java. Fig.1 provides a diagrammatic representation of AIL within the AJPF model checking architecture.

AJPF is our extension of the JPF model checker [74, 50], which includes interfaces linking AIL interpreters to the model checking framework and a property specification language. These interfaces also allow programming languages that do *not* have their own AIL-based interpreters to be model checked

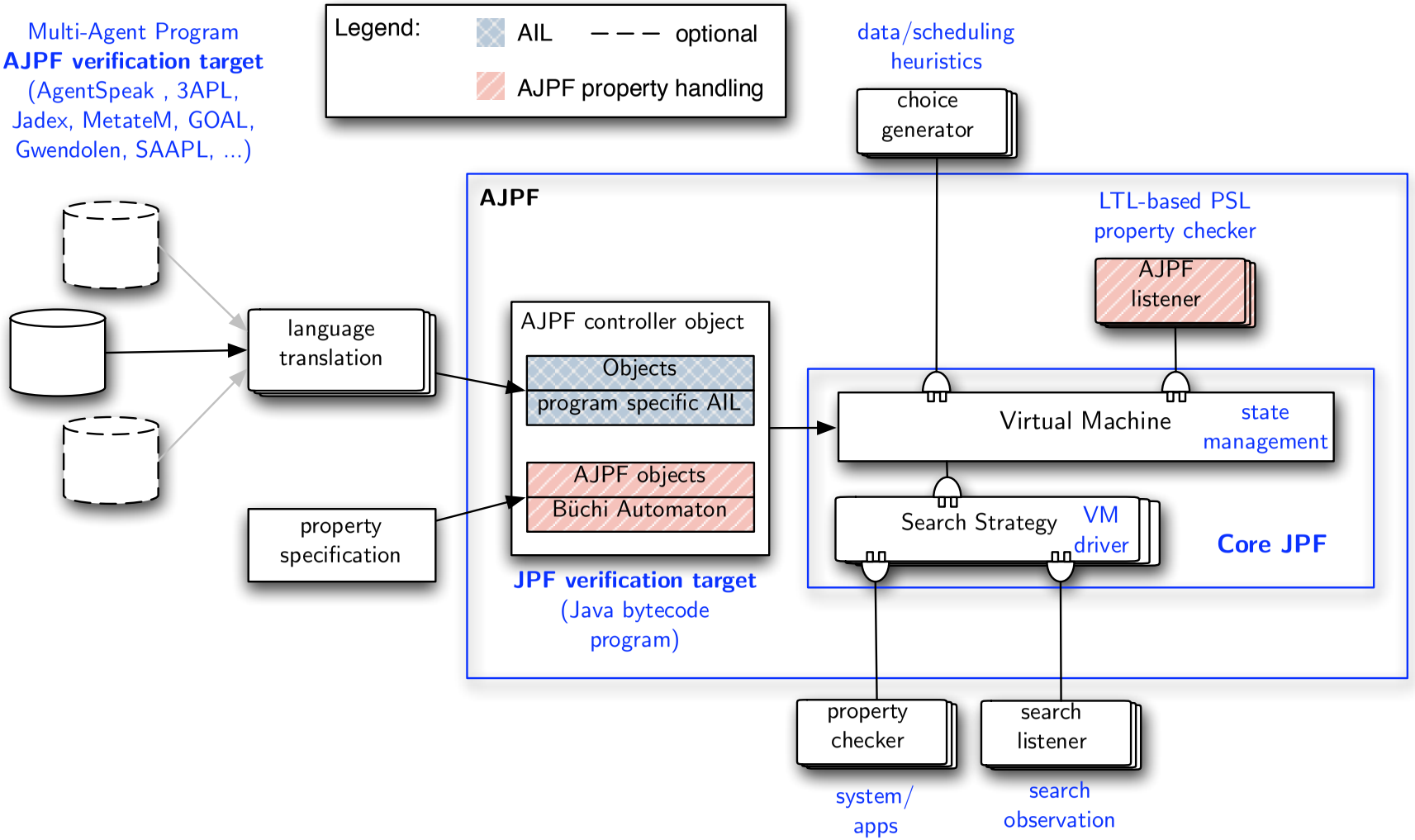


Figure 1: AJPF architecture [5].

against specifications written in the same property specification language, using AJPF. However these languages will not benefit from the efficiency improvements that the optimised AIL classes can provide. AJPF is discussed in more detail in Section 4.2.

Fig. 1 shows how the combination of the translated agent program(s) together with the property specification constitutes an AJPF controller object. This controller is used as the JPF verification target. The original program(s) with the original property specification can be viewed as the AJPF verification target; they are fed into the appropriate translators available as part of our framework.

3.1 Components of Agent Programming Languages

We will now proceed to give an overview of the AIL structures available.

3.1.1 Agents

Agents are specified by a large class with multiple fields. This class is intended for sub-classing or wrapping by language-specific interpreters which need only refer to the fields that are of interest to them. We discuss some of the key components of this class below. It also contains components for exploring organisational and group structures as outlined in [30], though we do not discuss these here.

When we have discussed many of the key data structures within an agent, we will provide a formal definition of an AIL agent in section 3.1.12.

3.1.2 Beliefs

Every agent contains a *belief base* of statements it believes to be true. This can be viewed as a set of first order formulae. AIL also supports Prolog-style reasoning based on a *rule base* of Horn clauses.

3.1.3 Guards and Logical Consequence

AIL uses the concept of a *guard* on a plan (as well as on other constructs). In the case of a plan, the guard is used to determine the applicability of the plan. In general, guards are used whenever an agent needs to check its own state.

A guard consists of two sets of expressions. These sets represent propositions the agent must hold (positive states) — i.e., statements it believes, goals it possesses etc. — and propositions the agent must not hold (negative states)[2]. A guard represents the conjunction of these expressions which can be checked for validity against the agent's internal state (so typically such expressions refer to things the agents believes or goals it possesses). AIL provides an interface for the expressions that may appear in guards and requires these expressions to implement a *logical consequence* relation (we will write $ag \models g$ to represent that the guard, g, is a logical consequence of the internal state of agent, ag). Procedurally speaking, the implementation of logical consequence provides a decision procedure for deciding whether a guard holds in an agent's current state.

AIL provides a default implementation of logical consequence which provides an algorithm for deciding whether some statement is believed by the agent, is a goal of the agent, has been sent as a message by the agent, as well as common propositional logical statements constructed from such atomic formulae (e.g. conjuction, disjunction, and negation).

However, since different languages may have different semantics for belief, goal, etc., this procedure can be overridden so, for instance, a different semantics can be supplied for what it means for a formula to be a goal of the agent. It is also possible, by this means, for individual languages to provide custom data structures for additional parts of an agent's state that those languages need to appear in guards.

The default implementation of logical consequence supports reasoning using Prolog-style Horn clauses where the literals appearing in such clauses may, themselves, be guards. This allows such rules to be used in languages with varying semantics for the literals that appear in such rules, without each language having to implement it's own version of Prolog-style reasoning.

[2]The system distinguishes between strong and weak negation, i.e. the difference between not believing something is the case and believing something is not the case.

The use of the logical consequence method for reasoning about an agent's internal state is a key aspect of the AIL system and it provides the bridge into the agent state for the atomic properties of the AJPF property specification language. Then, if such properties refer, for instance, to the fact that some agent believes something, the semantics for this is determined by the relevant agent's implementation of the logical consequence relation.

3.1.4 Goals

Goals are represented by a first order literal with a particular type — in the AIL toolkit the types are *achieve*, *perform*, *test* and *maintain* (following [24]). There are rules in the toolkit distinguishing between the first three types — the fourth is currently unsupported. There is no reason why further goal types should not be added.

Agents do not maintain an explicit list of goals. Instead they deduce their goals from the "commit to goal" events that are found in their set of intentions. However, the 'Agent' class provides a number of methods that allow the agent to be treated as if it *did* maintain an explicit set of goals. In reality these methods inspect the intention data structure.

3.1.5 Intentions

AIL's most complex data structure is that which represents an *intention*. BDI languages use intentions to store the *intended means* for achieving goals – this is generally represented as some from of *deed stack* (deeds include actions, belief updates, and the commitment to goals). Intention structures in BDI languages may also maintain information about the (sub-)goal they are intended to achieve or the event that triggered them. In AIL, we aggregate this information: an intention becomes a stack of tuples of an event, a guard, a deed, and a unifier. This AIL intention data structure is most simply viewed as a matrix structure consisting of four columns in which we record events (new perceptions, goals committed to and so forth), deeds (a plan of future actions, belief updates, goal commitments, etc.), guards (which must be true before a deed can be performed) and unifiers. These columns form an event stack, a deed stack, a guard stack, and a unifier stack. Rows associate a particular deed with the event that has caused the deed to be placed on the intention, a guard which must be believed before the deed can be executed, and a unifier. New events are associated with an empty deed, ε.

Example The following shows the full structure for a single intention to clean a room. We use a standard BDI syntax: $!g$ to indicate the goal g, and $+!g$ to indicate the commitment to achieve that goal (i.e., a new goal that g becomes true is adopted). Constants are shown starting with lower case letters, and variables with upper case letters.

event	deed	guard	unifier
`+!clean()`	`+!goto(Room)`	`dirty(Room)`	`Room = room1`
`+!clean()`	`+!vacuum(Room)`	$\top$	`Room = room1`

This intention has been triggered by a desire to clean — the commitment to the goal $clean()$ is the trigger event for both rows in the intention. An intention is processed from top to bottom so we see here that the agent first intends to commit to the goal $goto(Room)$, where *Room* is to be unified with $room1$. It will only commit to this goal if it believes the (guard) statement, $dirty(Room)$. Once it has committed to that goal it then commits to the goal $Vacuum(Room)$. In many languages the process of committing to a goal causes an expansion of the intention stack, pushing more deeds on it to be processed. So $goto(Room)$ may be expanded *before* the agent commits to vacuuming the room. In which case the above intention might become

event	deed	guard	unifier
`+!goto(Room)`	`+!planRoute(Room, Route)`	$\top$	`Room = room1`
`+!goto(Room)`	`+!follow(Route)`	$\top$	`Room = room1`
`+!goto(Room)`	`+!enter(Room)`	$\top$	`Room = room1`
`+!clean()`	`+!vacuum(Room)`	$\top$	`Room = room1`

At any moment, we assume there is a *current intention* which is the one being processed at that time. The function $\mathscr{S}_{\text{int}}$ (implemented as a method in AIL) may be used to select an intention. By default, this chooses the first intention from a queue, but this choice may be overridden for specific languages and applications. Intentions can be *suspended* which allows further heuristic control. A suspended intention is, by default, *not* selected by $\mathscr{S}_{\text{int}}$. Typically an intention will remain suspended until some trigger condition occurs, such as a message being received. Many operational semantic rules (such as those involved with perception) resume *all* intentions — this allows suspension conditions to be re-checked.

3.1.6 Events

Events are things that occur within the system to which an agent may wish to react. Typically we think of these as changes in beliefs or the new commitment to goals. In many (though not all) programming languages, events trigger plans (i.e., a plan might selected for execution only when the corresponding event has taken place).

In AIL there is a special event, '`start`', that is used to start off an intention which is not triggered by anything specific. This is mainly used for the initial goals of an agent — the intention begins as a `start` intention with the deed to commit to a goal. In some languages the belief changes caused by perception are also treated in this way. Rather than being added directly to the belief base, in AIL such beliefs are assigned to intentions with the event `start` and then added to the belief base when the intention is actually executed.

3.1.7 Deeds

Deeds appear in the bodies of plans and as stacks in intentions, and represent things the agent is planning to do but has not yet actually done. Deeds include:

Beliefs — an agent may both plan to add or to drop a belief.

Goals — an agent may both plan to commit to a goal (a sub-goal) or to drop a goal.

Actions — an agent may plan to perform an action.

No Plan Yet — the "no plan yet" deed is used when an intention contains an event that has yet to be planned for; we represent the "no plan yet" deed with the distinguished symbol, ε.

Lock/Unlock — a deed can lock or unlock an intention; the idea here is to allow an operational semantics to, under some conditions, force an intention to remain current until it is unlocked.

Waiting — the *wait for* deed allows intentions to be suspended until a particular guard is satisfied in the agent's state. We represent "waiting for guard g", as '$*g$'. The idea is to allow an operational semantics to remove an intention from consideration by $\mathscr{S}_{\text{int}}$ until some condition is met.

3.1.8 Plan Library

The Agent class also contains a plan library. Plans are matched against intentions and/or the agent's state and manipulate existing (or create new) intentions. There are four main components to a plan, as follows.

1. A *trigger event* which may match the top event of an intention.
2. A *prefix* which may match the top of an intention's deed stack.
3. A *guard stack*: the top guard is checked against the agent's state for plan applicability. The rest of the stack is paired off against the rows in the body and may provide additional conditions for that row's execution.
4. A *body* which is the new deed stack that the plan proposes for execution.

Reactive plans do not have trigger events but instead react to the current state (e.g. the beliefs and goals) of the agent. By convention, within the AIL these have a variable representing an *achieve* goal as a placeholder for a trigger but this is not used by plan selection which focuses simply on checking whether their guard follows from the agent state.

Example Recall our previous example intention:

event	deed	guard	unifier
`+!clean()`	`+!goto(Room)`	`dirty(Room)`	`Room = room1`
`+!clean()`	`+!vacuum(Room)`	$\top$	`Room = room1`

A plan that matches this intention is

trigger	`+!clean()`
prefix	`+!goto(Room)` `+!vacuum(Room)`
guard	$\neg$ `dirty(Room)`
body	`+!find_dirty_room(Room2)`

So if the current intention was triggered by goal !*clean*() (the trigger event), and it currently intends to go to a room and vacuum it (prefix), *but* that room is not dirty (guard), it proposes, instead, to replace that part of the intention with the goal of locating a dirty room. If this plan was applied, the intention would become:

event	deed	guard	unifier
`+!clean()`	`+!find_dirty_room(Room2)`	$\neg$ `dirty(Room)`	Room = room1

It is more common for plans to match only intentions which contain unplanned goals (i.e., those associated with the "no plan yet" deed, ε). For instance after a commitment to `goto(Room)` the above intention might appear as:

event	deed	guard	unifier
`+!goto(Room)`	ε	$\top$	`Room = room1`
`+!clean()`	`+!goto(Room)`	`dirty(Room)`	`Room = room1`
`+!clean()`	`+!vacuum(Room)`	$\top$	`Room = room1`

which would match the plan

trigger	`+!goto(Room)`
prefix	ε
guard	`current_floor(ground)` $\wedge$ `upstairs(Room)` $\top$
body	`+!goto(stairs)` `+!goto(Room)`

This would transform the intention to:

event	deed	guard	unifier
`+!goto(Room)`	`+!goto(stairs)`	`current_floor(ground)` $\wedge$ `upstairs(Room)`	`Room = room1`
`+!goto(Room)`	`+!goto(Room)`	$\top$	`Room = room1`
`+!clean()`	`+!goto(Room)`	`dirty(Room)`	`Room = room1`
`+!clean()`	`+!vacuum(Room)`	$\top$	`Room = room1`

Applicable Plans Applicable Plans represent an interim data structure that describes how a plan from an agent's plan library changes the current intentions. Essentially an applicable plan states how many rows are to be dropped from the intention and what new rows are to be added. The new rows are generated from an event, a guard stack, a unifier and a stack of deeds. The guard and deed stacks are the same size. The new intention rows are generated by creating a row for each deed and guard on the two stacks and associating the event and unifier with each of those rows (so the event and unifier are duplicated several times).

Therefore, an applicable plan is a tuple, $\langle p_e, p_g, p_{ds}, p_\theta, n\rangle$, of an event p_e, a guard stack p_g, a deed stack p_{ds}, a unifier p_θ and the number of rows to be dropped, n. The applicable plan in the first example above would be

$$\langle +!clean(), [\neg dirty(Room)], [+!find\_dirty\_room(Room2)], \{Room = room1\}, 2\rangle \tag{1}$$

or in the previous tabular presentation we could represent this as:

Drop 2 Lines

replace with

event	deed	guard	unifier
`+!clean()`	`+!find_dirty_room(Room2)`	`¬ dirty(Room)`	`Room = room1`

The applicable plan for the second example would be

$$\begin{aligned}\langle +!goto(Room), [current\_floor(ground) \wedge upstairs(Room); \top],\\ [+!goto(stairs); +!goto(Room)], \{Room = room1\}, 1\rangle\end{aligned} \tag{2}$$

or

Drop 1 Line

replace with

event	deed	guard	unifier
`+!goto(Room)`	`+!goto(stairs)`	`current_floor(ground)` $\wedge$ `upstairs(Room)`	`Room = room1`
`+!goto(Room)`	`+!goto(Roo)`	$\top$	`Room = room1`

Applicable plans are used because many APL reasoning cycles first go through a phase where they determine a list of applicable plans and then move to a phase where they pick one plan to be applied. The function $\mathscr{S}_{\text{plan}}$ (implemented as a Java method in AIL) is used to select *one* applicable plan from a set. By default, this treats the set as a list and picks the first plan, but it may be overridden by specific languages and applications.

Applicable Plan Generation Method For the purposes of this discussion, we will write plans using the syntax $p_e : p_p : p_{gu} : p_d$ where p_e is the trigger event, p_p is the prefix, p_{gu} is the guard, and p_d is the body.

AIL provides a default function, **appPlans**, for the generation of applicable plans from the current intention and an agent's internal state. This function creates two sets of applicable plans. The first set is, essentially, the applicable plans for continuing to process intention i without any changes:

$$\{\langle \mathtt{hd}_e(i), \mathtt{hd}_g(i), \mathtt{hd}_d(i), \theta \cup \theta^{\mathtt{hd}(i)}, 1\rangle \quad | \quad (ag \models \mathtt{hd}_g\theta^{\mathtt{hd}(i)}, \theta) \wedge (\mathtt{hd}_d(i)\theta\theta^{\mathtt{hd}(i)} \neq \varepsilon)\} \tag{3}$$

Here, $\mathtt{hd}_e(i)$ is the top event in i, $\mathtt{hd}_g(i)$ is the top guard, $\mathtt{hd}_d(i)$ is the top deed, and $\theta^{\mathtt{hd}(i)}$ is the top unifier. The notation $ag \models g, \theta$ means that the guard, g, is satisfied by agent ag given unifier θ. The notation $t\theta$ indicates the application of unifier θ to term t. So, for instance, $\mathtt{hd}_d(i)\theta\theta^{\mathtt{hd}(i)}$ is the result of applying the unifiers θ and $\theta^{\mathtt{hd}(i)}$ to the top deed on the intention.

Note that the above generates an empty set if the intention's top deed is the "no plan yet" deed, ε.

The second set is

$$\begin{aligned}\{\langle p_e, p_{gu}, p_d, \theta^{\mathtt{hd}(i)} \cup \theta, \#p_p\rangle | \quad & p_e : p_p : p_{gu} : p_d \in P \wedge\\ & \#p_p > 0 \rightarrow (\mathtt{hd}_e(i) \models p_e \wedge unifier(p_p, i) = \theta_e) \wedge ag \models \mathtt{hd}(p_{gu})\theta_e, \theta\}\end{aligned}$$

Here, $\mathtt{hd}_e(i) \models p_e$ means that the plan's trigger event follows from the top event on the current intention. This allows for Prolog-style reasoning on plan triggers and is a version of the logical consequence method. Function $unifier(p_p, i)$ generates the unifier of the plan prefix with the top n rows of intention i where $n = \#p_p$ (the size of the stack of the plan's prefix).

This general mechanism for deriving applicable plans has proved sufficient for all the APLs implemented in the AIL to date.

3.1.9 Actions

Actions are the means by which an agent affects the external world, and are normally represented as first order terms. In general, when an agent encounters an action, it will cause the execution of "native" code — i.e., code typically developed with traditional programming paradigms rather than code developed for rational behaviour in autonomous systems — to take place in the environment. For instance an action, in a robot, to pick up an object is likely to execute detailed control system code to handle the actual task of picking something up, and this is normally programmed in standard programming languages for control.

3.1.10 Inbox and Outbox

Agents maintain *Inboxes* and *Outboxes* for storing messages. The default logical consequence method checks the Outbox (to check that a message has been sent) but not the Inbox — so far the operational semantics implemented have used the Inbox only for temporary storage of messages which are then processed by the agent, but in principle the method could be extended to check for received messages as well. It should be noted that storing *all* messages sent or received is potentially inefficient and languages are not required to use the inboxes and outboxes in the agent class.

3.1.11 Reasoning Cycle

Agents are assigned a "reasoning cycle" by the language in which they are written. Each stage of a language's reasoning cycle is typically formalised as a disjunction of semantic rules which define how an agent's state may change during the execution of that stage. Any rule to be used as part of an AIL agent reasoning cycle has to implement a particular Java interface. This can be achieved either by implementing the interface directly or by sub-classing an existing rule. The combined rules of the various stages of the reasoning cycle define the operational semantics of that language. The construction of an interpreter for a language involves the implementation of these rules (which in some cases may already exist in the language's original toolkit) and the implementation of a reasoning cycle, by organising the rules into (the stages of) such a cycle.

In this way, we have implemented, for example, both GOAL [1] and SAAPL (Simple Abstract Agent Programming Language) [78] interpreters, following their respective operational semantics [29] as well as a GWENDOLEN interpreter [27] [3]. The implementations of these interpreters make use of the AIL operations together with some additional classes specifically added to reproduce faithfully the semantics of those languages.

3.1.12 Formal Definition of an AIL Agent

An AIL agent can be formalised as a tuple consisting of

- The agent's name (a string), ag.
- The agent's belief base (a set of beliefs), B.
- The agent's rule base (a set of prolog-style rules for reasoning about guards), R.
- The agent's plan library, P.
- The agent's current intention (which may be empty), i.
- The agent's other intentions, I.
- The agent's currently applicable plans, **appPlans**.
- The agent's inbox of unprocessed messages, In.
- The agent's outbox of sent messages, Out.

[3] The GWENDOLEN language was developed as a side effect of some of the initial design work on the AIL.

- The current stage of the agent's reasoning cycle, S.

Operational semantic rules operate on this tuple to change the agent's state. It should be noted that individual language implementations may add further elements to this tuple as well as ignore some elements.

A multi-agent system therefore consists of a tuple of several agents and an environment (ξ). The environment has to provide certain services to the agent (the ability to access perceptions, and take actions) but has no formal semantics of its own governing how it should alter from state to state; a model of the environment is assumed to be provided by AIL users in Java.

3.2 Example: Implementing an Interpreter for an Agent Programming Language

GOAL [44, 1] is a BDI language introduced by Hindriks et al. to illustrate the use of purely declarative goals in agent programming. An agent is defined by its mental state comprising two sets of formulæ: Σ for the agent's beliefs; and Γ for the agent's goals. GOAL assumes an underlying logic on its formula language, $\mathscr{L}$, with an entailment relation $\models_C$; its semantics then defines entailment for mental states as follows:

Definition 1 *Let $\langle \Sigma, \Gamma \rangle$ be a mental state:*

$$\langle \Sigma, \Gamma \rangle \models_M \mathbf{B}\phi \quad \textit{iff} \quad \Sigma \models_C \phi,$$

$$\langle \Sigma, \Gamma \rangle \models_M \mathbf{G}\psi \quad \textit{iff} \quad \psi \in \Gamma,$$

$$\langle \Sigma, \Gamma \rangle \models_M \neg\phi \quad \textit{iff} \quad \langle \Sigma, \Gamma \rangle \not\models_M \phi,$$

$$\langle \Sigma, \Gamma \rangle \models_M \phi_1 \wedge \phi_2 \quad \textit{iff} \quad \langle \Sigma, \Gamma \rangle \models_M \phi_1 \textit{ and } \langle \Sigma, \Gamma \rangle \models_M \phi_2.$$

An agent's behaviour is governed by its *capabilities* and *conditional actions*.

Capabilities are associated with a partial function $\mathscr{T} : Bcap \times \wp(\mathscr{L}) \to \wp(\mathscr{L})$, which operates on the belief base Σ in order to alter it. Capabilities may be *enabled* or not for an agent in a particular configuration. If the capability is *not* enabled then $\mathscr{T}$ is undefined. $\mathscr{T}$ is used by the mental state transformation function $\mathscr{M}$ to alter the agent state as follows:

Definition 2 *Let $\langle \Sigma, \Gamma \rangle$ be a mental state, and $\mathscr{T}$ be a partial function that associates belief updates with agent capabilities. Then the partial function $\mathscr{M}$ is defined by:*

$$\mathscr{M}(\mathbf{a}, \langle \Sigma, \Gamma \rangle) = \begin{cases} \langle \mathscr{T}(\mathbf{a}, \Sigma), \\ \quad \Gamma \backslash \{\psi \in \Gamma \mid \mathscr{T}(a, \Sigma) \models_C \psi\} \rangle & \textit{if } \mathscr{T}(\mathbf{a}, \Sigma) \textit{ is defined,} \\ \textit{is undefined for } \mathbf{a} \in Bcap & \textit{if } \mathscr{T}(\mathbf{a}, \Sigma) \textit{ is undefined} \end{cases} \tag{4}$$

$$\mathscr{M}(\mathbf{drop}(\phi), \langle \Sigma, \Gamma \rangle) = \langle \Sigma, \Gamma \backslash \{\psi \in \Gamma \mid \psi \models_C \phi\} \rangle \tag{5}$$

$$\mathscr{M}(\mathbf{adopt}(\phi), \langle \Sigma, \Gamma \rangle) = \begin{cases} \langle \Sigma, \\ \quad \Gamma \cup \{\phi' \mid \Sigma \not\models_C \phi', \models_C \phi \to \phi'\} \rangle & \textit{if } \not\models_C \neg\phi \textit{ and } \Sigma \not\models_C \phi \\ \textit{is undefined} & \textit{if } \Sigma \models_C \phi \textit{ or } \models_C \neg\phi \end{cases} \tag{6}$$

Lastly, an agent has a set of conditional actions, Π. Each conditional action consists of a guard, ϕ, and a capability or instruction, $\mathbf{a}$, written $\phi \rhd \mathbf{do}(\mathbf{a})$. The conditional actions together with a *commitment strategy* provide a mechanism for selecting which capability to apply next.

Definition 3 *Let $\langle \Sigma, \Gamma \rangle$ be a mental state with $b = \phi \rhd \mathbf{do}(\mathbf{a}) \in \Pi$. Then, as a rule, we have: If*

1. *the mental condition ϕ holds in $\langle \Sigma, \Gamma \rangle$, i.e. $\langle \Sigma, \Gamma \rangle \models_M \phi$, and*
2. *$\mathbf{a}$ is enabled in $\langle \Sigma, \Gamma \rangle$, i.e., $\mathscr{M}(\mathbf{a}, \langle \Sigma, \Gamma \rangle)$ is defined,*

then $\langle \Sigma, \Gamma \rangle \xrightarrow{b} \mathscr{M}(\mathbf{a}, \langle \Sigma, \Gamma \rangle)$ is a possible computation step. The relation $\longrightarrow$ is the smallest relation closed under this rule.

The commitment strategy determines how conditional actions are selected when several potentially apply and this is not specified directly by the GOAL semantics.

3.2.1 GOAL Implemented with AIL

We discuss the implementation of GOAL with AIL. This work was reported in [29] so we here focus on some key aspects, in order to use it as an example.

GOAL agents were implemented as sub-classes of the AIL agent class. To model GOAL's mental states we treated the AIL belief base as the GOAL belief base Σ. Although the AIL agent does not contain a specific goal base, we were able to use the method for extracting goals from intentions to represent Γ; a slight modification was needed to the method to return only achievement goals[4].

The implementation of $\models_M$ was straightforward. Belief formulæ of the form $\mathbf{B}(\phi)$ are equivalent to the expressions appearing in AIL's guards and use of AIL's own logical consequence method was sufficient.

3.2.2 Capabilities and Conditional Actions

We chose to model both capabilities and conditional actions as AIL plans, since both control how an agent reacts to its environment.

Inherent in the description of a capability is the idea that the agent performs an action associated with the capability and then makes a number of belief updates. We treated capabilities as *perform goals* because they function as steps/sub-goals that an agent *should* perform, yet they are not declarative. AIL requires the execution of actions to be triggered explicitly so we decided to treat $\mathscr{T}(\mathbf{a},\Sigma)$ as a function on the belief base paired with an optional action. We write this as $\mathscr{T}(\mathbf{a},\Sigma) = \mathbf{do}(a) + f(\Sigma)$ and represent it in the AIL as a plan, where the range of f is a deed stack of belief updates. The enabledness of a capability is then governed by the plan guard.

For simplicity, we abstract the full agent state and consider it just as a short tuple of the agent name, current intention, and other intentions: $\langle ag, i, I\rangle$. Conditional actions were modelled as reactive plans with ϕ as the plan guard. In both cases we were able to use the default mechanisms for plan matching and application.

Although we used the default **appPlans** algorithm, we incorporated custom operational semantic rules within this. For instance, for applying a capability we used the rule:

$$\frac{\begin{array}{c}\Delta = \{\langle \mathbf{a}, a'; f'(\Sigma)\rangle \mid \mathbf{a} \in Bcap \wedge enabled(\mathbf{a}) \wedge \mathscr{T}(\mathbf{a},\Sigma) = \mathbf{do}(a') + f'(\Sigma)\} \\ \Delta \neq \emptyset \quad \mathscr{S}_{\text{plan}}(\Delta) = \langle \mathbf{a}, a; f(\Sigma)\rangle\end{array}}{\langle ag, (\mathbf{a}, \varepsilon); i, I\rangle \rightarrow \langle ag, (\mathbf{a}, a; f(\Sigma)); i', I \backslash \{i'\} \cup \{i\}\rangle} \tag{7}$$

This was a new rule but made use of pre-existing AIL operations (such as $\mathscr{S}_{\text{plan}}$). Δ represents the output of **appPlans**.

By way of illustration, we show some of the Java code for the new "Plan with Capabilities" rule. The key methods of an AIL operational semantics rule are `checkPreconditions` and `apply` which, broadly speaking, represent the conditions for valid rule application and the effects of applying the rule. The precondition for the rule application is that

$$\Delta \neq \emptyset \tag{8}$$

The equivalent code is:

```
public boolean checkPreconditions(AILAgent a)
{
        Intention i = a.getIntention();
        if (i.empty())
                return false;

        if (i.hdE().getGoal().getGoalType() == Goal.performGoal)
        {
                if (a.filterPlans(a.appPlans(i)).isEmpty())
                        return false;
                else
                        return true;
```

[4] As we discuss later, we also used perform goals to model capabilities and did not want these to be returned as part of Γ.

```
        }

        return true;
    }
```

`appPlans` is the Java implementation of **appPlans** and returns the set Δ; the above method returns true if this set is non-empty. `filterPlans` is an overridable method allowing the applicable plan list to be modified in an application specific fashion if desired. By default it does nothing. The test

```
i.hdE().getGoal().getGoalType() == Goal.performGoal
```

restricts the method so it only succeeds if the head event of intention `i` is a "perform goal" (since capabilities are represented by perform goals). The application of the rule is then governed by the code:

```
public void apply(AILAgent a)
{
        Intention i = a.getIntention();
        LinkedList<ApplicablePlan> delta = a.filterPlans(a.appPlans(i));

        GOALAgent ga = (GOALAgent) a;
        ApplicablePlan p = ga.selectPlan(delta, i);

        i.dropP(p.getN());
        i.iConcat(p.getEvent(), p.getPrefix(),
                  p.getGuard(), p.getUnifier().clone());
}
```

Here we select a capability, `p`, from Δ (`selectPlan` is the implementation of $\mathscr{S}_{\text{plan}}$). We drop (using `dropP`) the appropriate number of lines (`i`) from the intention (as specified by the applicable plan), and then add (using `iConcat`) a new row onto the intention consisting of the event, prefix, guard, and unifier of `p`. All these methods are provided by the AIL.

3.2.3 The Reasoning Cycle

AIL assumes a reasoning cycle that passes through a number of explicit stages to which rules are assigned. We therefore analysed the GOAL operational semantics to identify stages. It first selects a conditional action (this involved a new rule, similar to the rule for planning with a capability shown above); then processes the conditional action by applying $\mathscr{M}$; this generally involves selecting a capability and applying it (equivalent to using our new rule above to select a capability) and then processing the effects of that capability as specified by $\mathscr{T}$ (this was easily implemented using existing rules for performing actions and handling belief updates).

3.2.4 Faithfulness of the Implementation

Any claim to have implemented the operational semantics of a language is faced with correctness issues involved in transferring a transition system to, in this case, a set of Java classes. Verifying implementations is a complex undertaking. Such a verification effort would be a significant task and falls outside the scope of this work. However, that aside, it is also the case that we have not directly implemented the system presented in [1] but a variant of it using, for instance, our representation of capabilities. The question arises: "Are these two transition systems equivalent?" Although we have not done such proofs, we believe they would not represent an insurmountable amount of work.

4 AJPF: Verification for Multi-Agent Systems

4.1 Java PathFinder

Java PathFinder (JPF) is an explicit-state Open Source model checker for Java programs [74, 50]. JPF is implemented in Java and provides a special Java Virtual Machine (JVM) running on top of the host JVM

and exploring all executions that a given program can have, resulting from thread interleavings and non-deterministic choices. JPF's backtracking JVM executes Java bytecodes using a special representation of JVM states. Essentially, JPF uses Java's *listener* mechanism to provide a representation of an automaton that is attempting to build a model based on the program execution. As the program proceeds, the listener recognises state changes and checks against user specified properties. At appropriate times this JVM might backtrack and the listener might be reset.

JPF employs state-of-the-art state-space reduction techniques, such as on-the-fly partial-order reduction, i.e., combining instruction sequences that only have effects inside a single thread and executing them as a single transition. This ensures that — out of the box — JPF is a usable (though not very efficient) program model checker; it also should be noted that recent improvements in JPF make it orders of magnitude faster than previous versions. Nevertheless, we have to ensure that the state space *relevant to an agent system* remains as small as possible. For instance, we want to ensure that only relevant backtrack points are stored (i.e., backtracks points in the execution of the operational semantics and not incidental backtrack points in the low-level Java code), thereby limiting the state space and improving the efficiency of model checking.

As well as its usability, we have chosen JPF for several other reasons. Firstly, a large number of APL interpreters are implemented in Java, so a model checker for Java programs was naturally the first choice. Secondly, in order to customise the model checker for our needs and provide it to the public as an Open Source project, we needed the underlying model checker to be Open Source too. Since we were to do significant work in extending the model checker to an agent context (for example, using interfaces), we preferred a model checker working within a standard but higher level programming language. Lastly, we preferred a model checker with an active development team and one that had been developed to provide good extensibility mechanisms. Taking into account these prerequisites, the obvious choice was to base our framework on the Open Source Java model checker JPF.

4.2 AJPF architecture

The AJPF architecture consists of the JPF model checker, the AJPF interfaces, a property specification language (PSL), the controller and AJPF listener classes, and a family of language parsers and translators. A schematic diagram of the architecture was shown in Fig. 1.

A given program, written in one of the supported APLs, is translated into its AIL representation and embedded in an AJPF controller object. The result is a Java program that includes parts of the AIL libraries. This Java program becomes half of the JPF verification target, but instead of invoking standard JPF, we use the JPF extension mechanisms to configure our own *listener*. This implements the property specification language and performs the checking of the property that is supplied with the program. The property is represented as a Büchi Automaton [70]. The AJPF controller object combines this Büchi Automaton and the AIL program and then the controller governs when properties are checked and the Büchi Automaton advanced. This is linked to the progress of the agent reasoning cycle.

4.3 The AJPF Agent System Interfaces — Use and Semantics

AJPF (see Fig. 1) provides interfaces for model checking a multi-agent program against a property specification written in the Property Specification Language (PSL) introduced later (in Section 4.4). AJPF requires that, for any given APL, two Java interfaces are implemented: one for individual agents and another for the overall multi-agent system. This software layer provides an AJPF controller which requests a list of agents from the multi-agent system and encapsulates each of them in a special thread object which alternately calls one reasoning step of the agent, anticipated to be one full run of the reasoning cycle, and then checks this against the specification by calling, for instance, methods that implement belief checking as defined by the specific language. It should be noted that this means the notions of belief, goal, etc. defined within the AIL classes are those invoked by the property checking algorithm. So any language that has an AIL-based interpreter inherits this semantics although individual methods can, of course, be over-ridden. For languages without an AIL-based interpreter, these notions have to be formalised in the implementation of the AJPF interface. Properties are also checked when JPF detects that an "end state" is reached (this could indicate a cycle in the states of a run as well as program termination).

4.3.1 BDI Languages without Formal Semantics

The AIL toolkit is designed for the prototyping of languages which have a clearly defined BDI-based operational semantics. The construction of AJPF however does mean that *in principle* it could be used with agent platforms that have no clear semantics but which are, nevertheless, implemented in Java. The JACK framework [77], for instance, provides a set of Java classes for programming agents which embody BDI concepts such as beliefs, goals, and plans. In theory, therefore, the appropriate modalities of the property specification language could be defined for JACK programs and, assuming a reasoning steps can be identified, then JACK programs could be model checked using AJPF.

In reality such an attempt is likely to encounter efficiency problems since the JACK code will not make use of atomic sections and state matching hints in order to reduce the model checking state space, so even for a small programs full verification may not be possible in practice.

4.3.2 Environmental Models

Since we are dealing with Java programs, we generally assume that the environment in which they operate is also a Java program that can be included in the closed system for model checking. Obviously there are situations where this is not the case, for instance if the programs we are checking are intended to run on robots that operate in the real world. In these cases they have to be model checked against an abstraction of the real environment that would have to be, again, implemented in Java. The same is the case for applications which are intended to be distributed across several different machines. AJPF relies upon Java's thread model for asynchronous execution of agents and can only check a single program executing on a single machine. Therefore the effects of a distributed environment would need to be abstracted into a (possibly multi-threaded) single program environment for model checking in AJPF.

4.4 Specification of Properties

In our framework, we are interested in verifying simple properties about *goals*, *beliefs*, *actions*, etc. We do not, in this approach, tackle the verification of properties involving nested modalities, such as beliefs of one agent concerning the beliefs of another agent. Hence, we do not implement a property specification language based on a fully expressive logic of beliefs or knowledge.

A typical property of an agent-based protocol specification could be described by a statement such as: "given an agent a with a goal g and a set of current beliefs $\{b_1, \ldots, b_n\}$, will a eventually believe g?"

Properties are specified at the AJPF level. For agents running on an AIL-based interpreter, the semantics of the properties are already specified as part of the AIL toolkit itself. The PSL allows users to refer to agent concepts at a high level, even though JPF carries out model checking at the Java bytecode level.

We use a property specification language based on propositional linear-time temporal logic (LTL) [33] with added modalities for agents' beliefs, goals, etc. It should be noted that, since it was released as Open Source, JPF no longer supports LTL model checking. Our implementation therefore represents a significant addition to JPF. Also, we do not tackle more complex properties such as those involving nested beliefs and so the PSL defined below is relatively "shallow".

The PSL syntax for property formulæ ϕ is as follows, where ag is an "agent constant" and f is a ground first-order atomic formula:

$$\phi ::= \quad \mathbf{B}(ag, f) \mid \mathbf{G}(ag, f) \mid \mathbf{A}(ag, f) \mid \mathbf{I}(ag, f) \mid \mathbf{P}(f) \mid \phi \vee \phi \mid \neg\phi \mid \phi \,\mathsf{U}\, \phi \mid \phi \,\mathsf{R}\, \phi$$

Intuitively, $\mathbf{B}(ag, f)$ is true if ag believes f to be true, $\mathbf{G}(ag, f)$ is true if ag has a goal to make f true, and so on (with $\mathbf{A}$ representing actions, $\mathbf{I}$ representing intentions, and $\mathbf{P}$ representing percepts, i.e., properties that are true in the environment).

We next examine the specific semantics of property formulæ. Consider a program, P, describing a multi-agent system and let MAS be the state of the multi-agent system at one point in the run of P. Let $ag \in MAS$ be an agent at this point in the program execution. Then

$$MAS \models_{MC} \mathbf{B}(ag, f) \quad \text{iff} \quad ag \models f$$

where $\models$ is logical consequence as implemented by the agent programming language. The interpretation of $\mathbf{G}(ag, f)$ is given as:

$$MAS \models_{MC} \mathbf{G}(ag, f) \quad \text{iff} \quad f \in ag_G$$

where ag_G is the set of agent goals (as implemented by the APL). The interpretation of $\mathbf{A}(ag, f)$ is:

$$MAS \models_{MC} \mathbf{A}(ag, f)$$

if, and only if, the last action changing the environment was action f taken by agent ag. Similarly, the interpretation of $\mathbf{I}(ag, f)$ is given as:

$$MAS \models_{MC} \mathbf{I}(ag, f)$$

if, and only if, $f \in ag_G$ and there is an *intended means* for f (in AIL this is interpreted as having selected some plan that can achieve f). Finally, the interpretation of $\mathbf{P}(f)$ is given as:

$$MAS \models_{MC} \mathbf{P}(f)$$

if, and only if, f is a percept that holds true in the environment.

The other operators in the AJPF property specification language have standard LTL semantics [33] and are implemented by the AJPF interface. Thus, the classical logic operators are defined by:

$$\begin{array}{rll} MAS \models_{MC} \varphi \vee \psi & \text{iff} & MAS \models_{MC} \varphi \text{ or } MAS \models_{MC} \psi \\ MAS \models_{MC} \neg\phi & \text{iff} & MAS \not\models_{MC} \phi. \end{array}$$

The temporal formulæ apply to runs of the programs in the JPF model checker. A run consists of a (possibly infinite) sequence of program states MAS_i, $i \geq 0$ where MAS_0 is the initial state of the program (note, however, that for model checking the number of *different* states in any run is assumed to be finite). Let P be a multi-agent program, then

$$\begin{array}{lll} MAS \models_{MC} \varphi \,\mathsf{U}\, \psi & \text{iff} & \text{in all runs of } P \text{ there exists a state } MAS_j \text{ such that} \\ & & MAS_i \models_{MC} \varphi \text{ for all } 0 \leq i < j \text{ and } MAS_j \models_{MC} \psi \\ MAS \models_{MC} \varphi \,\mathsf{R}\, \psi & \text{iff} & \text{either } MAS_i \models_{MC} \varphi \text{ for all } i \text{ or there exists } MAS_j \text{ such} \\ & & \text{that } MAS_i \models_{MC} \varphi \text{ for all } i \in \{0, \ldots, j\} \text{ and} \\ & & MAS_j \models_{MC} \varphi \wedge \psi \end{array}$$

The common temporal operators $\Diamond$ (eventually) and $\Box$ (always) are derivable from U and R [33].

Typically, APLs do not fully implement logics of belief so, as mentioned above, we use shallow modalities which are like special predicates — however, this does not preclude users, when implementing the AJPF interfaces, from developing a more complex belief logic based on their agent state. The implementation of the modalities defines their semantics (e.g., for *belief*) in that specific language. The AIL implements these interfaces and so defines an AIL specific semantics for the property specification language; supported languages that use the AIL must ensure that their AIL-based interpreters are constructed in a way that makes the AIL semantics of the properties consistent with the language's individual semantics for those modalities (otherwise they cannot use the AIL implementation and will need to override it using the AJPF interface).

4.5 Heterogeneous Multi-Agent Systems

As stated above, the AJPF controller object accepts a set of AJPF agent objects and executes steps of their reasoning cycles, followed by property checks. Since it simply accepts objects that satisfy its interfaces, it is agnostic about the actual semantics of their reasoning cycles and does not require each agent to be using the same semantics at all (although it is necessary for all the agents to interact with the same environment). This makes it as simple to encode a *heterogeneous* multi-agent system as it is to encode a system using only a single agent programming language.

In [29] we investigated a scenario comprising GWENDOLEN, GOAL and SAAPL agents working together. We were able to successfully implement and verify this system within AJPF. In that work, AIL was found to be appropriate for the three languages and we were able to implement interpreters for SAAPL

and GOAL with relative ease[5]. The SAAPL interpreter took about a week to implement and debug while the GOAL interpreter took about two weeks. Once correctly implemented, it was simple to incorporate, run, and verify a heterogeneous multi-agent system.

4.6 Benefits of Using AIL and AJPF

The benefits of using the AIL are many, with the main incentives being verification via model checking and the support for heterogeneous multi-agent systems. The flexibility of our approach arises from the fact that the agents can be programmed in a variety of agent-oriented programming languages. This unifying approach to model checking and execution of (heterogeneous) agent systems is an important step towards the practical use of verification techniques, which is essential as dependable systems are increasingly required in many areas of applications of agent technology.

Previous approaches to model checking multi-agent programs focused on a specific APL, e.g. AgentSpeak [7]. A language-specific translation of multi-agent systems into models written in the input language of exiting model checkers, generating models that were very difficult to understand, had to be developed; it should also be noted that there is an incredible number of different agent programming languages currently in use in the Agents community. Conceiving and implementing such translations is a tedious, complicated (given the restrictions of typical model checker input languages), and error-prone task that is avoided in the present approach by using an intermediate agent representation that is tailored to multi-agent systems. By lifting the implementation effort away from the model checker to AIL, we make it less tedious and error-prone as we provide tools that are a good match to the operational semantics of agent programming languages.

The architecture of the AIL and AJPF is much more flexible than previous approaches to model checking for agent-based systems. Despite the greater flexibility, we have taken precautions in the construction of the architecture and the internal optimisations of AJPF to ensure that it works relatively efficiently.

We have developed AIL so that new APLs can easily be incorporated into our framework. Even without re-programming a language interpreter using the AIL classes, it is possible to integrate agent programs written in a variety of languages into our verification and execution framework by interfacing their interpreters directly to AJPF. Property specification is uniform amongst all languages that use either the AIL data structures or implement their own notions of belief, goal, action, etc.

4.7 Efficiency Issues

In general, model checking suffers from what is known as the *state-space explosion problem*, i.e., the problem that the state space for search increases exponentially as the system increases in size. It is therefore important to make sure that only essential information is stored in the states that constitute the system to be checked. Using JPF means that the state space that is actually checked is the state space of the Java program representing the AIL agent program. In the remainder, when speaking of the *abstract state space*, we refer to the state space of the multi-agent system bar any additions to the state space that might have been introduced by the translation and use of AIL classes. For efficiency it is, of course, desirable to avoid as many additional states as possible that do not add to the overall behaviour, thus being theoretically harmless.

In spite of this, AJPF is *not* fast. Partly, this is because JPF is itself not a particularly fast model checker (though it is both flexible and appropriate) and partly it is because of the non-trivial additional semantic layer that is added in AJPF. Interestingly recent work on the comparison of model checkers for the GOAL language [49] concluded that all existing approaches to model-checking agent programming languages that were based on pre-existing model checkers suffered from similar time inefficiencies.

In the following sections, we describe our efforts towards dealing with the issues of efficiency.

4.7.1 Atomic Execution

We employ *atomic sections* to reduce the state space of executions whose internal states are not relevant to the execution of the multi-agent system. An atomic section excludes all backtracking within that section.

[5] An interpreter for GWENDOLEN had been developed while implementing the AIL.

We use this in a number of places where thread interaction is not relevant to the agent transition system.

For instance, using an atomic section for the initialisation phase of the agents and of the MAS leads to a significant speed-up. This portion of the code is executed many times as JPF backtracks (thereby initialising the agents in a different order) and substantial savings result from this. Further use of atomic sections in the reasoning cycle also help improve efficiency.

4.7.2 State Matching

In model checking it is important to take care that, in the system to be verified, the states that are conceptually identical do not contain any components that would make the model checker distinguish between them. It is therefore essential to restrict the data structures to the abstract agent system, hiding from the model checker any components that might have been introduced for operational reasons or to provide statistics, such as counters, that do not inherently belong to the agent state. JPF's state matching is an important mechanism to avoid unnecessary work. The execution state of a program mainly consists of heap and thread-stack snapshots. During the execution of a program, JPF checks every state it reaches against its history. If an equivalent state had been reached before, there is no need to continue along the current execution path. In this case, JPF backtracks to the last unexplored non-deterministic choice.

JPF provides a means to tag variables or parts of data structures, so that the model checker ignores them. This is essential to get state matching to work in the presence of counters, etc. JPF supports this abstraction through the `@FilterField` annotation. Applying this to (part of) a data structure explicitly declares the structure to be exempt from state matching.

JPF uses its own internal mechanisms for state matching based on Jenkins hashes [47]. It is outside the scope of this work to discuss the implementation and the trade-offs involved in the efficiency of compiling the hash. We have observed that, since execution of the Java engine is comparatively slow and is independent of the implementation of AIL, there are important efficiency gains to be made if as many states can be matched as possible.

4.7.3 Property Checking

Checking temporal properties, as defined in our property specification language, can cause branches of the search space where the property automaton branches. We therefore limit the places in which these properties need to be checked. In general we only check properties at the end of the execution of a whole reasoning cycle rather than, for instance, after the application of every operational semantic rule.

5 Evaluation

Having described both AIL and AJPF, we now provide a number of scenarios showing the whole system in action. While these examples are relatively simple, they exhibit all the functionality of the MCAPL system.

In these examples we target one particular BDI language. Although this language is simple, it is designed to exhibit many features common to BDI languages in general. Agents are represented as sets of initial beliefs and goals together with a library of plans. A multi-agent system is a set of agents, together with an environment, through which communication occurs and in which actions are performed.

Below, we first describe the various scenarios and then, in Section 5.2, discuss their verification.

5.1 Verified Scenarios

5.1.1 Contract Net Example

The *Contract Net* scenario [72] is a well-known, and widely used, model of cooperation in distributed problem-solving. Essentially, a particular agent (the *manager*) broadcasts tasks (goals) to be accomplished, and then agents capable of doing so bid for the contract. In real scenarios, the bidding, allocation, and sub-contracting can be quite complex. However, we consider a very simple version: the manager does not broadcast to all the agents in the system at once but instead contacts them in turn; there is no bidding

process nor sub-contracting; agents volunteer for a task if, and only if, they can perform it; and the manager simply accepts the first proposal it receives.

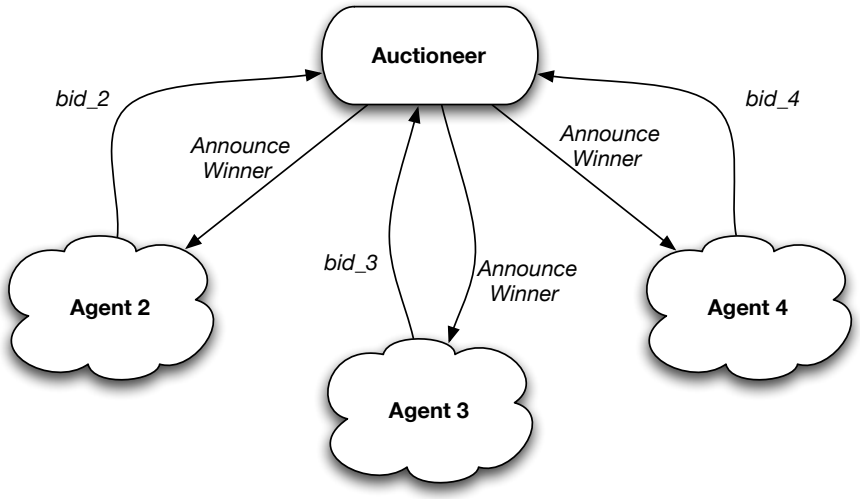
We investigated the model checking of this scenario with up to six agents attempting to achieve either one or two goals requested by the manager. The code for the version with two bidding agents and two goals can be found in Appendix B.2.

5.1.2 Auction Scenarios

We also considered a simple auction example. The basic idea of an *auction* [73, 52] is at the heart of many multi-agent scenarios [12]. Not only are auctions central to e-commerce applications [20, 37], they are implicit within many *market-based* approaches to agent computation [75]. These include areas where resource allocation or task allocation is required, for example in telecommunications [40, 39], electricity supply management [21], agent mobility [15], logistics [23], and scheduling [65]. However, although much work has been carried out on deep analysis of auction mechanisms, such as through formal mechanism design [80], the analysis of *implementations* of auction mechanisms has lagged behind. While there has been some work on the formal verification of auction implementations, such as [31], this has lacked an agent perspective. Thus, the more sophisticated agent aspects such as goals, intentions, beliefs, and deliberation are not typically verified within an auction context.

The basic version of this study is initially very simple. We describe the basic scenario below and then, in subsequent sections, we describe more sophisticated variants, each becoming increasingly realistic.

A Very Basic Auction The idea here is simple. A number of agents (in the diagram below, three) make bids of some value to an auctioneer agent. The auctioneer agent then awards the resource to the highest bidder and announces this. This cycle can then repeat, if necessary (note that, in our verified scenarios, the bidding process does *not* cycle).



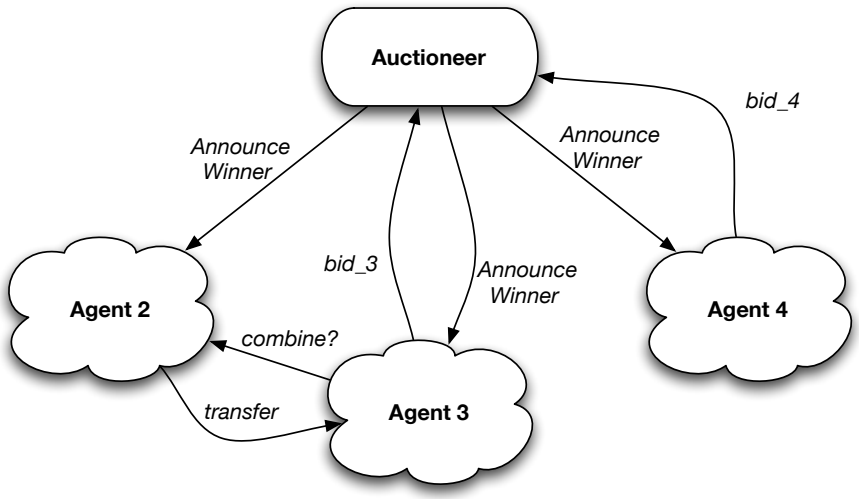
Versions of this scenario with increasing numbers of agents were implemented in GWENDOLEN. The code for the four-agent version can be found in Appendix B.3.

Auction Coalition Scenario The above basic scenario was next extended to include the possibility of *coalitions* [68, 54]. In our model, a coalition occurs when several agents collaborate by pooling their bid resources in order to win the auction. For example, if three agents x, y, and z bid 100, 150, and 200 respectively, then z ought to win every time. However, if x and y form a coalition, their collective bid of 250 will be enough to win the auction.

A simple coalition scenario was implemented in GWENDOLEN with an auctioneer and a variable number of bidders. A version of the code for this scenario with 4 agents is shown in Appendix B.4. In that

version, all but one of the bidders bid straight away, but one of the agents attempts to form a coalition by communicating with one of the other bidders. The contacted bidder agrees to form the coalition, and informs the coalition former of its bidding amount. The coalition instigator then combines its own bidding amount with that of its coalition partner and submits this bid to the auctioneer. Then, having received all of the bids, the auctioneer announces the winner.

Below, Agent 3 instigates the coalition:



The main difference in the implementation of this scenario, as compared with our earlier one, is that one agent, Agent 3, has a goal to form a coalition. Agent 3 then contacts Agent 2 and proposes a coalition. If Agent 2 agrees then Agent 3 can now bid a winning 250 (i.e., $100 + 150$). Clearly, we would like to verify that this approach does, indeed, lead to Agent 3 winning the auction. This is one of the properties we verify in Section 5.2.

Dynamic Auction Coalition Scenario A further variant on the auction coalition scenario was implemented. In this case, a round of bidding takes place in which all agents bid. Then, after an agent discovers that it has lost the auction, it sends a message to another agent (excluding the previous winner) to form a coalition. Then, the agents bid again. Sample code can be found in Appendix B.5.

Coalition Trust Scenario This auction scenario is similar to that described in Section 5.1.2, except the coalition forming agent now has a belief about which other agent(s) it can trust, i.e., the other agents with which it would *prefer* to form a coalition. This trust aspect is static, that is, the coalition-forming agent starts the auction with belief(s) about which agents it can trust, and these do not change during the auction. Sample code for this scenario can be seen in Appendix B.6.

Dynamic Trust Scenario This final auction scenario builds upon the previous one. Here, if the coalition-forming agent loses the auction, it tries to form a coalition with an agent it trusts. Then, if its coalition is successful in winning the auction, it stops. However, if its coalition is *unsuccessful* then it no longer believes that it can trust the other agent in the coalition, and will try to form another coalition with another agent it trusts (excluding the winner). Again, sample code for this scenario can be seen in Appendix B.7.

5.1.3 Trash Collection Robots

Our last example is based on the garbage collection agents reported in [11, 7]. This example was previously written in AgentSpeak and then verified in both the Spin [46] and JPF model checkers. This work was the immediate precursor of the work reported here that represents an attempt to re-engineer the model checking system to make it more generic. In our previous examples, we were interested in the effects on the system of adding more agents to it. Here, our intention was to gain some idea of the cost of the more generic

architecture. The scenario investigated involved two robots (theoretically on Mars) detecting and burning two pieces of garbage placed at random on a 5 by 5 grid. The first robot searches for garbage, picks it up and takes it to the second robot. The second robot then picks up the garbage and incinerates it. We translated the original AgentSpeak code directly into GWENDOLEN. This code is shown in Appendix B.8.

5.2 Results

We investigated a number of aspects of the verification of our case studies, as reported in the sections below.

5.2.1 Effect of Scaling the Program

We started by investigating the effect of the complexity of the program on the size of the state space. As a crude measure of an increase in the program complexity we investigated the effect of adding an additional agent into the contract net and auction scenarios. We did not investiage the trash collection robot scenario in this way since this scenario involved no communication between the agents and was developed primarily in order to provide a comparison with previous work. In the contract net scenario, we verified that eventually the manager believed all its goals were achieved. In the auction scenario, we verified that eventually the agent making the highest bid believed it had won.

The effects of adding additional agents on the state space are shown in Fig. 2. As can be seen, the

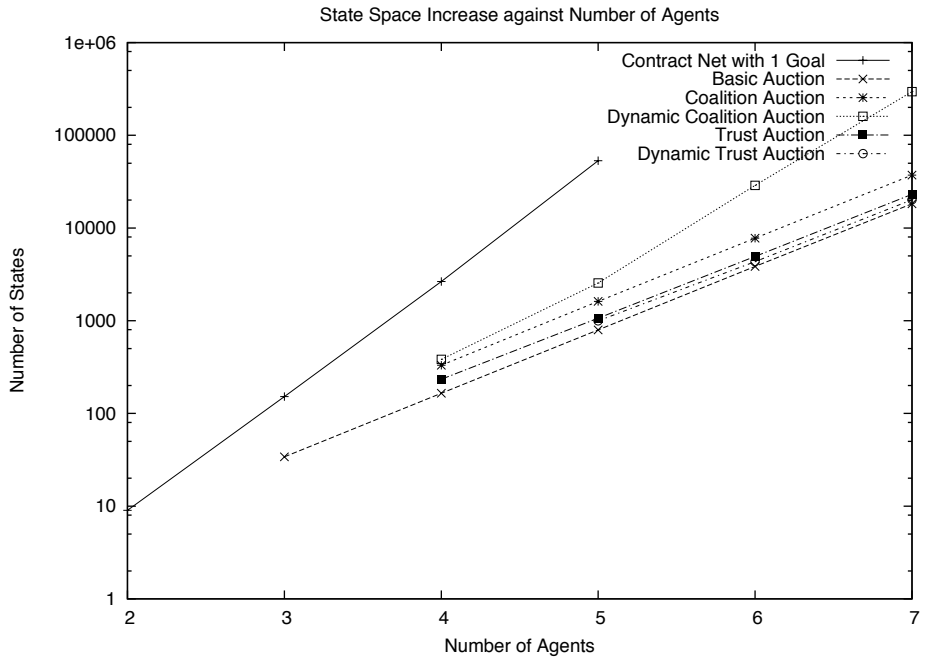


Figure 2: State Space Increase as Number of Agents Increases

size of the state space increases exponentially as more agents are added into a scenario. This represents a typical result for a model checking exercise.

Although less informative since execution time can easily be affected by factors other than the program under consideration, we also investigated the effect of extra agents against the time taken for a program to be verified. The results of this are shown in Fig. 3. As can be seen, although we have comparatively few states in our space compared to many model checking systems, we nevertheless take considerable time to verify a program. This is because each transition in the state graph of the model checker takes significant time to execute as the JPF JVM processes many bytecode instructions.

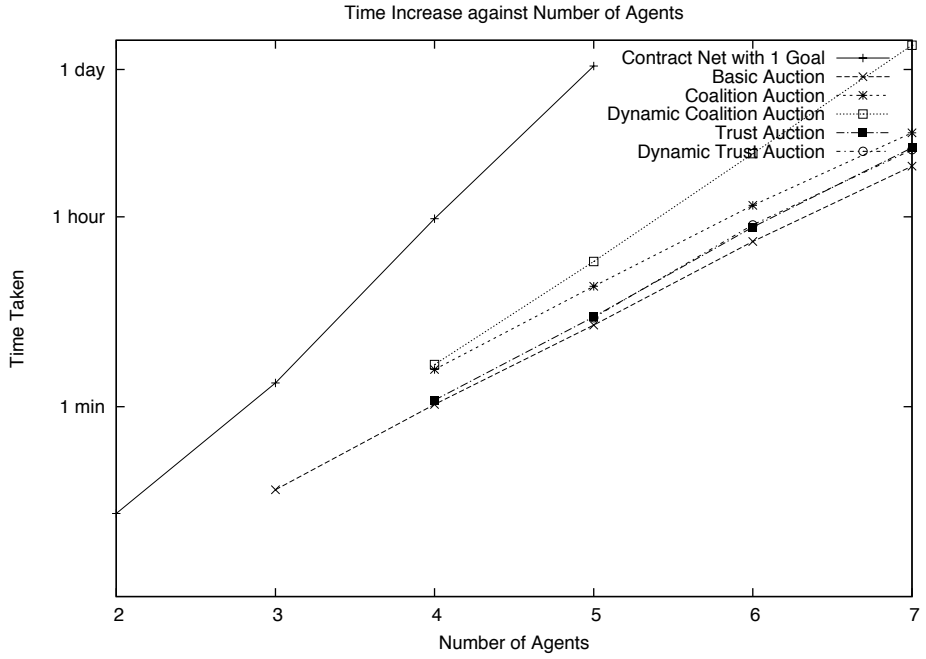


Figure 3: Time Taken as Number of Agents Increases

5.2.2 Effect of the Property Size

As well as checking properties about the beliefs of the agents, we also checked properties related to their goals, actions, intentions, etc. In theory, since the BDI modalities of the property specification language are treated as propositions by the property automata, the effect of the property on the model checking state space should be the same as for standard LTL model checking [46].

In the Contract Net scenario, on a system consisting of three agents bidding to perform one goal, we checked the validity of the properties shown in Fig. 4 with their resulting state space size. As predicted, the size of the state space appears primarily to depend upon the LTL elements with disjunctions involving eventualities creating the largest state spaces.

Property	No. of States
$\Diamond \mathbf{B}(ag1, g)$	152
$\Diamond \mathbf{P}(g)$	106
$\Box(\mathbf{G}(ag2, g) \rightarrow \Diamond \mathbf{A}(ag2, g))$	281
$\Diamond(\mathbf{G}(ag2, g) \vee \mathbf{G}(ag3, g))$	79
$\Diamond(\mathbf{I}(ag2, g) \vee \mathbf{I}(ag3, g))$	79
$\Box(\mathbf{G}(ag3, respond(g, ag1)) \rightarrow \Diamond(\mathbf{B}(ag2, award(g)) \vee \mathbf{A}(ag3, a)))$	294

Figure 4: Varying the Property Checked

5.2.3 Comparison with Previous Work

Lastly, we compared the performance of our model checking system against the previous work for model checking AgentSpeak systems. Bordini et al. [11, 7] reported that verifying the property $\Diamond(\mathbf{I}(r1, continue(check)) \wedge \mathbf{B}(r1, checking(slots))$ took 333,413 states and 65 seconds to verify in Spin and 236,946 states and 18 hours to verify in JPF. In our (JPF-based) system the verification of the same property used 23,655 states and took 9 hours. Clearly the Spin based system remains vastly superior in terms of efficiency. It is hard to accu-

rately compare the results for the JPF based systems since JPF itself has been the subject of continuous development. However, it is clear that the generic approach is unlikely to be *worse* in terms of efficiency than the language-specific approach taken previously, although it is known that JPF is currently much faster than it used to be.

6 Conclusions

6.1 Summary

In this paper we have described the development of the MCAPL framework, incorporating the AIL intermediate semantic layer and the AJPF enhanced model checker. We have seen that the AIL semantic structures are sufficiently expressive to allow developers to capture a range of BDI programming languages and that the semantic rules fit well with the modified AJPF model checker. The efficacy and generality of the AIL toolkit has also been established by the implementation of a variety of different agent programming languages and the verification of multi-agent systems implemented in those languages. Interpreters have been implemented for GWENDOLEN [27], GOAL [1], SAAPL [78], and ORWELL [25]. Interpreters for AgentSpeak [10] and 3APL [24] are also being developed. Importantly, the MCAPL framework is also appropriate for verifying *heterogeneous* multi-agent systems, as well as homogeneous ones.

The overall approach has been designed and implemented. It has also been tested on some small multi-agent programs: variations of the *contract net* protocol [71] and *auction systems*, but with five or fewer agents [76]. Specifically, we have focused on a series of scenarios of increasing complexity in order to demonstrate that, although the difficulty of the model checking task increases with each scenario, it is nevertheless realistic to model-check the properties of interesting multi-agent implementations within a reasonable time.

Thus, the MCAPL framework provides a generic harness for automatically verifying agent software. Clearly, for bigger scenarios, improved efficiency will be required (see the discussion in the next section), but the examples implemented and verified in this paper demonstrate that simple properties of multi-agent systems can already be tackled.

6.2 Efficiency Problems

A typical problem in model checking, particularly of concurrent systems (where various entities have independent, yet interacting, behaviour), is that of *state space explosion*. The model checker needs to build an in-memory model of the states of the system, and the number of such states grows exponentially for example in the number of different entities being modelled.

Even with refined representation techniques, such as the BDDs used in *symbolic model checking* [16], the formulæ/structures required to represent the state spaces of realistic systems are huge. JPF is an explicit-state, on-the-fly model checker, and a further problem is that the underlying JPF virtual machine is rather slow. Thus, our current verification system is also slow (although recent work shows that its performance is comparable to a similar system implemented in Maude [49]). Although speed is the main problem, space required can also be problematic [5] (though note that the slow examples above actually explore fewer than 500,000 states in total). We should also note that the success of program model checking relies a great deal on *state-space reduction techniques*, which we have also adapted for agent verification in [8], but have not yet implemented to work at the AIL level.

However, our approach is no less efficient than the language-specific work reported in [7]. Thus, it is our belief that the generic design principles embodied in the MCAPL framework could be transported to other model checking systems and it is not the *generality* of the framework which is the main issue in terms of efficiency.

6.3 Future Work

Our proposed future work falls into three main areas.

Firstly, we would like to extend the agent programming languages available within the system to include, at the least, the *Jason* [9, 10] implementation of AgentSpeak and 3APL [43, 24]. At the same time we would like to improve the support for the languages we already have implemented in terms of supplying more complete parsers and translators for them so that programs written for other implementations of those languages can be easily imported into our system and run.

Secondly, we intend to improve the model checking aspects of the framework. In particular, we would like to investigate the use of "mixed execution" in JPF [22]. This would allow us to delegate the operation of parts of the Java code to the native, efficient Java Virtual Machine rather than using the JPF virtual machine. This involves identifying appropriate methods and data structures which are irrelevant to the correct storage of the system state for backtracking. We have made some initial, inconclusive, investigations into delegating the unification algorithm in this way, but work elsewhere suggests that we should be able to achieve significant time improvements.

Lastly, we are interested in replacing the JPF back end to the system with a different model checker such as Spin [46] or NuSMV [17]. Previous work [11, 7] suggests that a considerable speed up may be possible in another model checker but that more work would be required in creating a framework in which the model checker could simply tackle a range of different agent programming languages. It is our belief that much of the design work reported here could be adapted to an alternative system.

Acknowledgements The authors would like to thank Berndt Farwer for help in initial stages of this work.

References

[1] de Boer, F.S., Hindriks, K.V., van der Hoek, W., Meyer, J.J.C.: A Verification Framework for Agent Programming with Declarative Goals. Journal of Applied Logic **5**(2), 277–302 (2007)

[2] Bond, A.H., Gasser, L. (eds.): Readings in Distributed Artificial Intelligence. Morgan Kaufmann (1988)

[3] Bordini, R.H., Dastani, M., Dix, J., El Fallah Seghrouchni, A. (eds.): Multi-Agent Programming: Languages, Platforms and Applications. Springer-Verlag (2005)

[4] Bordini, R.H., Dastani, M., Dix, J., El Fallah Seghrouchni, A. (eds.): Multi-Agent Programming: Languages, Tools and Applications. Springer, 2009.

[5] Bordini, R.H., Dennis, L.A., Farwer, B., Fisher, M.: Automated Verification of Multi-Agent Programs. In: Proc. 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 69–78. 2008 (2008)

[6] Bordini, R.H., Fisher, M., Visser, W., Wooldridge, M.: Model Checking Rational Agents. IEEE Intell. Syst. **19**(5), 46–52 (2004)

[7] Bordini, R.H., Fisher, M., Visser, W., Wooldridge, M.: Verifying Multi-Agent Programs by Model Checking. Journal of Autonomous Agents and Multi-Agent Systems **12**(2), 239–256 (2006)

[8] Bordini, R.H., Fisher, M., Wooldridge, M., Visser, W.: Property-Based Slicing for an Agent-Oriented Programming Language. J. Logic and Comput. pp. exp029+ (2009)

[9] Bordini, R.H., Hübner, J.F., Vieira, R.: Jason and the Golden Fleece of Agent-Oriented Programming. In: Bordini et al. [3], chap. 1, pp. 3–37

[10] Bordini, R.H., Hübner, J.F., Wooldridge, M.: Programming Multi-Agent Systems in AgentSpeak Using Jason. Wiley Series in Agent Technology. John Wiley & Sons (2007)

[11] Bordini, R.H., Visser, W., Fisher, M., Wooldridge, M.: Verifiable multi-agent programs. In: First International Workshop on Programming Multiagent Systems: Languages, Frameworks, Techniques and Tools (ProMAS-03), *Lecture Notes in Artificial Intelligence*, vol. 3067. Springer (2003)

[12] Boutilier, C., Shoham, Y., Wellman, M.P.: Economic Principles of Multi-Agent Systems. Artif. Intell. **94**(1-2), 1–6 (1997)

[13] Bratman, M.E.: Intentions, Plans, and Practical Reason. Harvard University Press: Cambridge, MA (1987)

[14] Bratman, M.E., Israel, D.J., Pollack, M.E.: Plans and Resource-Bounded Practical Reasoning. Computational Intelligence **4**, 349–355 (1988)

[15] Bredin, J., Kotz, D., Rus, D., Maheswaran, R.T., Ç. Imer, Basar, T.: Computational Markets to Regulate Mobile-Agent Systems. Journal of Autonomous Agents and Multi-Agent Systems **6**(3), 235–263 (2003)

[16] Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, L.J.: Symbolic Model Checking: 10^{20} States and Beyond. Inf. Comput. **98**(2), 142–170 (1992)

[17] Cimatti, A., Clarke, E.M., Guinchiglia, E., Guinchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV 2: An OpenSource Tool for Symbolic Model Checking. In: Proc. International Conference on Computer-Aided Verification (CAV 2002), Lecture Notes in Computer Science. Springer, Copenhagen, Denmark (2002)

[18] Clarke, E., Grumberg, O., Peled, D.: Model Checking. MIT Press (1999)

[19] Cohen, P.R., Levesque, H.J.: Intention is Choice with Commitment. Artificial Intelligence **42**, 213–261 (1990)

[20] Collins, J., Faratin, P., Parsons, S., Rodríguez-Aguilar, J.A., Sadeh, N.M., Shehory, O., Sklar, E. (eds.): Agent-Mediated Electronic Commerce and Trading Agent Design and Analysis (AMEC/-TADA), *Lecture Notes in Business Information Processing*, vol. 13. Springer (2009)

[21] Corera, J.M., Laresgoiti, I., Jennings, N.R.: Using Archon, Part 2: Electricity Transportation Management. IEEE Intelligent Systems **11**(6), 71–79 (1996)

[22] D'Amorim, M.: Efficient Explicit-state Model Checking for Programs with Dynamically Allocated Data. Ph.D. thesis, University of Illinois at Urbana-Champaign, Champaign, IL, USA (2007). Adviser-Marinov, Darko

[23] Dash, R.K., Vytelingum, P., Rogers, A., David, E., Jennings, N.R.: Market-Based Task Allocation Mechanisms for Limited-Capacity Suppliers. IEEE Trans. Systems, Man, and Cybernetics, Part A **37**(3), 391–405 (2007)

[24] Dastani, M., van Riemsdijk, M.B., Meyer, J.J.C.: Programming Multi-Agent Systems in 3APL. In: Bordini et al. [3], chap. 2, pp. 39–67

[25] Dastani, M., Tinnemeier, N.A.M., Meyer, J.J.C.: A programming language for normative multi-agent systems. In: Dignum, V. (ed.) Multi-Agent Systems: Semantics and Dynamics of Organizational Models, chap. 16. IGI Global (2009)

[26] Davis, R., Smith, R.G.: Negotiation as a Metaphor for Distributed Problem Solving. Artificial Intelligence **20**(1), 63–109 (1983)

[27] Dennis, L.A., Farwer, B.: Gwendolen: A BDI Language for Verifiable Agents. In: Löwe, B. (ed.) Logic and the Simulation of Interaction and Reasoning. AISB, Aberdeen (2008). AISB'08 Workshop

[28] Dennis, L.A., Farwer, B., Bordini, R.H., Fisher, M., Wooldridge, M.: A Common Semantic Basis for BDI Languages. In: Proc. 7th International Workshop on Programming Multiagent Systems (ProMAS), *Lecture Notes in Artificial Intelligence*, vol. 4908, pp. 124–139. Springer Verlag (2008)

[29] Dennis, L.A., Fisher, M.: Programming Verifiable Heterogeneous Agent Systems. In: Proc. 6th International Workshop on Programming in Multi-Agent Systems (ProMAS), *LNCS*, vol. 5442, pp. 40–55. Springer Verlag (2008)

[30] Dennis, L.A., Hepple, A., Fisher, M.: Language Constructs for Multi-Agent Programming. In: Proc. 8th International Workshop on Computational Logic in Multi-Agent Systems (CLIMA), *Lecture Notes in Artificial Intelligence*, vol. 5056, pp. 137–156. Springer (2008)

[31] Doghri, I.: Formal Verification of WAHS: an Autonomous and Wireless P2P Auction Handling System. In: Proc. 8th International Conference on New Technologies in Distributed Systems (NOTERE), pp. 1–10. ACM, New York, NY, USA (2008)

[32] Durfee, E.H., Lesser, V.R., Corkill, D.D.: Trends in Cooperative Distributed Problem Solving. IEEE Trans. Knowledge and Data Engineering **1**(1), 63–83 (1989)

[33] Emerson, E.A.: Temporal and Modal Logic. In: van Leeuwen, J. (ed.) Handbook of Theoretical Computer Science, pp. 996–1072. Elsevier (1990)

[34] Fisher, M., and Hepple, A: Executing Logical Agent Specifications. In Bordini et al. [4], pages 1–27.

[35] Fisher, M., and Ghidini, C: Executable Specifications of Resource-Bounded Agents. *Journal of Autonomous Agents and Multi-Agent Systems*, 21(3):368–396, 2010.

[36] Fisher, M., Singh, M.P., Spears, D.F., Wooldridge, M.: Logic-Based Agent Verification (Editorial). Journal of Applied Logic **5**(2), 193–195 (2007)

[37] Fortnow, L., Riedl, J., Sandholm, T. (eds.): Proc. 9th ACM Conference on Electronic Commerce (EC). ACM (2008)

[38] Gabbay, D., Kurucz, A., Wolter, F., Zakharyaschev, M.: Many-Dimensional Modal Logics: Theory and Applications. No. 148 in Studies in Logic and the Foundations of Mathematics. Elsevier Science (2003)

[39] Gibney, M.A., Jennings, N.R., Vriend, N.J., Griffiths, J.M.: Market-Based Call Routing in Telecommunications Networks Using Adaptive Pricing and Real Bidding. In: Proc. 3rd International Workshop on Intelligent Agents for Telecommunication Applications (IATA), *Lecture Notes in Computer Science*, vol. 1699, pp. 46–61. Springer (1999)

[40] Haque, N., Jennings, N.R., Moreau, L.: Resource Allocation in Communication Networks using Market-based Agents. Knowledge-Based Systems **18**(4-5), 163–170 (2005)

[41] Hierons, R.M., Bogdanov, K., Bowen, J.P., Cleaveland, R., Derrick, J., Dick, J., Gheorghe, M., Harman, M., Kapoor, K., Krause, P., Lüttgen, G., Simons, A.J.H., Vilkomir, S.A., Woodward, M.R., Zedan, H.: Using Formal Specifications to Support Testing. ACM Comput. Surv. **41**(2), 1–76 (2009)

[42] Himoff, J., Skobelev, P., Wooldridge, M.: MAGENTA Technology: Multi-agent Systems for Industrial Logistics. In: Proc. 4th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS), pp. 60–66. ACM Press, New York, NY, USA (2005)

[43] Hindriks, K., de Boer, F., van der Hoek, W., Meyer, J.J.: Formal Semantics for an Abstract Agent Programming Language. In: Intelligent Agents IV: Proc. 4th International Workshop on Agent Theories, Architectures and Languages, *Lecture Notes in Artificial Intelligence*, vol. 1365, pp. 215–229. Springer-Verlag (1998)

[44] Hindriks, K., de Boer, F., van der Hoek, W., Meyer, J.J.: Agent Programming with Declarative Goals. In: Intelligent Agents VII — Proc. 6th International Workshop on Agent Theories, Architectures, and Languages (ATAL), *Lecture Notes in Artificial Intelligence*, vol. 1986, pp. 228–243. Springer-Verlag (2001)

[45] Hirsch, B., Fricke, S., Kroll-Peters, O., Konnerth, T.: Agent Programming in Practise – Experiences with the JIAC IV Agent Framework. In: Proc. Workshop "From Agent Theory to Agent Implementation" (AT2AI) (2008)

[46] Holzmann, G.J.: The Spin Model Checker: Primer and Reference Manual. Addison-Wesley (2003)

[47] Jenkins, B.: Hash Functions. Dr. Dobbs Journal, September 1997.

[48] Jennings, N.R., Wooldridge, M.: Applications of Agent Technology. In: Agent Technology: Foundations, Applications, and Markets. Springer-Verlag, Heidelberg (1998)

[49] Jongmans, S.S., Hindriks, K., van Riemsdijk, M.: Model checking agent programs by using the program interpreter. In: Dix, J., Leite, J.a., Governatori, G., Jamroga, W. (eds.) Computational Logic in Multi-Agent Systems, *Lecture Notes in Computer Science*, vol. 6245, pp. 219–237. Springer Berlin / Heidelberg (2010). URL http://dx.doi.org/10.1007/978-3-642-14977-1_17. 10.1007/978-3-642-14977-1_17

[50] Java PathFinder (2009). http://javapathfinder.sourceforge.net

[51] Kacprzak, M., Lomuscio, A., Penczek, W.: Verification of Multiagent Systems via Unbounded Model Checking. In: Proc. 3rd International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS), pp. 638–645. IEEE Computer Society (2004)

[52] Klemperer, P.: Auctions: Theory and Practice. Princeton University Press, Princeton, USA (2004). See also http://www.nuff.ox.ac.uk/users/klemperer/VirtualBook/VBCrevisedv%2.asp

[53] Klügl, F., Bazzan, A., Ossowski, S. (eds.): Applications of Agent Technology in Traffic and Transportation. Whitestein Series in Software Agent Technologies and Autonomic Computing. Birkhäuser (2005)

[54] Konishi, H., Ray, D.: Coalition Formation as a Dynamic Process. Journal of Economic Theory **110**(1), 1 – 41 (2003)

[55] Ljunberg, M., Lucas, A.: The OASIS Air Traffic Management System. In: Proc. 2nd Pacific Rim International Conference on AI (PRICAI) (1992)

[56] Luck, M., McBurney, P., Shehory, O., Willmott, S.: Agent Technology: Computing as Interaction (A Roadmap for Agent Based Computing). AgentLink (2005)

[57] Moreno, A., Garbay, C.: Software Agents in Health Care. Artificial Intelligence in Medicine **27**(3), 229–232 (2003)

[58] Muscettola, N., Nayak, P.P., Pell, B., Williams, B.: Remote Agent: To Boldly Go Where No AI System Has Gone Before. Artificial Intelligence **103**(1-2), 5–48 (1998)

[59] Owre, S., Shankar, N.: A brief overview of pvs. In: Proc. 21st International Conference on Theorem Proving in Higher Order Logics (TPHOLs), *Lecture Notes in Computer Science*, vol. 5170, pp. 22–27. Springer (2008)

[60] Pokahr, A., Braubach, L., Lamersdorf, W.: Jadex: A BDI Reasoning Engine. In: Bordini et al. [3], pp. 149–174

[61] Raimondi, F., Lomuscio, A.: Automatic Verification of Multi-agent Systems by Model Checking via Ordered Binary Decision Diagrams. Journal of Applied Logic **5**(2), 235–251 (2007)

[62] Rao, A.: AgentSpeak(L): BDI Agents Speak Out in a Logical Computable Language. In: Proc. 7th European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW), *Lecture Notes in Computer Science*, vol. 1038, pp. 42–55. Springer (1996)

[63] Rao, A.S., Georgeff, M.: BDI Agents: From Theory to Practice. In: Proc. 1st International Conference on Multi-Agent Systems (ICMAS), pp. 312–319. San Francisco, USA (1995)

[64] Rao, A.S., Georgeff, M.P.: An Abstract Architecture for Rational Agents. In: Rich, C., Swartout, W., Nebel, B. (eds.) Proceedings of Knowledge Representation and Reasoning (KR&R-92), pp. 439–449 (1992)

[65] Reeves, D.M., Wellman, M.P., MacKie-Mason, J.K., Osepayshvili, A.: Exploring Bidding Strategies for Market-based Scheduling. Decision Support Systems **39**(1), 67–85 (2005)

[66] van Riemsdijk, B., van der Hoek, W., Meyer, J.J.: Agent Programming in Dribble: from Beliefs to Goals with Plans. In: Proc. 2nd International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS), pp. 393–400. ACM (2003)

[67] Rimassa, G., Burmeister, B.: Achieving Business Process Agility in Engineering Change Management with Agent Technology. In: Proc, 8th AI*IA/TABOO Joint Workshop "From Objects to Agents" — Agents and Industry: Technological Applications of Software Agents (WOA), pp. 1–7. Seneca Edizioni Torino (2007)

[68] Sandholm, T., Lesser, V.R.: Coalitions Among Computationally Bounded Agents. Artificial Intelligence **94**(1-2), 99–137 (1997)

[69] Shoham, Y.: Agent-Oriented Programming. Artificial Intelligence **60**(1), 51–92 (1993)

[70] Sistla, A.P., Vardi, M., Wolper, P.: The Complementation Problem for Büchi Automata with Applications to Temporal Logic. Theor. Comput. Sci. **49**, 217–237 (1987)

[71] Smith, R.G.: A Framework for Distributed Problem Solving. UMI Research Press (1980)

[72] Smith, R.G., Davis, R.: Frameworks for Cooperation in Distributed Problem Solving. IEEE Transactions on Systems, Man, and Cybernetics **11**(1), 61–70 (1980)

[73] Vickrey, W.: Counterspeculation, Auctions, and Competitive Sealed Tenders. The Journal of Finance **16**(1), 8–37 (1961)

[74] Visser, W., Havelund, K., Brat, G.P., Park, S., Lerda, F.: Model Checking Programs. Automated Software Engineering **10**(2), 203–232 (2003)

[75] Walsh, W.E., Wellman, M.P.: A Market Protocol for Decentralized Task Allocation. In: Proc. 3rd International Conference on Multiagent Systems (ICMAS), pp. 325–332. IEEE Computer Society (1998)

[76] Webster, M.P., Dennis, L.A., Fisher, M.: Model-Checking Auctions, Coalitions and Trust. Tech. Rep. ULCS-09-004, Department of Computer Science, University of Liverpool (2009). `http://www.csc.liv.ac.uk/research`

[77] M. Winikoff. JACK^{TM} intelligent agents: An industrial strength platform. In Bordini et al. [3], chapter 7, pages 175–193.

[78] Winikoff, M.: Implementing Commitment-Based Interactions. In: Proc. 6th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS), pp. 1–8. ACM, New York, USA (2007)

[79] Winikoff, M., Padgham, L., Harland, J., Thangarajah, J.: Declarative and Procedural Goals in Intelligent Agent Systems. In: Proc. 8th International Conference on Principles of Knowledge Representation and Reasoning (KR), pp. 470–481. Morgan Kaufmann (2002)

[80] Wooldridge, M., Ågotnes, T., Dunne, P.E., van der Hoek, W.: Logic for Automated Mechanism Design - A Progress Report. In: Proc. 22nd National Conference on Artificial Intelligence (AAAI), pp. 9–. AAAI Press (2007)

APPENDIX

A AIL Operational Semantics Rules

A.1 Introduction and Notation

The purpose of this chapter is to provide a reference for the presupplied rules that may be used in the operational semantics of a language implemented in the AIL. We have tried to present these in as clear a fashion as possible excluding implementation details where possible.

An agent can be viewed as a large tuple consisting of the fields of the AILAgent class. Writing out every element of this tuple makes the presentation largely unreadable, therefore we restrict outselves to including only those elements of the tuple that are relevant to the rule itself. These can be identified by the naming conventions shown below and, where there is a possibility of ambiguity we will indicate these with equalities – i.e. $i = (a, \varepsilon)$ indicates that the current intention is (a, ε). Where a value is changed by the transition it will often be primed to indicate the new value - e.g. $i' = null$ shows that the new value of the current intention is $null$.

ag	The name of the agent.
B	The agent's Belief base.
i	The current intention.
I	The agent's other intentions.
Pl	The agent's applicable plans.
A	The agent's queue of pending actions.
In	The agent's inbox.
Out	The agent's outbox.
ξ	The agent's environment.

Many of the operational rules make a check on a deed to see what type it is (e.g. the addition of a belief, the deletion of a goal). We represent these checks implicitly using notatation as follows:

a	An AIL data structure of action type.
b	An AIL data structure of belief type.
$+b$	A belief addition.
$-b$	A belief removal.
$b\{source\}$	A belief, from source $source$.
$!_\tau g$	A goal of type τ.
$+!_\tau g$	A goal addition.
$-!_\tau g$	A goal drop.
$\times!_\tau g$	A goal there is a problem with.
`lock`	An AIL lock structure.
`unlock`	An AIL unlock structure.
$\uparrow^{ag} m$	A message m sent to ag.
$\downarrow^{ag} m$	A message m received from ag.
$\top$	An AIL structure who's logical content is trivially true.
ε	A special marker indicating that some event has no plan yet.

Many of the rules make reference to methods that exist in the AILAgent class. Obviously subclassing these methods potentially changes the semantics of the rule. This is intentionally the case.

Notation	Method Name	Description
allsuspended	allintentionssuspended	All the intentions in the agent are suspended.
consistent(b)	consistent(b)	b is consistent with the belief base (defaults to true).
appPlans(i)	appPlans(i))	Generate application plans
filter(Pl)	filterPlans(Pl)	Filter plans from the set (defaults to none).
$ag \models gu, \theta$	θ = logicalconsequence(gu)	The agent believes the guard gu given unifier θ. .
relevant(s, s')	relevant(s, s')	s and s' are sources of information relevant to each other (defaults to true).
$\mathscr{S}_{\text{int}}(I)$	selectIntention(I)	Select an intention from the set I.
$\mathscr{S}_{\text{plan}}(Pl)$	selectPlan(Pl)	Select a plan from the set Pl.

Rules that specifically deal with accessing information from the environment, ξ, refence methods specificed in the AILEnv interface that have to be implemented by any environment. Again this means the semantics of the rules will depend upon the environment used.

Notation	Method Name	Description
ξ.**do**(a)	executeAction	Executing an action in the environment.
ξ.**getMessages**()	getMessages	Returns new messages.
ξ.**Percepts**(ag)	getPercepts	Returns new perceptions.

Many of the rules also manipulate, or check information about single intentions. Again these reference methods in the Intention class.

Notation	Method Name	Description
$\mathtt{U}_\theta$	compose	Compose the unifier with the top unifier on the intention.
$\mathtt{drop}(g)$	dropGoal	Drop all rows from the intention until one is reached with $+!_\tau g$ as it's event.
$\mathtt{drop}_\mathtt{p}(N,i)$	dropP	Drop N rows from the top of the intention.
$\mathtt{empty}(i)$	empty	The deed stack of the intention is empty.
$\mathtt{events}(i)$	events	The set/stack of events associated with the intention.
$\mathtt{hd}_d(i)$	hdD	The top deed on the intention.
$\mathtt{hd}_e(i)$	hdE	The top event on the intention.
$\mathtt{hd}_g(i)$	hdG	The top guard on the intention.
$\theta^{\mathtt{hd}(i)}$	hdU	The top unifier on the itnention.
@	iConcat	Add a new event, deed stack, guard stack and unifier to the top of the inention.
$;_p$	iCons	Add a new event, deed, guard and unifier as the top row of the inention.
$\mathtt{new}(e)$	Intention	Create a new intention from the event e.
$\mathtt{new}(e, source)$	Intention	Create a new intention from the event e and the source $source$..
$\mathtt{new}(e, Gs, Ds, \theta, source)$	Intention	Create a new intention from an event, guard stack, deed stack, unifier and source..
$\mathtt{lock}(i)$	lock	Mark the intention as locked.
$\mathtt{locked}(i)$	locked	The intention is locked.
$\mathtt{noplan}(i)$	noplan	The intention has not been planned (i.e. the deed stack is empty or contains only the ε "no plan yet" deed).
$\mathtt{suspend}(i)$	suspend	Mark the intention as suspended.
$\mathtt{tl}_i(i)$	tlI	Drop the top deed (with associated event, guard and unifier) from the intention.
$\mathtt{unlock}(i)$	unlock	Mark the intention as unlocked.

Lastly we use a few functions as shorthand for more complex processes with the AIL toolkit.

$oldPercepts()(P)$	Any beliefs in an agent's belief base which are marked as percepts (i.e. their source is percept) which are not in the set P.
$wake(ag)$	Unset any flags telling the agent to sleep it's thread next opportunity.
$unify(l_1, l_2)$	Unify the two Unifiable structures l_1 and l_2. This generally creates an empty unifier and then calls its unifies method.
$unsuspend(I)$	Unsuspend all the intentions in I.
$\tau_a(a)$	Returns the "type" of action, a. Useful when a semantics wants to separate actions into categories and treat them differently.
$d\theta$	Represents the application of a unifier, θ, to some data structure, d. d may be an action, a belief, a goal, a message, an event, a guard or a deed.

A.2 The Rules

ApplyApplicablePlans

$$\frac{Pl \neq \emptyset \quad \langle 0, e, g; G, Ds, \theta \rangle = \mathscr{S}_{\text{plan}}(Pl) \quad g \neq \top}{\langle i, Pl \rangle \rightarrow \langle i' = \texttt{new}(+state(g), \top; G, Ds, \theta, \texttt{self}), Pl' = \emptyset \rangle} \tag{9}$$

$$\frac{Pl \neq \emptyset \quad \langle N, e, g; G, Ds, \theta \rangle = \mathscr{S}_{\text{plan}}(Pl) \quad N > 0 \vee g = \top}{\langle i, Pl \rangle \rightarrow \langle i' = (e, g; G, Ds, \theta) \mathbin{@} \texttt{drop}_{\text{p}}(N, i), Pl' = \emptyset \rangle} \tag{10}$$

Notes: This rule selects a plan from the agent's applicable plans. The plan is represented as a tuple of the number of rows to be dropped, the trigger event, the plan's guard stack, deed stack and unifier.

If it is a reactive plan then N is equal to 0. In this case a new trigger is created $+state(g)$ where g is the top guard on the plan's guardstack. This is supposed to represent the state of the world that triggered the plan. A new intention is created from the applicable plan.

Otherwise the applicable plan is "glued" to the top of the current intention.

DirectPerception

$$\frac{P = \xi.\textbf{Percepts}(ag)}{\begin{array}{c}\langle ag, B, In \rangle \rightarrow \\ wake(\langle ag, B' = B \cup P \backslash oldPercepts()(P), In' = In \cup \xi.\textbf{getMessages}() \rangle)\end{array}} \tag{11}$$

Notes: A simple perception rule. It adds all percepts to the belief base and removes all beliefs no longer perceived. It also add all messages to the inbox. A key part of the working of the rule depends on AIL's annotation of all beliefs in the belief base with a source and its use of a special annotation for beliefs whose source is perception.

DirectPerceptionwEvent

$$\frac{\begin{array}{c}P = \xi.\textbf{Percepts}(ag) \quad P^{-} = oldPercepts()(P) \\ I_1 = \{\texttt{new}(+b) | b \in P\} \quad I_2 = \{\texttt{new}(-b) | b \in P^{-}\}\end{array}}{\begin{array}{c}\langle ag, B, I, In \rangle \rightarrow \\ wake(\langle ag, I' = I \cup I_1 \cup I_2, B' = B \cup P \backslash P^{-}, In' = In \cup \xi.\textbf{getMessages}() \rangle)\end{array}} \tag{12}$$

Notes: Similar to DirectPerception this rule also creates new intentions triggered by the addition (or removal) of all the beliefs allowing the agent to react to the changes.

DoNothing

$$\frac{}{A \rightarrow A} \tag{13}$$

Notes: The `DoNothing` rule, as its name suggests, makes no changes to the state of the agent. This is intended as a default rule that can be used in a reasoning cycle stage to do nothing if none of the other rules in the stage apply but nevertheless still allow the stage to progress.

DropIntention

$$\frac{I \neq \emptyset \quad \mathscr{S}_{\text{int}}(I) = (i', I')}{\langle ag, i, I \rangle \rightarrow \langle ag, i', I' \rangle} \tag{14}$$

Notes: `DropIntention` is really intended for sub-classing. It simply drops the current intention, i, and selects a new one from the intention set. A sub-class would be expected to place extra conditions on i to make sure they are only dropped in very specific circumstances. See `DropIntentionIfEmpty` for an example.

DropIntentionIfEmpty

$$\frac{i \neq null \quad \texttt{empty}(i) \quad I \neq \emptyset \quad \mathscr{S}_{\text{int}} I = (i', I')}{\langle ag, i, I \rangle \rightarrow \langle ag, i', I' \rangle} \tag{15}$$

Notes: `DropIntentionIfEmpty` drops the intention i if it is empty and selects a new current intention from the intention set. The additional $i \neq null$ is necessary since a few rules can leave the agent state with no current intention.

GenerateApplicablePlans

$$\frac{}{\langle ag, i, Pl \rangle \rightarrow \langle ag, i, Pl' = \textbf{filter}(\textbf{appPlans}(i)) \rangle} \tag{16}$$

Notes: This rule considers all the plans in the agent's plan library and examines all possible instantions of these plans, if there is more than one. It converts these instantiated plans to Applicable Plans, filters them according to any language specific heuristics (as defined by over-riding of the filterPlans method), and places them in the agents Applicable Plan list. The rule is primarily indended for subclassing by more sophisticated rules.

GenerateApplicablePlansEmpty

$$\frac{\textbf{filter}(\textbf{appPlans}(i)) = \emptyset \quad \texttt{noplan}(i)}{\langle ag, i, Pl \rangle \rightarrow \langle ag, i, Pl' = [(1, \texttt{hd}_e(i), [], [], \emptyset)] \rangle} \tag{17}$$

$$\frac{\textbf{filter}(\textbf{appPlans}(i)) = \emptyset \quad \neg\texttt{noplan}(i)}{\langle ag, i, Pl \rangle \rightarrow \langle ag, i, Pl' = [(0, \texttt{hd}_e(i), [], [], \emptyset)] \rangle} \tag{18}$$

Notes: This is a special case of the GenerateApplicablePlans rule for when the set of applicable plans is empty. It does two different things depending on whether or not the current intention has a plan at the top. If it does then the rule provides an applicable plan that will leave the intention unchanged allowing the plan to continue processing. If not it provides a plan that will drop the top row of the intention (for instance if the intention indicates a belief change event then the absense of a plan means that the system has no need to respond to that event. This rule will thus cause that belief change notification to be dropped).

GenerateApplicablePlansEmptyProblemGoal

$$\frac{\textbf{filter}(\textbf{appPlans}(i)) = \emptyset \quad \texttt{noplan}(i) \quad \texttt{hd}_e(i) = +!_\tau g}{\langle ag, i, Pl \rangle \rightarrow \langle ag, i, Pl' = [0, \texttt{x}!_\tau g, [\varepsilon], [\top], \theta^{\texttt{hd}(()}i))] \rangle} \tag{19}$$

$$\frac{\textbf{filter}(\textbf{appPlans}(i)) = \emptyset \quad \texttt{noplan}(i) \quad \neg\texttt{hd}_e(i) = g}{\langle ag, i, Pl \rangle \rightarrow \langle ag, i, Pl' = [(1, \texttt{hd}_e(i), [], [], \emptyset)] \rangle} \tag{20}$$

$$\frac{\textbf{filter}(\textbf{appPlans}(i)) = \emptyset \quad \neg\texttt{noplan}(i) \quad \neg\texttt{hd}_e(i) = g}{\langle ag, i, Pl \rangle \rightarrow \langle ag, i, Pl' = [(0, \texttt{hd}_e(i), [], [], \emptyset)] \rangle} \tag{21}$$

Notes: This is a further specialisation of GenerateApplicablePlansEmpty. In this case if the current intention has no plan and the trigger event at the top of the intention is a goal then, instead of simply dropping that row (as GenerateApplicablePlansEmpty does) it adds a new row triggered by a "problem goal" event. The agent can then react to that problem goal if it has an appropriate plan..

GenerateApplicablePlansIfNonEmpty

$$\frac{\textbf{filter}(\textbf{appPlans}(i)) \neq \emptyset}{\langle ag, i, Pl\rangle \rightarrow \langle ag, i, Pl' = \textbf{filter}(\textbf{appPlans}(i))\rangle} \tag{22}$$

Notes: As `GenerateApplicablePlans` except with a check for non-emptiness.

HandleAction

$$\frac{ag \models \mathtt{hd}_g(i)\theta^{\mathtt{hd}(i)}, \theta_b \quad \mathtt{hd}_d(i)\theta^{\mathtt{hd}(i)} = a \quad \xi.\textbf{do}(a\theta_b) = \theta_a}{\langle ag, i\rangle \rightarrow \langle ag, i' = \mathtt{tl}_i(i)\mathtt{U}_\theta(\theta^{\mathtt{hd}(i)} \cup \theta_a)\rangle} \tag{23}$$

$$\frac{ag \models \mathtt{hd}_g(i)\theta^{\mathtt{hd}(i)}, \theta_b \quad \mathtt{hd}_d(i)\theta^{\mathtt{hd}(i)} = a \quad \neg\xi.\textbf{do}(a\theta_b)}{\langle ag, i\rangle \rightarrow \langle ag, i' = \mathtt{tl}_i(i)\mathtt{U}_\theta\theta^{\mathtt{hd}(i)}\rangle} \tag{24}$$

Notes: This is a basic action handling rule. It attempts to execute the action in the environment (**do**). If this succeeds it gets a unifier which is handed back to the intention. If the action fails it is simply ignored. Most languages will want to explicitly handle action failure in some way, possibly by sub-classing this rule.

HandleActionwProblem

$$\frac{ag \models \mathtt{hd}_g(i)\theta^{\mathtt{hd}(i)}, \theta_b \quad \mathtt{hd}_d(i)\theta^{\mathtt{hd}(i)} = a \quad \xi.\textbf{do}(a\theta_b) = \theta_a}{\langle ag, i\rangle \rightarrow \langle ag, i' = \mathtt{tl}_i(i)\mathtt{U}_\theta(\theta^{\mathtt{hd}(i)} \cup \theta_a)\rangle} \tag{25}$$

$$\frac{ag \models \mathtt{hd}_g(i)\theta^{\mathtt{hd}(i)}, \theta_b \quad \mathtt{hd}_d(i)\theta^{\mathtt{hd}(i)} = a \quad \neg\xi.\textbf{do}(a\theta_b) \quad \mathtt{hd}_e(i) = +!_{\tau_g}g}{\langle ag, i\rangle \rightarrow \langle ag, i' = (\mathtt{x}!_{\tau_g}g, \theta^{\mathtt{hd}(i)} \cup \theta_a);_p i\rangle} \tag{26}$$

Notes: This extends `HandleAction` with some failure handling. If the action appears on the deed stack because of a goal commitment then the intention gets a new intention noting there is a problem with the goal. The agent can then react to this – e.g. by attempting the action again or dropping the goal or in some other fashion.

HandleAddAchieveTestGoal

$$\frac{ag \models \mathtt{hd}_g(i)\theta^{\mathtt{hd}(i)}, \theta_b \quad \mathtt{hd}_d(i)\theta^{\mathtt{hd}(i)} = +!_{\tau_g}g \quad \tau_g = a \vee \tau_g = t \quad ag \models g\theta_b, \theta_g}{\langle ag, i\rangle \rightarrow \langle ag, i' = \mathtt{tl}_i(i)\mathtt{U}_\theta(\theta^{\mathtt{hd}(i)} \cup \theta_a \cup \theta_g)\rangle} \tag{27}$$

$$\frac{ag \models \mathtt{hd}_g(i)\theta^{\mathtt{hd}(i)}, \theta_b \quad \mathtt{hd}_d(i)\theta^{\mathtt{hd}(i)} = +!_{\tau_g}g \quad \tau_g = a \vee \tau_g = t \quad ag \not\models g\theta_b}{\langle ag, i\rangle \rightarrow \langle ag, i' = (g\tau_g, \theta^{\mathtt{hd}(\cup)}\theta_b);_p i\rangle} \tag{28}$$

Notes: This rule handles the commitment to an achieve or test goal. These are detected in the condition $\tau_g = a \vee \tau_g = t$. An achieve or test goal is one that triggers a plan if it not already believed but does no more than set a unifier if it is. If it is to trigger a plan then we register the commitment to planning the goal as an event on the top of the intention stack.

HandleAddBelief

$$\frac{ag \models \mathtt{hd}_g(i)\theta^{\mathtt{hd}(i)}, \theta_b \quad \mathtt{hd}_d(i)\theta^{\mathtt{hd}(i)} = +b \quad \textbf{consistent}(B, b)}{\begin{array}{c}\langle ag, B, i, I\rangle \rightarrow \\ \langle ag, B' = B \cup b\{\mathtt{src}(i)\}, i' = \mathtt{tl}_i(i)\mathtt{U}_\theta(\theta^{\mathtt{hd}(i)} \cup \theta_b), I' = \mathit{unsuspend}(I)\rangle\end{array}} \tag{29}$$

Notes: A Basic rule for adding a new belief to the belief base. It assigns a source to the belief, which is the source of the intention. As a side effect it "unsuspends" all intentions.

HandleAddBeliefwEvent

$$\frac{ag \models \mathtt{hd}_g(i)\theta^{\mathtt{hd}(i)}, \theta_b \quad \mathtt{hd}_d(i)\theta^{\mathtt{hd}(i)} \cup \theta_b = +b \quad \textbf{consistent}(B,b)}{\begin{array}{c}\langle ag, B, i, I\rangle \rightarrow \\ \langle ag, B' = B \cup b\{\mathtt{src}(i)\}, \\ i' = \mathtt{tl}_i(i)\mathtt{U}_\theta(\theta^{\mathtt{hd}(i)} \cup \theta_b), I' = \mathit{unsuspend}(I) \cup \mathtt{new}(+b, \mathtt{src}(\mathtt{self}))\rangle\end{array}} \tag{30}$$

Notes: A modification of the basic add belief rule which also generates an event noting the belief change. Note here that the new intention is given the source "self" not the source of the original intention - this is because any further changes triggered by the belief change are dependent on the agent's internal reasoning and not on the original source of the belief change.

HandleAddGoal (Abstract)

$$\frac{ag \models \mathtt{hd}_g(i)\theta^{\mathtt{hd}(i)}, \theta_b \quad \mathtt{hd}_d(i)\theta^{\mathtt{hd}(i)} \cup \theta_b = +!_{\tau_g} g}{\text{undefined}} \tag{31}$$

Notes: Sets up the necessary preconditions for handling an event involving the addition of a goal but doesn't define any transition.

HandleAddPerformGoal

$$\frac{ag \models \mathtt{hd}_g(i)\theta^{\mathtt{hd}(i)}, \theta_b \quad \mathtt{hd}_d(i)\theta^{\mathtt{hd}(i)} \cup \theta_b = +!_{\tau_g} g}{\langle ag, i\rangle \rightarrow \langle ag, i' = (+!_{\tau_g} g, \theta^{\mathtt{hd}(i)} \cup \theta_b);_p (+!\mathtt{hd}_e, \mathit{null}, \theta^{\mathtt{hd}(i)});_p i\rangle} \tag{32}$$

Notes: Commit to a perform goal. Unlike `HandleAddAchieveTestGoal` there are no checks for whether we believe the goal. We leave a null action on the stack though so we don't loose track of the previous event (which can happen if that event was planned with only one perform goal).

HandleBelief (Abstract)

$$\frac{ag \models \mathtt{hd}_g(i)\theta^{\mathtt{hd}(i)}, \theta_b \quad \mathtt{hd}_d(i)\theta^{\mathtt{hd}(i)} \cup \theta_b = ?b}{\text{undefined}} \tag{33}$$

Notes: Sets up the necessary preconditions for handling a belief (either addition or deletion) but doesn't define the transition. The preconditions are that the top guard on the intention is believed and that the top deed is off belief type. We use ? here to show that the abstract rule does not differentiate between adding and removing a belief.

HandleDelayedAction

$$\frac{ag \models \mathtt{hd}_g(i)\theta^{\mathtt{hd}(i)}, \theta_b \quad \mathtt{hd}_d(i)\theta^{\mathtt{hd}(i)} = a}{\langle ag, i, A\rangle \rightarrow \langle ag, i' = \mathtt{tl}_i(i)\mathtt{U}_\theta(\theta^{\mathtt{hd}(i)} \cup \theta_b), A' = A; a\rangle} \tag{34}$$

Notes: This rule is intended for use in languages which use an action queue to store actions as they are processed, but delay their actual execution in the environment until some later point. This rule is untested.

HandleDropBelief

$$\frac{\begin{array}{c}ag \models \mathtt{hd}_g(i)\theta^{\mathtt{hd}(i)}, \theta_b \quad \mathtt{hd}_d(i)\theta^{\mathtt{hd}(i)} = -b \\ B^1 = \{b' | b' \in B \wedge \mathit{unify}(b', b) \wedge \textbf{relevant}(\mathtt{src}(b'), \mathtt{src}(b))\}\end{array}}{\langle ag, B, i, I\rangle \rightarrow \langle ag, B' = B \backslash B^1, i' = \mathtt{tl}_i(i)\mathtt{U}_\theta(\theta^{\mathtt{hd}(i)} \cup \theta_b), I\rangle} \tag{35}$$

Notes: This is the basic rule for dropping beliefs from the belief base. If the deed, $-b$ is on top of the current intention all beliefs that unify with b are removed from the belief base.

HandleDropBeliefwEvent

$$\frac{\begin{array}{c} ag \models \mathtt{hd}_g(i)\theta^{\mathtt{hd}(i)}, \theta_b \quad \mathtt{hd}_d(i)\theta^{\mathtt{hd}(i)} = -b \\ B^1 = \{b' | b' \in B \wedge \mathit{unify}(b', b) \wedge \mathbf{relevant}(\mathtt{src}(b'), \mathtt{src}(b))\} \end{array}}{\begin{array}{c} \langle ag, B, i, I\rangle \rightarrow \\ \mathit{wake}(\langle ag, B' = B \backslash B^1, i' = \mathtt{tl}_i(i)\mathtt{U}_\theta(\theta^{\mathtt{hd}(i)} \cup \theta_b), I' = I \cup \mathtt{new}(-b, \mathtt{src}(i))\rangle) \end{array}} \tag{36}$$

Notes: This rule drops a belief from the belief base (providing the source of the instruction to drop the belief is deemed "relevant" to the source of the belief). At the same time it generates a new intention containing the event that the belief has been dropped. Appropriate handling of this event can allow the agent to form plans in reaction to it.

HandleDropGeneralGoal

$$\frac{\begin{array}{c} ag \models \mathtt{hd}_g(i)\theta^{\mathtt{hd}(i)}, \theta_b \quad \mathtt{hd}_d(i)\theta^{\mathtt{hd}(i)} \cup \theta_b = -!_{\tau_g} g \quad \tau_g \notin \mathit{Exclusions} \\ \exists e \in \mathtt{events}(i).\mathit{unify}(e, +!_{\tau_g} g) \end{array}}{\langle ag, i\rangle \rightarrow \langle ag, i' = i.\mathtt{drop}_E(e)\rangle} \tag{37}$$

$$\frac{\begin{array}{c} ag \models \mathtt{hd}_g(i)\theta^{\mathtt{hd}(i)}, \theta_b \quad \mathtt{hd}_d(i)\theta^{\mathtt{hd}(i)} \cup \theta_b = -!_{\tau_g} g \quad \tau_g \notin \mathit{Exclusions} \\ \neg\exists e \in \mathtt{events}(i).\mathit{unify}(e, +!_{\tau_g} g) \quad I_1 = \{i' | i' \in I \wedge \exists e' \in i'.\mathit{unify}(e', +!_{\tau_g} g)\} \end{array}}{\langle ag, I\rangle \rightarrow \langle ag, I' = I \cup \{i'' | \exists i' \in I_1 \wedge i'' = i'.\mathtt{drop}_E(e)\} \backslash I_1\rangle} \tag{38}$$

Notes: This rule can be parameterised by a set of excluded goal types (*Exclusions*) above. This allows it to act as a default rule for handling "drop goal" instructions. It does this by searching the current intention for the most recent add goal event that unifies with the goal to be dropped and then deletes all rows on the intention above that. $\mathtt{drop}_E$ isn't a built-in AIL function but is created by recursing through the intention's events. The second rule drops the event from all the intentions that contain it, assuming the goal doesn't occur in the current intention.

HandleDropGoal (Abstract)

$$\frac{ag \models \mathtt{hd}_g(i)\theta^{\mathtt{hd}(i)}, \theta_b \quad \mathtt{hd}_d(i)\theta^{\mathtt{hd}(i)} \cup \theta_b = -!_{\tau_g} g}{\text{undefined}} \tag{39}$$

Notes: Sets up the necessary preconditions for handling a drop goal event but doesn't define the transition.

HandleEmptyDeedStack

$$\frac{\mathtt{empty}(i)}{\langle ag, i\rangle \rightarrow \langle ag, i\rangle} \tag{40}$$

Notes: Does nothing if the current intention's deed stack is empty.

HandleGeneralAction

$$\frac{ag \models \mathtt{hd}_g(i)\theta^{\mathtt{hd}(i)}, \theta_b \quad \mathtt{hd}_d(i)\theta^{\mathtt{hd}(i)} = a \quad \xi.\mathbf{do}(a\theta_b) = \theta_a \quad \tau_a(a) \notin Excluded}{\langle ag, i\rangle \rightarrow \langle ag, i' = \mathtt{tl}_i(i)\mathtt{U}_\theta(\theta^{\mathtt{hd}(i)} \cup \theta_a)\rangle} \tag{41}$$

$$\frac{\begin{array}{c} ag \models \mathtt{hd}_g(i)\theta^{\mathtt{hd}(i)}, \theta_b \quad \mathtt{hd}_d(i)\theta^{\mathtt{hd}(i)} = a \quad \neg\xi.\mathbf{do}(a\theta_b) \quad \mathtt{hd}_e(i) = +!_{\tau_g} g \\ \tau_a(a) \notin Excluded \end{array}}{\langle ag, i\rangle \rightarrow \langle ag, i' = \mathtt{new}(\mathtt{x}!_{\tau_g} g, \theta^{\mathtt{hd}(i)} \cup \theta_a);_p i)\rangle} \tag{42}$$

$$\frac{\begin{array}{c} ag \models \mathtt{hd}_g(i)\theta^{\mathtt{hd}(i)}, \theta_b \quad \mathtt{hd}_d(i)\theta^{\mathtt{hd}(i)} = a \quad \neg\xi.\mathbf{do}(a\theta_b) \quad \neg\mathtt{hd}_e(i) = +!_{\tau_g} g \\ \tau_a(a) \notin Excluded \end{array}}{\langle ag, i\rangle \rightarrow \langle ag, i' = \mathtt{tl}_i(i)\mathtt{U}_\theta\theta^{\mathtt{hd}(i)}\rangle} \tag{43}$$

Notes: HandleGeneralAction extends HandleActionwProblem. The rule is parameterized by a set of action types that are "Excluded" – for instance the GWENDOLEN implementation handles "send" actions differently to other types of action so these are excluded from consideration by this particular rule.

HandleGeneralDelayedAction

$$\frac{ag \models \mathtt{hd}_g(i)\theta^{\mathtt{hd}(i)}, \theta_b \quad \mathtt{hd}_d(i)\theta^{\mathtt{hd}(i)} = a \quad \tau_a(a) \notin Excluded}{\langle ag, i, A\rangle \rightarrow \langle ag, i' = \mathtt{tl}_i(i)\mathtt{U}_\theta(\theta^{\mathtt{hd}(i)} \cup \theta_b), A' = A; a\rangle} \tag{44}$$

Notes: This extends HandleDelayedAction in the same way that HandleGeneralAction extends HandleActionwProblem. That is the rule can be parameterized with a list of "Excluded" action types which are to be handled by alternative rules.

HandleGoal (Abstract)

$$\frac{ag \models \mathtt{hd}_g(i)\theta^{\mathtt{hd}(i)}, \theta_b \quad \mathtt{hd}_d(i)\theta^{\mathtt{hd}(i)} \cup \theta_b = ?!_{\tau_g} g}{\text{undefined}} \tag{45}$$

Notes: Sets up the necessary preconditions for handling an event involving a goal but doesn't define any transition.

HandleGuardNotSatisfied

$$\frac{ag \not\models \mathtt{hd}_g(i)\theta^{\mathtt{hd}(i)}}{A \rightarrow A} \tag{46}$$

Notes: The agent does nothing if the guard on an intention can not be satisfied. Should be used with caution since it can cause programs to loop.

HandleLockUnlock

$$\frac{ag \models \mathtt{hd}_g(i)\theta^{\mathtt{hd}(i)}, \theta_b \quad \mathtt{hd}_d(i)\theta^{\mathtt{hd}(i)} \cup \theta_b = \mathtt{lock}}{\langle ag, i\rangle \rightarrow \langle ag, i' = \mathtt{lock}(\mathtt{tl}_i(i)\mathtt{U}_\theta(\theta^{\mathtt{hd}(i)} \cup \theta_b))\rangle} \tag{47}$$

$$\frac{ag \models \mathtt{hd}_g(i)\theta^{\mathtt{hd}(i)}, \theta_b \quad \mathtt{hd}_d(i)\theta^{\mathtt{hd}(i)} \cup \theta_b = \mathtt{unlock}}{\langle ag, i\rangle \rightarrow \langle ag, i' = \mathtt{unlock}(\mathtt{tl}_i(i)\mathtt{U}_\theta(\theta^{\mathtt{hd}(i)} \cup \theta_b))\rangle} \tag{48}$$

Notes: This allows an intention to be "locked" as the current intention, for instance to allow a complete sequence of belief changes be processed before any other reasoning takes place. Once finished the intention has to be unlocked. See the SelectIntention rule to see how this works. In languages without the lock construct locking isn't used, obviously, and so doesn't affect intention seletion..

HandleMessages

$$\frac{}{\begin{array}{c}\langle ag, I, In\rangle \rightarrow \\ \langle ag, I' = I \cup \{\texttt{new}(+received(m), \texttt{src}(ag)) \mid \downarrow^{ag} m \in In, In' = []\}\rangle\end{array}} \tag{49}$$

Notes: This rule does not poll the environement for messsages. It takes all messages currently in an agent's inbox and converts them to intentions (triggered by a perception that the message has been recieved), emptying the inbox in the process. It should be noted that it does not store the message anywhere once the inbox is emptied. It assumes that some plan will act appropriately to the message received event. If this does not happen then the message content may be lost.

HandleNull

$$\frac{ag \models \texttt{hd}_g(i)\theta^{\texttt{hd}(i)}, \theta_b \quad \texttt{hd}_d(i)\theta^{\texttt{hd}(i)} \cup \theta_b = null}{\langle ag, i\rangle \rightarrow \langle ag, i' = \texttt{tl}_i(i)\texttt{U}_\theta(\theta^{\texttt{hd}(i)} \cup \theta_b)\rangle} \tag{50}$$

Notes: The null action is used as a place holder to preserve, in some situations, a record of an event in an intention stack. This rule simply ignores the null action when it is encountered.

HandleSendAction

$$\frac{ag \models \texttt{hd}_g(i)\theta^{\texttt{hd}(i)}, \theta_b \quad \texttt{hd}_d(i)\theta^{\texttt{hd}(i)} = \uparrow^{ag} m \quad \xi.\textbf{do}((\uparrow^{ag} m)\theta_b) = \theta_a}{\begin{array}{c}\langle ag, i, I, Out\rangle \rightarrow \\ \langle ag, i' = \texttt{tl}_i(i)\texttt{U}_\theta(\theta^{\texttt{hd}(i)} \cup \theta_a), I'I \cup \{\texttt{new}(+\uparrow^{ag} m, \texttt{src}(\texttt{self}))\}, \\ Out' = Out \cup \{m\}\rangle\end{array}} \tag{51}$$

$$\frac{ag \models \texttt{hd}_g(i)\theta^{\texttt{hd}(i)}, \theta_b \quad \texttt{hd}_d(i)\theta^{\texttt{hd}(i)} = \uparrow^{ag} m \quad \neg\xi.\textbf{do}((\uparrow^{ag} m)\theta_b) \quad \texttt{hd}_e(i) = +!_{\tau_g} g}{\begin{array}{c}\langle ag, i, I, Out\rangle \rightarrow \\ \langle ag, i' = (\texttt{x}!_{\tau_g} g, \theta^{\texttt{hd}(i)} \cup \theta_b);_p i, I' = I \cup \{\texttt{new}(+\uparrow^{ag} m, \texttt{src}(\texttt{self}))\}, \\ Out' = Out \cup \{m\}\rangle\end{array}} \tag{52}$$

Notes: This rule is implemented as an extension of HandleActionwProblem. As well as executing a send action it also adds a record the message has been sent to the outbox and generates an intention from the message sending event.

HandleTopDeed (Abstract)

$$\frac{ag \models \texttt{hd}_g(i)\theta^{\texttt{hd}(i)}, \theta_b}{\text{undefined}} \tag{53}$$

Notes: Sets up the necessary preconditions for handling the top of the deed stack but doesn't define the transition.

HandleWaitFor

$$\frac{\begin{array}{c}ag \models \texttt{hd}_g(i)\theta^{\texttt{hd}(i)}, \theta_b \quad \texttt{hd}_d(i)\theta^{\texttt{hd}(i)} = *b \\ P = \xi.\textbf{Percepts}(ag) \quad OP = oldPercepts()(P) \quad b \in B \cup P \backslash OP\end{array}}{\begin{array}{c}\langle ag, B, i, I, In\rangle \rightarrow \\ wake(\langle ag, B' = B \cup P \backslash OP, i' = \texttt{tl}_i(i)\texttt{U}_\theta(\theta^{\texttt{hd}(i)} \cup \theta_b), \\ I' = I \cup \{\texttt{new}(+b') \mid b' \in P \wedge b \notin B\} \cup \{\texttt{new}(-b') \mid b' \in OP\}, \\ In' = In \cup \xi.\textbf{getMessages}()\rangle)\end{array}} \tag{54}$$

$$\frac{\begin{array}{c} ag \models \mathtt{hd}_g(i)\theta^{\mathtt{hd}(i)}, \theta_b \quad \mathtt{hd}_d(i)\theta^{\mathtt{hd}(i)} = *b \\ P = \xi.\mathbf{Percepts}(ag) \quad OP = oldPercepts()(P) \quad b \notin B \cup P \backslash OP \\ (\neg \mathbf{allsuspended} \vee \mathbf{P} \neq \emptyset) \end{array}}{\begin{array}{c} \langle ag, B, i, I, In\rangle \rightarrow \\ wake(\langle ag, B' = B \cup P \backslash OP, i' = \mathtt{suspend}(i) \\ I' = I \cup \{\mathtt{new}(+b') \mid b' \in P \wedge b \notin B\} \cup \{\mathtt{new}(-b') \mid b' \in OP\} \\ In' = In \cup \xi.\mathbf{getMessages}()\rangle) \end{array}} \tag{55}$$

$$\frac{\begin{array}{c} ag \models \mathtt{hd}_g(i)\theta^{\mathtt{hd}(i)}, \theta_b \quad \mathtt{hd}_d(i)\theta^{\mathtt{hd}(i)} = *b \\ b \notin B \quad \mathbf{allsuspended} \quad \xi.\mathbf{Percepts}(ag) = \emptyset \end{array}}{\begin{array}{c} \langle ag, B, In, i\rangle \rightarrow \\ \mathbf{sleep}(\langle ag, B, In \cup \xi.\mathbf{getMessages}(), \mathtt{suspend}(i)\}\rangle) \end{array}} \tag{56}$$

Notes: Because sleeping and waiting behaviour is critical to model checking with JPF we found it necessary to introduce new syntax which allows an intention to wait until the agent holds a certain belief. This syntax is $*b$. If the relevant belief is not held then that intention is suspended and if all intentions are suspended then the agent is told to sleep at the next opportunity. This rule for handling the waiting behaviour is actually implemented as an extension to perception and so makes the decision to suspend the intention based on the most recent information available to the agent. It should be noted that several other rules, e.g. ones that add new beliefs to the belief base automatically unsuspend all intentions allowing the wait for deed to be rechecked.

HandleWaitForDirect

$$\frac{\begin{array}{c} ag \models \mathtt{hd}_g(i)\theta^{\mathtt{hd}(i)}, \theta_b \quad \mathtt{hd}_d(i)\theta^{\mathtt{hd}(i)} = *b \\ P = \xi.\mathbf{Percepts}(ag) \quad OP = oldPercepts()(P) \quad b \in B \cup P \backslash OP \end{array}}{\begin{array}{c} \langle ag, B, i, In\rangle \rightarrow \\ wake(\langle ag, B' = B \cup P \backslash OP, i' = \mathtt{tl}_i(i)\mathtt{U}_\theta(\theta^{\mathtt{hd}(i)} \cup \theta_b), \\ In' = In \cup \xi.\mathbf{getMessages}()\rangle) \end{array}} \tag{57}$$

$$\frac{\begin{array}{c} ag \models \mathtt{hd}_g(i)\theta^{\mathtt{hd}(i)}, \theta_b \quad \mathtt{hd}_d(i)\theta^{\mathtt{hd}(i)} = *b \\ P = \xi.\mathbf{Percepts}(ag) \quad OP = oldPercepts()(P) \quad b \notin B \cup P \backslash OP \\ (\neg \mathbf{allsuspended} \vee \mathbf{P} \neq \emptyset) \end{array}}{\begin{array}{c} \langle ag, B, i, In\rangle \rightarrow \\ wake(\langle ag, B' = B \cup P \backslash OP, i' = \mathtt{suspend}(i) \\ In' = In \cup \xi.\mathbf{getMessages}()\rangle) \end{array}} \tag{58}$$

$$\frac{\begin{array}{c} ag \models \mathtt{hd}_g(i)\theta^{\mathtt{hd}(i)}, \theta_b \quad \mathtt{hd}_d(i)\theta^{\mathtt{hd}(i)} = *b \\ b \notin B \quad \mathbf{allsuspended} \quad \xi.\mathbf{Percepts}(ag) = \emptyset \end{array}}{\begin{array}{c} \langle ag, B, In, i\rangle \rightarrow \\ \mathbf{sleep}(\langle ag, B, In \cup \xi.\mathbf{getMessages}(), \mathtt{suspend}(i)\}\rangle) \end{array}} \tag{59}$$

Notes: This rule is essentially the same as HandleWaitFor. The only difference is that it sub-classes DirectPerception rather than Perceive. This means it does not create new intentions from incoming belief changes.

IgnoreUnplannedProblemGoal

$$\frac{\mathtt{hd}_e(i) = \mathtt{x!}_{\tau_g} g \quad \mathtt{hd}_d(i) = \varepsilon}{\langle ag, i\rangle \rightarrow \langle ag, i\rangle} \tag{60}$$

Notes: This rule ignores an unplanned problem goal. It simply does nothing but allows the reasoning cycle of the agent to continue processing on the assumption that planning of the goal may become possible later.

MatchDropGoal

$$\frac{\begin{array}{c}\texttt{deeds}(i) = [\varepsilon] \quad \texttt{hd}_e = \texttt{x!}_{\tau_g} g \\ I' = \{\langle i', \theta\rangle \mid i' \in I \wedge \exists +!g' \in \texttt{events}(i').\, \mathit{unify}(g', g) = \theta \wedge \texttt{src}(i') = \texttt{src}(i)\} \\ I'' = \{(-!_{\tau_g} g, \varepsilon, T, \theta);_p i \mid \langle i, \theta\rangle \in I'\} \quad \mathscr{S}_{\text{int}}(I \backslash I' \cup I'') = (i_1, I_1)\end{array}}{\langle ag, i, I\rangle \rightarrow \langle ag, i' = i_1, I' = I_1\rangle} \tag{61}$$

Notes: If an intention consists only of a single unplaned problem goal event then this should be matched with and placed as a row on the top of all intentions containing that goal. The intention is then removed and a new current intention selection. This is intended to assist in dealing with requests to drop goals that come from some external source.

Perceive

$$\frac{ag \models \texttt{hd}_g(i)\theta^{\texttt{hd}(i)}, \theta_b \quad P = \xi.\textbf{Percepts}(ag) \quad OP = \mathit{oldPercepts}()(P)}{\begin{array}{c}\langle ag, B, I, In\rangle \rightarrow \\ \mathit{wake}(\langle ag, B' = B \cup P \backslash OP, \\ I \cup \{\texttt{new}(+b) \mid b \in P) \wedge b \notin B\} \cup \{\texttt{new}(-b) \mid b \in OP\}, \\ In \cup \xi.\textbf{getMessages}()\rangle)\end{array}} \tag{62}$$

Notes: This is a complex perception rule. It adds all percepts to the belief base and removes all beliefs no longer perceived. It also add all messages to the inbox. It then creates new intentions, each triggered by the events of acquiring or losing one of the percepts. A key part of the working of the rule depends on AIL's annotation of all beliefs in the belief base with a source and its use of a special annotation for beliefs whose source is perception.

ProcessDelayedAction

$$\frac{\xi.\textbf{do}(a)}{\langle ag, A = a; A'\rangle \rightarrow \langle ag, A'\rangle} \tag{63}$$

Notes: This rule executes the top action on the agent's action queue. The assumption is that a (FIFO) queue of actions has been created which the agent will execute in order at some point. During planning actions are put on the queue but not actually executed. This rule is untested.

SelectIntention

$$\frac{\neg\texttt{empty}(i) \quad \neg\texttt{locked}(i) \quad \mathscr{S}_{\text{int}}(I \cup \{i\}) = (i', I') \quad \neg\textbf{allsuspended}}{< ag, i, I > \rightarrow < ag, i', I' >} \tag{64}$$

$$\frac{\neg\texttt{empty}(i) \quad \texttt{locked}(i) \quad \neg\textbf{allsuspended}}{< ag, i, I > \rightarrow < ag, i, I >} \tag{65}$$

Notes: This is the the basic rule for intention selection. It works by calling the agent method selectIntention ($\mathscr{S}_{\text{int}}$) which is expected to be a common candidate for over-riding. The only situation in which $\mathscr{S}_{\text{int}}$ is not called is if the current intention is locked in which case that intention is selected again.

SelectIntentionNotUnplannedProblemGoal

$$\frac{\neg\texttt{empty}(i) \quad \neg\texttt{locked}(i) \quad \mathscr{S}_{\text{int}}(I \cup \{i\}) = (i', I') \quad \texttt{hd}_e(i') \neq -!_{\tau_g} g \vee \neg\texttt{noplan}(i') \quad \neg\textbf{allsuspended}}{< ag, i, I > \rightarrow < ag, i', I' >} \tag{66}$$

$$\frac{\neg\texttt{empty}(i) \quad \texttt{locked}(i) \quad \texttt{hd}_e(i') \neq -!_{\tau_g} g \vee \neg\texttt{noplan}(i') \quad \neg\textbf{allsuspended}}{< ag, i, I > \rightarrow < ag, i, I >} \tag{67}$$

Notes: This rule extends SelectIntention with one additional condition which is that the top event on the selected intention is not a problem or drop goal event which has no associated plan. We assume that, where this rule is used, another rule (e.g. MatchDropGoal) is employed to handle these intentions.

SleepIfEmpty

$$\frac{(i = null \vee \texttt{empty}(i) \vee \textbf{is\_suspended}(i)) \wedge (I = \emptyset \vee \textbf{allsuspended})}{\langle ag, i, I \rangle \rightarrow \textbf{sleep}(\langle ag, i, I \rangle)} \tag{68}$$

Notes: This rule sleeps an agent thread in a controlled fashion if all its intention are empty or suspended. It is quite important to include this rule, or one similar to it, into a language semantics even if one isn't there. The JPF model checker does not assume a fair JAVA scheduling algorithm so unless a multi-agent system forces agents (and so their threads) to sleep it will investigate runs in which one agent executes continuously and none of the others do so. Note that this does not immediately sleep the thread. It sets a flag that the agent wishes to sleep. The agent controller then decides when this should happen.

B Sample GWENDOLEN Code for the Verified Scenarios

B.1 Notation

GWENDOLEN uses the AIL's plan mechanisms "off the shelf". It makes no use of prefix matching in plan execution and only has plans whose prefix is ε. Similarly, it is assumed that the guard stacks for plans are all $\top$ except for the very top guard that governs the plan applicability. Plans are therefore a triple of a triggering event (if relevant), a guard, and a body of deeds to be performed. We therefore represent GWENDOLEN's version of the plan

trigger	$+!_a$`clean()`
prefix	$[\varepsilon]$
guard	`dirty(Room)` $\top$
body	$+!_a$`goto(Room)` $+!_a$`vacuum(Room)`

that appeared in Section 3.1.8 as

$$+!_a clean() : dirty(Room) \;\; <- \;\; +!_a goto(Room); +!_a vacuum(Room) \tag{69}$$

GWENDOLEN agents also distinguish two sorts of goals. *Achievement goals*, $!_a g$, make statements about beliefs the agent wishes to hold. They remain goals until the agent gains an appropriate belief. *Perform goals*, $!_p g$, simply state a sequence of deeds to be performed and cease to be a goal as soon as that sequence is complete. When an agent takes an action, it executes code specific to that action in the environment. Typically, this code alters the set of propositions that agents are able to perceive. It may also cause messages to be added to an agent's inbox. Agents go through a specific perception phase when they check their beliefs against the environment's percepts and modify them accordingly. At this point, agents also handle the messages currently in their inbox.

$\uparrow (a,p,m)$ indicates the sending of a message m, with performative, p, to agent a, and $\downarrow (p,m)$ indicates the receipt of a message m with performative p. Since guards may also refer to sent and received messages, the syntax $\uparrow (a,p,m)$ is also used in plan guards, as is $\downarrow (p,m)$. In the following examples, the relevant performatives are **p** (for *perform* the message content) and **b** (for *believe* the message content).

$\mathscr{B}b$ is used in plan guards to indicate the condition that the agent believes b. The `lock`, `unlock`, and $*b$ deeds (see Section 3.1.7) appear as such in the code.

Throughout this section, we will use ',' to indicate concatenation of deeds on a stack. We follow the Prolog convention of representing variables as starting with upper-case letters, while constants start with lower-case letter. The belief rules used for Prolog-style reasoning in guards are represented in a Prolog style as `l1 :- l2, l3, l4;`

B.2 Contract Net Code

Action a makes fact g true in the environment. Similarly, action $a2$ makes the $g2$ true. These facts can then be perceived by all agents.

Code Example 2.1 Contract Net

```
:name: ag1                                                    1
                                                              2
: Initial Beliefs :                                           3
                                                              4
¬cando(g)                                                     5
¬cando(g2)                                                    6
ag(ag2)                                                       7
ag(ag3)                                                       8
my_name(ag1)                                                  9
                                                              10
: Initial Goals:                                              11
                                                              12
```

$!_a$ g
$!_a$ g2

:**Plans**:

$+!_a$ g : $\{\mathscr{B}$ cando(g)$\}$ $\leftarrow$ a;
$+!_a$ g2 : $\{\mathscr{B}$ cando(g2)$\}$ $\leftarrow$ a2;
$+!_a$ Gl : $\{\mathscr{B}$ $\neg$cando(Gl)$\}$ $\leftarrow$ $+!_p$ cfp(Gl);
$+!_p$ cfp(G) :
$\{\mathscr{B}$ ag(A1), $\mathscr{B}$ my_name(Name), $\neg\uparrow$ (A1, **p**, respond(G, Name)), $\neg\mathscr{B}$ proposal(Ag, G)$\}$
$\leftarrow$ $\uparrow$(A1, **p**, respond(G, Name));
$+!_p$ cfp(G1) : $\{\mathscr{B}$ proposal(A, G1)$\}$ $\leftarrow$ $*$G1;
+proposal(A, G4) : $\{\mathscr{B}$ $\neg$cando(G4), $\neg\mathscr{B}$ awarded(G4)$\}$ $\leftarrow$ $\uparrow$(A, **b**, award(G4)),
+awarded(G4);
$+\downarrow$(**p**, Ga) : $\{\top\}$ $\leftarrow$ $+!_p$ Ga;
$+\downarrow$(**b**, B): $\{\top\}$ $\leftarrow$ +B;
$+!_p$ respond(G2, Name) : $\{\mathscr{B}$ cando(G2), $\mathscr{B}$ my_name(A)$\}$ $\leftarrow$ $\uparrow$(Name, **b**, proposal(A, G2));
$+!_p$ respond(G3, Name) : $\{\mathscr{B}$ $\neg$cando(G3), $\mathscr{B}$ my_name(A)$\}$ $\leftarrow$ $\uparrow$(Name, **b**, sorry(A, G3));
+award(G5) : $\{\top\}$ $\leftarrow$ $+!_a$ G5 ;
$+!_p$ cfp(G6) : $\{\top\}$ $\leftarrow$ $*$G6;

:**name**: ag2

: **Initial Beliefs** :

my_name(ag2)
cando(g)
$\neg$cando(g2)

:**Plans**:

$+!_a$ g : $\{\mathscr{B}$ cando(g)$\}$ $\leftarrow$ a;
$+!_a$ g2 : $\{\mathscr{B}$ cando(g2)$\}$ $\leftarrow$ a2;
$+!_a$ Gl : $\{\mathscr{B}$ $\neg$cando(Gl)$\}$ $\leftarrow$ $+!_p$ cfp(Gl);
$+!_p$ cfp(G) :
$\{\mathscr{B}$ ag(A1), $\mathscr{B}$ my_name(Name), $\neg\uparrow$ (A1, **p**, respond(G, Name)), $\neg\mathscr{B}$ proposal(Ag, G)$\}$
$\leftarrow$ $\uparrow$(A1, **p**, respond(G, Name));
$+!_p$ cfp(G1) : $\{\mathscr{B}$ proposal(A, G1)$\}$ $\leftarrow$ $*$G1;
+proposal(A, G4) : $\{\mathscr{B}$ $\neg$cando(G4), $\neg\mathscr{B}$ awarded(G4)$\}$ $\leftarrow$ $\uparrow$(A, **b**, award(G4)),
+awarded(G4);
$+\downarrow$(**p**, Ga) : $\{\top\}$ $\leftarrow$ $+!_p$ Ga;
$+\downarrow$(**b**, B): $\{\top\}$ $\leftarrow$ +B;
$+!_p$ respond(G2, Name) : $\{\mathscr{B}$ cando(G2), $\mathscr{B}$ my_name(A)$\}$ $\leftarrow$ $\uparrow$(Name, **b**, proposal(A, G2));
$+!_p$ respond(G3, Name) : $\{\mathscr{B}$ $\neg$cando(G3), $\mathscr{B}$ my_name(A)$\}$ $\leftarrow$ $\uparrow$(Name, **b**, sorry(A, G3));
+award(G5) : $\{\top\}$ $\leftarrow$ $+!_a$ G5 ;
$+!_p$ cfp(G6) : $\{\top\}$ $\leftarrow$ $*$G6;

:**name**: ag3

: **Initial Beliefs** :

my_name(ag3)
cando(g2)
$\neg$cando(g)

:**Plans**:

$+!_a$ g : $\{\mathscr{B}$ cando(g)$\}$ $\leftarrow$ a;
$+!_a$ g2 : $\{\mathscr{B}$ cando(g2)$\}$ $\leftarrow$ a2;
$+!_a$ Gl : $\{\mathscr{B}$ $\neg$cando(Gl)$\}$ $\leftarrow$ $+!_p$ cfp(Gl);
$+!_p$ cfp(G) :
$\{\mathscr{B}$ ag(A1), $\mathscr{B}$ my_name(Name), $\neg\uparrow$ (A1, **p**, respond(G, Name)), $\neg\mathscr{B}$ proposal(Ag, G)$\}$ $\leftarrow$
$\uparrow$(A1, **p**, respond(G, Name));
$+!_p$ cfp(G1) : $\{\mathscr{B}$ proposal(A, G1)$\}$ $\leftarrow$ $*$G1;
+proposal(A, G4) : $\{\mathscr{B}$ $\neg$cando(G4), $\neg$ $\mathscr{B}$ awarded(G4)$\}$ $\leftarrow$ $\uparrow$(A, **b**, award(G4)),
+awarded(G4);
$+\downarrow$(**p**, Ga) : $\{\top\}$ $\leftarrow$ $+!_p$ Ga;

+↓(**b**, B): {⊤} ← +B;
+!$_p$ respond(G2, Name) : {$\mathscr{B}$ cando(G2), $\mathscr{B}$ my_name(A)} ← ↑(Name, **b**, proposal(A, G2));
+!$_p$ respond(G3, Name) : {$\mathscr{B}$ ¬cando(G3), $\mathscr{B}$ my_name(A)} ← ↑(Name, **b**, sorry(A, G3));
+award(G5) : {⊤} ← +!$_a$ G5 ;
+!$_p$ cfp(G6) : {⊤} ← ∗G6;

B.3 Basic Auction Code

Code Example 2.2 Basic Auction

:**name**: ag1

: **Initial Beliefs** :

:**Plans**:

+↓(**b**, B): {⊤} ← +B;
+bid(Z, A) : {$\mathscr{B}$ bid(X1, ag2), $\mathscr{B}$ bid(X2, ag3), $\mathscr{B}$ bid(X3, ag4), $\mathscr{B}$ bid(200, Ag)} ←
↑(Ag, **b**, win);

:**name**: ag2

: **Initial Beliefs** :

my_name(ag2)

: **Initial Goals**:

!$_p$ bid

:**Plans**:

+↓(**b**, B): {⊤} ← +B;
+!$_p$ bid : {$\mathscr{B}$ my_name(Name), ¬↑(ag1, **b**, bid(100, Name))} ← ↑(ag1, **b**, bid(100, Name));

Agents 3 and 4 are identical to agent 2 except for the amount they bid.

B.4 Auction Coalition Code

The code for agent 1 is the same as in the Basic Auction example.

Code Example 2.3 Auction Coaltion

:**name**: ag2

: **Initial Beliefs** :

my_name(ag2)

: **Initial Goals**:

!$_p$ coalition

:**Plans**:

+↓(**b**, B): {⊤} ← +B;
+!$_p$ bid : {$\mathscr{B}$ my_name(Name), ¬↑(ag1, **b**, bid(250, Name))} ← ↑(ag1, **b**, bid(250, Name));
+!$_p$ coalition : {$\mathscr{B}$ my_name(Ag), ¬ ↑(ag3, **b**, coalition (Ag))} ← ↑(ag3, **b**, coalition (Ag));
+agree(A, X) : {⊤} ← +!$_p$ bid;

:**name**: ag3

: **Initial Beliefs** :

my_name(ag3)

: **Initial Goals**:

$!_p$ bid

:**Plans**:

+↓(**b**, B): {⊤} ← +B;
+$!_p$ _p bid : {$\mathscr{B}$ my_name(Name), ¬↑(ag1, **b**, bid(150, Name))} ← ↑(ag1, **b**, bid(150, Name));
+ coalition (A) : { $\mathscr{B}$ my_name(Name), ¬↑(A, **b**, agree(Name, 150))} ← ↑(A, **b**, agree(Name, 150));

The code for agent 4 is the same as for agent 3 except with a different bid amount.

B.5 Dynamic Auction Coalition Code

The action $win(Z,A)$ makes the fact that agent A has won with amount Z available to all agents by perception.

Code Example 2.4 Dynamic Auction

:**name**: ag1

: **Initial Beliefs** :

my_name(ag1)

: **Belief Rules**:

$\mathscr{B}$ allbids :− $\mathscr{B}$ bid_processed(ag2), $\mathscr{B}$ bid_processed(ag3), $\mathscr{B}$ bid_processed(ag4);

:**Plans**:

+↓(**b**, bid(D, From)) : {$\mathscr{B}$ bid(E, From)} ← −bid(From, E),
    +bid(From, D);
+↓(**b**, bid(D, From)) : {¬$\mathscr{B}$ bid(E, From)} ← +bid(From, D);
+bid(Z, A) : {$\mathscr{B}$ current_winner(Ag1, Amw), Amw < A, $\mathscr{B}$allbids} ← `lock`,
    −current_winner(Ag1, Amw),
    +ann_winner,
    +current_winner(Z, A),
    win(Z, A),
    `unlock`;
+bid_processed(Ag) : {$\mathscr{B}$ current_winner(Agw, Amw), $\mathscr{B}$allbids , ¬$\mathscr{B}$ ann_winner} ← `lock`,
    +ann_winner,
    win(Agw, Amw),
    `unlock`;
+bid(Ag, Am) : {¬$\mathscr{B}$ current_winner(Ag2, Amw)} ←
    +current_winner(Ag, Am),
    +bid_processed(Ag);
+bid(Ag, Am) : {$\mathscr{B}$ current_winner(Agw, Amw), ¬(Am < Amw), ¬$\mathscr{B}$ allbids } ← `lock`,
    +current_winner(Ag, Am),
    +bid_processed(Ag),
    −current_winner(Agw, Amw),
    `unlock`;
+bid(Ag, Am) : {$\mathscr{B}$ current_winner(Agw, Amw), Am < Amw, ¬$\mathscr{B}$allbids} ← +bid_processed(Ag);

:**name**: ag2

: **Initial Beliefs** :

my_name(ag2)
collaborator (ag3)
cash(150)

: **Initial Goals**:

$!_p$ bid

:**Plans**:

+↓(**b**, B): {⊤} ← +B;
+!$_p$ bid : {$\mathscr{B}$ my_name(Name), $\mathscr{B}$cash(C), ¬ ↑(ag1, **b**, bid(C, Name))} ← ↑(ag1, **b**, bid(C, Name));
+agree(A, X): {$\mathscr{B}$ cash(C), $\mathscr{B}$ my_name(Name)} ← ↑(ag1, **b**, bid((C + X), Name));
+win(Ag, X): {$\mathscr{B}$ my_name(Name), ¬$\mathscr{B}$win(Name, Any), $\mathscr{B}$ collaborator (Coll)} ←
    +!$_a$ coalition (Coll) ;
+!$_a$ _p coalition (Coll) : {$\mathscr{B}$ my_name(Ag), ¬ ↑(Coll, **b**, coalition (Ag))} ←
    ↑(Coll, **b**, coalition (Ag)),
    + coalition (Coll);

:**name**: ag3

: **Initial Beliefs** :

my_name(ag3)
cash(150)

: **Initial Goals**:

!$_p$ bid

:**Plans**:

+↓(**b**, B): {⊤} ← +B;
+!$_p$ bid : {$\mathscr{B}$ my_name(Name), $\mathscr{B}$cash(C), ¬ ↑(ag1, **b**, bid(C, Name))} ← ↑(ag1, **b**, bid(C, Name));
+ coalition (A) : { $\mathscr{B}$ my_name(Name), $\mathscr{B}$cash(C), ¬ ↑(A, **b**, agree(Name, C))} ←
    ↑(A, **b**, agree(Name, C));

The code for agent 4 is the same as for agent 3 except with a different bid amount.

B.6 Auction Trust Code

The action $win(A,Z)$ makes the fact that agent A has won with amount Z available to all agents by perception.

Code Example 2.5 Trust Auction

:**name**: ag1

: **Initial Beliefs** :

:**Plans**:

+↓(**b**, bid(D, From)) : {$\mathscr{B}$ bid(From, E)} ← −bid(From, E),
    +bid(From, D);
+↓(**b**, bid(D, From)) : {¬$\mathscr{B}$ bid(From, E)} ← +bid(From, D);
+bid(Z, A) : {$\mathscr{B}$ bid(ag2, X1), $\mathscr{B}$ bid(ag3, X2), $\mathscr{B}$ bid(ag4, X3),
        ¬$\mathscr{B}$ winning_amount(Am), X2 < X1, X3 < X1} ←
    +winning_amount(X1),
    win(ag2, X1);
+bid(Z, A) : {$\mathscr{B}$ bid(ag2, X1), $\mathscr{B}$ bid(ag3, X2), $\mathscr{B}$ bid(ag4, X3),
        ¬$\mathscr{B}$ winning_amount(Am), X1 < X2, X3 < X2} ←
    +winning_amount(X2),
    win(ag3, X2);
+bid(Z, A) : {$\mathscr{B}$ bid(ag2, X1), $\mathscr{B}$ bid(ag3, X2), $\mathscr{B}$ bid(ag4, X3),
        ¬$\mathscr{B}$ winning_amount(Am), X2 < X3, X1 < X3} ←
    +winning_amount(X2),
    win(ag4, X3);
+bid(Z, A) : {$\mathscr{B}$ winning_amount(Am), Am < A} ← −winning_amount(Am),
    +winning_amount(A),
    win(A, Z);

:**name**: ag2

: **Initial Beliefs** :

my_name(ag2)
trust (ag3)

: **Initial Goals**:

$!_p$ bid

:**Plans**:

+↓(**b**, B): {⊤} ← +B;
+$!_p$ _p bid : {$\mathscr{B}$ my_name(Name), ¬↑(ag1, **b**, bid(150, Name))} ← ↑(ag1, **b**, bid(150, Name));
+win(A, X) : {$\mathscr{B}$ my_name(Name), ¬$\mathscr{B}$win(Name, Y), $\mathscr{B}$ trust (Ag), ¬ ↑(Ag, **b**, coalition (Name))}
← ↑(Ag, **b**, coalition (Name));
+agree(A, X) : {⊤} ← ↑(ag1, **b**, bid(300, ag2));

:**name**: ag3

: **Initial Beliefs** :

my_name(ag3)

: **Initial Goals**:

$!_p$ bid

:**Plans**:

+↓(**b**, B): {⊤} ← +B;
+$!_p$ bid : {$\mathscr{B}$ my_name(Name), ¬↑(ag1, **b**, bid(150, Name))} ← ↑(ag1, **b**, bid(150, Name));
+ coalition (A) : { $\mathscr{B}$ my_name(Name), ¬↑(A, **b**, agree(Name, 150))} ← ↑(A, **b**, agree(Name, 150));

The code for agent 3 is the same as for agent 4 except with a different bid amount.

B.7 Auction Dynamic Trust Code

Code Example 2.6 Dynamic Trust Action

:**name**: ag1

:**Plans**:

+↓(**b**, bid(D, From)) : {$\mathscr{B}$ bid(From, E)} ← −bid(From, E),
+multiple_bidder (From),
+bid(From, D);
+↓(**b**, bid(D, From)) : {¬$\mathscr{B}$ bid(From, E)} ← +bid(From, D);
+bid(Z, A) : {$\mathscr{B}$ bid(ag2, X1), $\mathscr{B}$ bid(ag3, X2), $\mathscr{B}$ bid(ag4, X3), $\mathscr{B}$ bid(ag5, X4),
¬$\mathscr{B}$ winning_amount(Am), X2 < X1, X3 < X1, X4 < X1} ←
+winning_amount(X1),
win(ag2, X1);
+bid(Z, A) : {$\mathscr{B}$ bid(ag2, X1), $\mathscr{B}$ bid(ag3, X2), $\mathscr{B}$ bid(ag4, X3), $\mathscr{B}$ bid(ag5, X4),
¬$\mathscr{B}$ winning_amount(Am), X1 < X2, X3 < X2, X4 < X2} ←
+winning_amount(X2),
win(ag3, X2);
+bid(Z, A) : {$\mathscr{B}$ bid(ag2, X1), $\mathscr{B}$ bid(ag3, X2), $\mathscr{B}$ bid(ag4, X3), $\mathscr{B}$ bid(ag5, X4),
¬$\mathscr{B}$ winning_amount(Am), X2 < X3, X1 < X3, X4 < X3} ←
+winning_amount(X3),
win(ag4, X3);
+bid(Z, A) : {$\mathscr{B}$ bid(ag2, X1), $\mathscr{B}$ bid(ag3, X2), $\mathscr{B}$ bid(ag4, X3), $\mathscr{B}$ bid(ag5, X4),
¬$\mathscr{B}$ winning_amount(Am), X2 < X4, X1 < X4, X3 < X4} ←
+winning_amount(X4),
win(ag5, X4);
+bid(Z, A) : {$\mathscr{B}$ winning_amount(Am), Am < A} ← −winning_amount(Am),
+winning_amount(A), win(A, Z);
+bid(Z, A) : {$\mathscr{B}$ multiple_bidder (Z), $\mathscr{B}$ winning_amount(Am), A < Am} ← ↑(Z, **b**, failed_bid);

```
:name: ag2

: Initial Beliefs :

my_name(ag2)
 trust (ag3)
 trust (ag4)

: Initial Goals:

!_p bid

:Plans:

+↓(b, B): {⊤} ← +B;
+!_p bid : {B my_name(Name), ¬↑(ag1, b, bid(150, Name))} ← ↑(ag1, b, bid(150, Name));
+win(A, X) : {B my_name(Name), ¬Bwin(Name, Y), B trust (Ag),
                ¬B formed_coalition (AgB), ¬ ↑(Ag, b, coalition (Name))} ←
    ↑(Ag, b, coalition (Name)),
    +formed_coalition (Ag);
+ failed_bid : {B my_name(Name), ¬Bwin(Name, Y),
                B trust (Ag), B formed_coalition (AgB), ¬ ↑(Ag, b, coalition (Name))} ←
    ↑(Ag, b, coalition (Name)),
    +formed_coalition (Ag),
    −trust (AgB);
+agree(A, X) : {⊤} ← ↑(ag1, b, bid((X + 150), ag2));

:name: ag3

: Initial Beliefs :

my_name(ag3)

: Initial Goals:

!_p bid

:Plans:

+↓(b, B): {⊤} ← +B;
+!_p bid : {B my_name(Name), ¬↑(ag1, b, bid(25, Name))} ← ↑(ag1, b, bid(25, Name));
+ coalition (A) : { B my_name(Name), ¬↑(A, b, agree(Name, 25))} ← ↑(A, b, agree(Name, 25));
```

The code for agents 4 and 5 are the same as for agent 3 except with a different bid amount.

B.8 Cleaning Robot Code

The action $next(slot)$ moves the robot to the next space on the grid and updates all perceivable facts accordingly (e.g., whether the agent can see any garbage etc.). The action $drop(garb)$ makes garbage perceivable in that grid slot (assuming the agent was holding garbage). The action $moveTowards(X1,Y1)$ makes the agent move one square towards the coordinates $(X1,Y1)$. The action $burn(garb)$ destroys a piece of garbage.

Code Example 2.7 Cleaning Robots

```
:name: r1

: Initial Beliefs :

pos(r2, 2, 2)
checking( slots )

:Plans:

+pos(r1, X1, Y1) : {B checking( slots ), ¬B garbage(r1)} ← next( slot );
```

```
+garbage(r1) : {𝒷 checking( slots )} ← +!_p stop(check),
    +!_p take(garb, r2),
    +!_p continue(check);
+!_p stop(check) : {⊤} ← +!_a pos(r1, X1, Y1),
    +pos(back, X1, Y1),
    −checking(slots );
+!_p take(S, L) : {⊤} ← +!_p ensure_pick(S),
    +!_p go(L),
    drop(S);
+!_p ensure_pick(S) : {𝒷 garbage(r1)} ← pick(garb),
    +!_p ensure_pick(S);
+!_p ensure_pick(S) : {⊤} ← donothing;
+!_p continue(check) : {⊤} ← +!_p go(back),
    −pos(back, X1, Y1),
    +checking( slots ),
    next( slot );
+!_p go(L) : {𝒷 pos(L, X1, Y1), 𝒷 pos(r1, X1, Y1)} ← donothing;
+!_p go(L) : {⊤} ← +!_a pos(L, X1, Y1),
    moveTowards(X1, Y1),
    +!_p go(L);

:name: r2

:Plans:

+garbage(r2) : {⊤} ← burn(garb);
```