Translating into an Intermediate Agent Layer: A Prototype in Maude*

(Extended Abstract)

Berndt Farwer[1], Louise Dennis[2]

[1] Durham University,UK
[2] University of Liverpool, UK
Contact: berndt.farwer@durham.ac.uk

Abstract. This paper summarises how agent programming languages are be embedded into an intermediate agent layer called *Agent Infrastructure Layer (AIL)* that can be seen as a unifying framework for such languages. We discuss common concepts of the agent programming languages on which the AIL's design is based and outline a translation of AgentSpeak programs into this intermediate layer. An executable prototype of the AIL is implemented in Maude, allowing language translators to be tested at this prototype level.

1 Introduction

The notion of a software *agent* was introduced about three decades ago [11] with the term 'agent programming' being introduced in the late 1980s and made popular by Shoham [17]. It has since developed into a vast array of agent programming languages [2]. Though being very diverse in nature, most languages share some common agent-related concepts. However, agent programming as a whole still lacks a comprehensive approach to model checking.

Among the agent programming languages, some of the most widely used rely on the *belief, desire, and intention* paradigm. For this reason, we have chosen to focus on some major players in this league as primary candidates for integration into the intermediate language. This should by no means exclude other languages even based on different paradigms from the theory, it might just not be quite as straightforward to embed them.

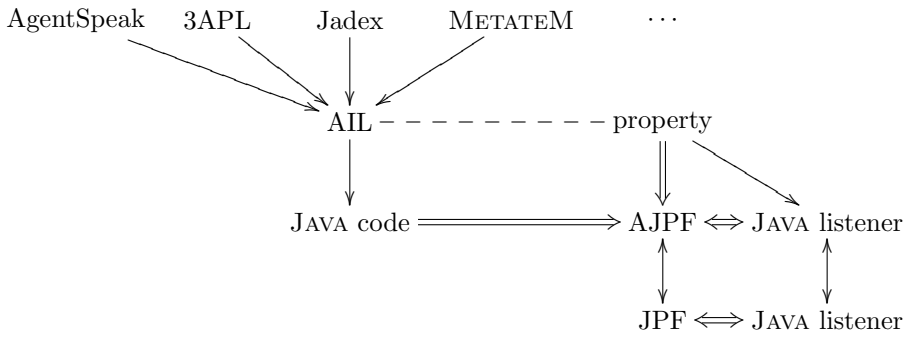
Another prerequisite of our endeavour was to include languages that have practical relevance, i.e., not to restrict ourselves to languages that are so limited that they would not be considered for serious software projects. This decision ruled out some of the lean languages that have nice mathematical properties but no relevance for real-world programming.

* Supported by EPSRC grants EP/D054788 (Durham) and EP/D052548 (Liverpool).

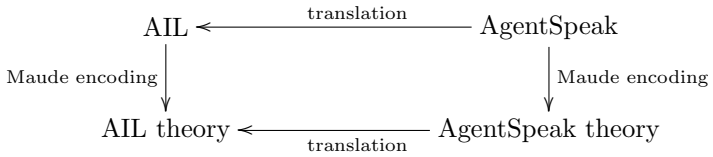
2 The Agent Infrastructure Layer – AIL

In this section we briefly summarise the idea and the current state of our intermediate layer called the *The Agent Infrastructure Layer (AIL)*. It is worth noting that the AIL is not intended as a new programming language, but rather as a compiler language, consisting mainly of a set of suitable-for-the-task data structures. We will not discuss in detail the operational semantics of AIL, since this can be found elsewhere [8,7].

AIL is intended to be a general framework with which a number of agent programming languages can interface through translators with the ultimate goal of providing a base for efficient model checking. The properties to be checked are specified at the AIL level that provides access to the agent-specific components of a program in a unified way. The language-specific translators take care of a faithful embedding of the original program into the lower-level AIL representation. This involves a translation of the state representation (beliefs, plans, intentions, etc.) as well as the adaption of some crucial functions used in the reasoning cycle, such as the selection functions and the action execution function. The following diagram summarises the translation process:



To have a prototype and an execution engine for the proposed intermediate language translation scheme, we are coding large parts of the AIL and target languages in Maude [4,5]. The goal of the Maude implementation is to have an operational prototype as a testbed for our translations, as shown in the following commuting diagram:



In order to accommodate as many agent programming languages as possible, we have decided to allow a dynamic configuration of the AIL's reasoning cycle. This is handled by the last component of the state tuple, a list of reasoning cycle states, the head of which is the current state, with the second element being the next state to visit. When moving on to the next state, the list elements are rotated, i.e., the head of the list gets appended to the back and the

second element becomes the new head. The reason for this representation of the reasoning cycle within the state is the desire to let language translators define a suitable sequence of states for the AIL to accommodate the specific target language's needs. For each language translation, however, the sequence of states will be static, just as the reasoning cycle is well defined and does not change dynamically at run time for any of the languages that we are targeting with this approach.

Eventually, AIL will be implemented as a set of JAVA classes, tuned to allow efficient model checking using an extended version of JPF. Before discussing a specific translation in Section 4 we summarise some of the languages to be considered for translation and their concepts.

3 The Languages Considered

At present we are considering four languages for translation into AIL. These languages are briefly summarised below. It should be noted, however, that while the AIL semantics was designed to accommodate these languages, we expect that various other programming languages for multi-agent systems will also be compilable into AIL.

3.1 3APL

3APL [12] (pronounced "triple-a-p-l") is a popular BDI language which is under continued development at Utrecht. It has well-defined operational semantics, and many extensions and refinements. The building blocks of a 3APL agent are *beliefs*, *goals*, *plans* and *rules for revising plans*. The language itself is based on the logic programming paradigm, but is also combined with imperative programming (JAVA in particular). 3APL also contains semantics for sending and receiving messages and there is research on adapting 3APL programs to wider MAS settings [6].

3.2 AgentSpeak

AgentSpeak(L) was originally designed as an abstract agent programming language [16], but has since been extended to become a programming language that can incorporate most major MAS techniques usable for real-world programming while still providing formal (operational) semantics. The AgentSpeak dialect we are considering is based on the current implementation of *Jason* [14,3], an interpreter for an extended version of AgentSpeak. Like 3APL, AgentSpeak is also based on logic programming, and its key components are *beliefs*, *goals*, *plans*, and *intentions*.

3.3 Jadex

Jadex [15], like JADE [13], is based on JAVA. Unlike 3APL and AgentSpeak, its formal semantics is yet to be established. Jadex is also based on the BDI

paradigm, thus sharing its main concepts – *beliefs*, *goals*, and *plans* – with the other languages. Beliefs are stored as *named facts* and the *belief state* can trigger actions to be carried out. Goals describe the *desires* of an agent and are explicitly represented as objects contained in a *goalbase*. They are accessible to the reasoning component as well as to plans, but plans and goals are stored separately. The *goal life cycle* distinguishes between three *goal states* (i.e., option, active, and suspended) and four *types of goals* (i.e., perform, achieve, query, and maintain). Plans consist of *head* and *body*, where the head contains a trigger and possibly some bindings, while the body contains the reference to some JAVA-based actions to be performed.

3.4 MetateM

METATEM is quite distinct from the three languages above [9]. It was originally developed as a language for executable temporal specifications [1], but has been extended to incorporate (bounded) beliefs, deliberation mechanisms, etc. Two aspects make it significantly different from the first two of the above approaches and also different from the approach taken by Jadex. The first is that the basic computational mechanism is *not* logic programming of the Prolog form. It is (maximal) forward chaining, much more akin to tableau construction. Second, the emphasis, since the development of Concurrent METATEM, has primarily been on the organisational and grouping aspects within multi-agent systems, rather than on individual agents [10]. Indeed, a 'motto' of work on METATEM is: "every group is an agent, and every agent is a group".

4 Translating AgentSpeak

We focus in this presentation on the translation of AgentSpeak into AIL. Only the initial state of an AgentSpeak program is translated into a state of AIL. It can be shown that the executions possible in AIL correspond exactly to the behaviour of the AgentSpeak counterpart.

4.1 State Representation and Transformation

State Representation in AIL An agent state in AIL is given as a tuple $< ag, i, I, Pl, A, BBR, P, C, In, Out, Cn, Cx, Ann, RC >$ where ag is a unique identifier for the agent, i is the current intention, I is all extant intentions, Pl the currently applicable plans (only used in one phase of the cycle), A are actions to be executed, B the agent's beliefs, BR the agent's belief rules, P the agent's plans, C the agent's constraints, In the agent's inbox, Out the agent's outbox, Cn the agent's content, Cx the agent's context and Ann a set of annotations (not used by AIL but which may be used by translators to store additional information) and RC is the agent's reasoning cycle - a sequence of AIL stages **A**, **B**, **C**, **D**, **E**, **F**, indicating the current stage and the subsequent stages of the reasoning cycle for the agent. Figure 1 shows the basic reasoning cycle for

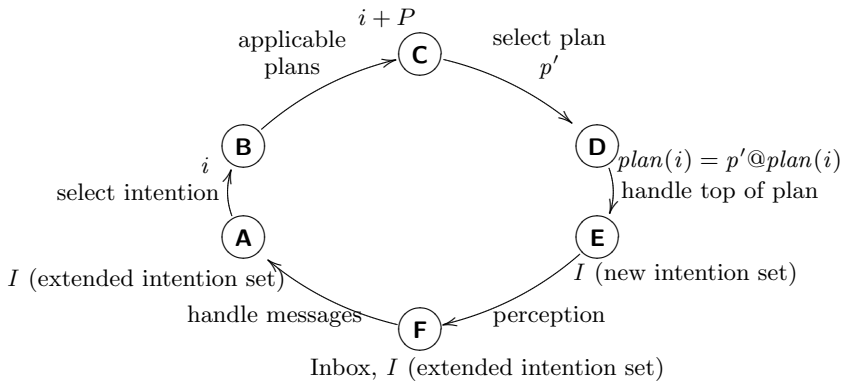


Fig. 1. AIL's reasoning cycle

RC = **ABCDEF**. This sequence can be configured by the language translator but will be fixed for each language, i.e., no dynamic re-configuration is allowed or even possible using the transition rules of the operational semantics.

State Representation in AgentSpeak In AgentSpeak the state of an agent is referred to as its configuration given as a tuple $\langle ag, C, M, T, s\rangle$. The last component $s \in \{\mathsf{ProcMsg}, \mathsf{SelEv}, \mathsf{RelPl}, \mathsf{ApplPl}, \mathsf{SelAppl}, \mathsf{AddIM}, \mathsf{SelInt}, \mathsf{ExecInt}, \mathsf{ClrInt}\}$ denotes the stage of the reasoning cycle the agent is currently engaging in. All other components of the configuration are themselves structures: The agent's *circumstance* C is a tuple $\langle I, E, A\rangle$ consisting of a set of *intentions* I, a set of *events* E and a set of *actions* A. Both E and A do not have an explicit counterpart in AIL states. The messages' component $M = \langle In, Out, SI\rangle$ comprises an *inbox*, an *outbox*, and information on *suspended intentions*.

The following table shows some basic correspondences. Of course, the entities have to be individually translated into equivalent representations with respect to the used data structures.

AgentSpeak	bs	ps	I	In	Out	Ap	ι	s
AIL	B, BR	P	I	In	Out	Pl	i	RC

Many modern agent programming languages allow Prolog-style reasoning, which in the case of AgentSpeak is included in the belief set $ASbeliefset$. Since AIL distinguishes beliefs and belief rules into the sets B and BR, respectively, the beliefs from the AgentSpeak specification have to be mapped into these two sets.

The set Ann in the AIL state is provided for the language translators to store language specific information that has no direct representation in the other AIL components. For AgentSpeak this includes the event set E and the action set A, since AgentSpeak does not require the immediate execution of an action. We

therefore supply a mechanism by which additional information is temporarily stored in *Ann* and the function *do* that is provided by the language interpreter overrides AIL's default and thus implementing the AgentSpeak way of dealing with events and actions. Newer versions of AIL (from v1.2) provide an action set in the agent state.

AgentSpeak's reasoning cycle is depicted in Figure 2. The dotted lines represent short-cuts that are given as special rules in the AgentSpeak semantics. What becomes obvious when comparing this with the AIL's reasoning cycle (Fig 1) is that, e.g., the selection of an intention initiates the AIL reasoning cycle while it takes place at a later stage in AgentSpeak. This mismatch leads to a custom reasoning cycle *RC* provides by the AgentSpeak-to-AIL translator.

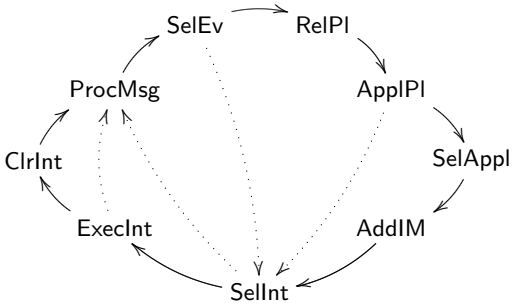


Fig. 2. AgentSpeak's reasoning cycle

4.2 Agent Components

Instead of giving a formal translation, we discuss the components that have to be taken into consideration for the translation, including the AIL functions that are definable by the translator, thereby overriding the defaults in AIL.

Plans In AgentSpeak plans take the form

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

where *te* is a *triggering event*, *ct* is a *context*, and *h* is a list of *actions*. In AIL there is no special syntax for plans. They are however given in a data structure consisting of a trigger, a prefix, a guard stack, and a body. These are given in the form of a table. For example, the plan `+concert(A,V)` : `likes(A)` ← `!book_tickets(A,V)` is represented as

PLAN:	
trigger	`+concert(A,V)`
prefix	`[`ϵ`]`
guard stack	`likes(A)`
body	`!book_tickets(A,V)`

A slightly more complex example is the plan
`+book_tickets(A,V)` : ¬`busy(phone)` ← `?phone_number(V,N);call(N);...;!choose_seats(A,V)`

PLAN:	
trigger	`+book_tickets(A,V)`
prefix	[ϵ]
guard stack	¬`busy(phone)`

body	`?phone_number(V,N)`
	`call(N)`
	⋮
	`!choose_seats(A,V)`

For the purpose of formalising the translation we represent the tables as quadruples: $\langle trigger, prefix, guardstack, body\rangle$. Stacks are given as $h; t_1; \ldots; t_n$ where h is the head, i.e. the topmost element of the stack, and $t_1; \ldots; t_n$ is the remainder or tail of the stack. For the example this yields $\langle$`+book_tickets(A,V)`, [ϵ], ¬`busy(phone)`, `?phone_number(V,N);call(N);...; !choose_seats(A,V)`$\rangle$

AgentSpeak Annotations Annotations are used in AgentSpeak to specify the source of beliefs. E.g. in `winter[self]` the annotation `self` denotes that the agent has gained the belief that it is winter by their own perception. In contrast, the belief `winter[Tom]` denotes the fact that Tom has told the agent that it is winter. Multiple annotations are possible, for instance for cases in which the belief has been added from several sources., e.g. `winter[self, Tom]` denoting that in addition to my own perception of winter, Tom has also told me that it is winter.

The AIL data structures do not directly support this kind of annotation that is taken into consideration whenever there is a search for a matching belief. To overcome this situation, the translation uses a fresh (i.e. otherwise unused) binary predicate `ann` to keep track of annotations. This predicate can then be used in the context of plans to provide the necessary information.

Extending the example from the previous section, the plan

`+concert(A,V)` : `likes(A)[self]` ← `!book_tickets(A,V)`

is represented as

PLAN:	
trigger	`+concert(A,V)`
prefix	[ϵ]
guard stack	`ann(likes(A),self)` ∧ `likes(A)`
body	`!book_tickets(A,V)`

The relevant plans are not affected by annotations, but the applicable plans are restricted by the annotations. When evaluating the context of an AgentSpeak rule, each predicate in the conjunction has to be derivable with an annotation that contains the context's annotations. Formally: Let I be a finite index set, then a context $\bigwedge_{i\in I} c_i[A_i]$ with predicates c_i and possibly empty lists of annotations A_i is satisfied iff $\bigwedge_{i\in I} c_i[A'_i]$ with $\forall i \in I.\text{set}(A_i) \subseteq \text{set}(A'_i)$ is derivable.

Beliefs AgentSpeak beliefs can be annotated with an array of annotations. These annotations will be accessed when a belief is checked. They can contain information on the source of information, etc. Each belief can have annotations, but there cannot be the same belief with different annotations, therefore it is best to view beliefs always in conjunction with their individual annotation set. The latter would then be empty for the case that a specific belief has no annotations. If there are annotations, then all annotations are collected in the belief's annotation set.

Let $b \in bs$ be a belief, then we denote by $ann(b)$ its annotation set. An annotation set can itself be modelled as a belief in AIL such that any operation involving a belief in AgentSpeak will effectively have to involve two beliefs in the translation as discussed in the previous section on AgentSpeak annotations.

Prolog-style Reasoning The Prolog rules in AgentSpeak are stored as beliefs. In AIL we only allow first order terms as beliefs, so that the Prolog rules have to be filtered into the BR component.

4.3 AIL Functions

In this section we summarise the AIL functions that can be overridden by the language-specific translators:

Perception This function monitors the environment and the agents received messages. It is used to add new intentions to the agent's intention set. This function has to be implemented to mimic the behaviour of the target language's perception model.

Perform Action The execution of an action is handled directly in the AIL semantics, whereas it is left to the environment in AgentSpeak's semantics. The translation handles this by overriding the default action execution function:

The do Function AIL provides a special function to handle actions: *do*. For the AgentSpeak translation, this function has to access the Ann component of the state in order to handle the extraction of actions from the intentions in AgentSpeak correctly. The reason for this is that actions do not have to be executed immediately in AgentSpeak. Instead they are temporarily stored in the A component of the agent's circumstance. The environment thereafter takes care that these actions are executed.

Select Intention In *Jason* as in the proposed AIL the intentions are actually stored in a list representing the set of intentions. When an intention is selected this is usually done by selecting the head of the list and then rotating the intentions in the list. The user can program the selection function in *Jason*, so this has to be translated for use with AIL. The translation can work on the data structure of the intention set such that the translation only has to replace the respective lists.

Select Plan Based on Current Intention The AIL provides a default selection function that can be overridden by the translator to accommodate the target language or a user-programmed special plan-selection function.

Initial Annotation Not to be confused with the AgentSpeak annotations, the component *Ann* of the agent configuration is a place to store arbitrary information that the individual functions provided by the translator can use. For the AgentSpeak translation, for instance, the actions to be executed are temporarily be stored here.

Relevance The relevance function(s) are provided in AIL primarily to model dynamic communities by making information from certain sources irrelevant in certain situations at the stage of belief update. Since we do not have the concept of groups or communities in AgentSpeak everything is regarded as relevant here.

Filter Unwanted Plans This function gives the translator the possibility to exclude some plans deemed applicable by the AIL. For the AgentSpeak translation this is not needed, since the AIL's *applicable plans* are exactly those that AgentSpeak regards as *relevant* and *applicable* in states RelPl and ApplPl of the reasoning cycle (cf. Fig. 2).

Consistency For reasons of efficiency, most agent programming languages do no check for consistency of the belief base. Hence, the default for this function is the constant true function. However, if it is desirable to introduce a consistency check, then this function can be overridden by the translator. Apart from this, there is no need to apply any further changes to the AIL, since its semantics is designed to query this function prior to any belief update.

5 The Maude Implementations

Both AIL and AgentSpeak implementations use several *sorts* like sets, stacks, and other language specific sorts. For space reasons we cannot discuss all of these and restrict the presentation to the representation of the agents states. We then showcase one rule each of the AgentSpeak semantics and the AIL semantics together with their respective Maude translations.

5.1 Agent Definitions

The Maude definition of an agent's state vector for AgentSpeak is given in Listing 1.1. For AIL the definition is given in Listing 1.2. Both use externally defined sorts and functions that are loaded prior to the definition of the agent state sort and constructor. The space limitations for this paper prevent us from discussing the implementation in detail.

```
mod AS-AGENT is
    protecting SET{Plan} .
    protecting SET{ExtBelief}.
    protecting SET{Intention} .
    protecting SET{Action} .
    protecting SET{Event} .
    protecting INTENTION .

*** steps in the reasoning cycle
    sort RCStep .

    ops  SelEv RelPl SelInt ApplPl SelAppl AddIM
    ExecInt ClrInt SelEv ProcMsg : -> RCStep .

*** components of configurations
        sorts AgentProgram AgentCircumstance Conf .
        sort Messages .
        sort TempInf .

    op [_,_] : Set{ExtBelief} Set{Plan}
        -> AgentProgram [ctor] .
    op <_,_,_> : Set{Intention} Set{Event} Set{Action}
        -> AgentCircumstance [ctor] .
    op <_,_,_,_,_> : AgentProgram AgentCircumstance Messages
        TempInf RCStep
        -> Conf [ctor] .
    op (_,_,_,_,_) : Set{Plan} Set{Plan} Intention Event Plan
        -> TempInf .

endm
```

Listing 1.1. Module AS-AGENT

```
fmod AIL-AGENT is
    protecting DEFAULT-SETTINGS .

    sorts AilAgent AilAgentProgram AILStage ReasoningCycle .

    ops StA StB StC StD StE StF : -> AILStage .
    subsort AILStage < ReasoningCycle .
    op _;rc_ : ReasoningCycle ReasoningCycle -> ReasoningCycle [assoc ctor] .

    op <_,_,_,_,_,_,_,_,_,_> : Const Stack{PExtBelief} Stack{PSource}
    Stack{PSource} Stack{PBeliefRule} GoalStack Stack{PPlan}
    Stack{PConstraint} ReasoningCycle -> AilAgentProgram [ctor] .

        .
        .
        .

endfm
```

Listing 1.2. Module AIL-AGENT

5.2 Operational Semantics

The semantics is given as a set of rules formalising the possible state transitions of the agent. Fig. 1.3 shows the Maude implementation of one of the rules for finding the applicable plans in AgentSpeak:

$$\frac{\mathsf{AppPlans}(ag_{bs}, T_R) \neq \emptyset}{\langle ag, C, M, T, \mathsf{ApplPl}\rangle \rightarrow \langle ag, C, M, T', \mathsf{SelAppl}\rangle} \ (\mathrm{Appl}_1)$$

where $T'_{Ap} = \mathsf{AppPlans}(ag_{bs}, T_R)$

This rule updates the temporary information in the agent's state with the applicable plans with respect to the agent's beliefs and the currently relevant plans. It also advances the stage in the reasoning cycle from ApplPl to SelAppl, where one of the applicable plans will be selected.

```
crl [Appl1] :
  < [bs,ps], C, M, (R, Ap, iota, epsilon, rho), ApplPl >
    => < [bs,ps], C, M, (R,AppPlans(bs,R),iota,epsilon,rho), SelAppl >
    if  AppPlans(bs,R) =/= (empty).Plan .
```

Listing 1.3. Rule Appl1

The condition of the rule appears as the conditional of the Maude rewriting rule. The additional condition $T'_{Ap} = \mathsf{AppPlans}(ag_{bs}, T_R)$ is coded directly into the right-hand side of the rule. $\mathsf{AppPlans}(\cdot,\cdot)$ is a function defined by:

```
op AppPlans(_,_) : Set{ExtBelief} Set{Plan} -> Set{Plan} .

ceq AppPlans(abs,([te : ct <- bd],ps)) = [te : ct <- bd],AppPlans(abs,ps)
        if AnnMatch(ct,abs).
ceq AppPlans(abs,([te : ct <- bd],ps)) = AppPlans(abs,ps)
        if not(AnnMatch(ct,abs)) .
eq AppPlans(abs,empty) = empty .
```

Listing 1.4. Function AppPlans

In AIL we are faced with the following rule and its implementation in Maude as shown in Listing 1.5:

$$\frac{Pl' = \mathbf{filter}(appPlans(ag, i)) \qquad Pl' \neq \emptyset}{< ag, Pl, \mathbf{B}; RC > \rightarrow < ag, Pl', RC \circledcirc \{\mathbf{B}\} >} \quad (26)$$

This rule illustrates the use of the reasoning cycle sequence in the agent's state vector. On the left-hand side of the consequent, the head of this sequence is **B**. This is a prerequisite for the rule to be applicable. When applying the rule, the next stage is taken, which can be seen on the right-hand side, where **B** is appended to the end of the reasoning cycle while second symbol of the original reasoning-cycle sequence becomes the active stage.

```
crl [26] :
  < Name, as(I, ain(ai(Is, APs, Act), Bels, Libs), Comms, Ann), (StB ;rc RC) >
    => < Name, as(I, ain(ai(Is, AP, Act), Bels, Libs), Comms, Ann), (RC ;rc StB) >
 if AP := filter(appPlans(< Name, as(I, ain(ai(Is, APs, Act), Bels, Libs),
                                    Comms, Ann), (StB ;rc RC) >, I), I)
    /\ AP =/= [] ail .
```

Listing 1.5. Rule 26 of the AIL semantics

6 Conclusion

We have outlined a new approach to model checking of agent programming languages by introducing an intermediate language that preserves essential notions of agent programs, such that the properties can be specified independently of any programming language specific requirements.

Although the design has been optimised for BDI-style languages, we believe that our framework is sufficiently general to accommodate other agent programming languages with just slightly more sophisticated translators.

The next steps in making our approach operational involve implementing the AIL and the language-specific translators, devising a property specification language based on temporal/modal logics and facilitating JPF for the actual model checking. In the long term we aim to extend JPF with agent specific (pre-)processing to make the model checking more efficient.

References

1. H. Barringer, M. Fisher, D. Gabbay, R. Owens, and M. Reynolds, editors. *The Imperative Future: Principles of Executable Temporal Logics*. Research Studies Press, Chichester, United Kingdom, 1996.
2. Rafael H. Bordini, Mehdi Dastani, Jürgen Dix, and Amal El Fallah Seghrouchni, editors. *Multi-Agent Programming: Languages, Platforms and Applications*. Springer, 2005.
3. Rafael H. Bordini and Jomi F. Hübner. *Jason: A Java-based interperter for an extended version of AgentSpeak*, 2006. Available from http://jason.sourceforge.net.
4. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, José Meseguer, and J. Quesada. *Maude: Specification and Programming in Rewriting Logic*, 1999.
5. Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. *Maude Manual (Version 2.3)*. SRI International, Menlo Park, CA 94025, USA, January 2007.
6. Mehdi Dastani, M. Birna van Riemsdijk, and John-Jules Ch. Meyer. Programming multi-agent systems in 3APL. In Bordini et al. [2].
7. L. A. Dennis. Agent Infrastructure Layer (AIL): Design and operational semantics v1.2. Technical report, University of Liverpool, Department of Computer Science, July 2007.
8. Louise Dennis, Berndt Farwer, Rafael Bordini, Michael Fisher, and Michael Wooldridge. A common semantic basis for BDI languages. In Mehdi Dastani, Amal El Fallah Seghrouchni, Alessandro Ricci, and Michael Winikoff, editors, *Proceedings of the International Workshop on Programming Multi-Agent Systems (ProMAS 2007)*, pages 88–103, May 2007.
9. M. Fisher. METATEM: The story so far. In *Proceedings of the Third International Workshop on Programming Multiagent Systems (ProMAS-05)*, volume 3862 of *Lecture Notes in Artificial Intelligence*, pages 3–22. Springer, 2005.
10. M. Fisher, C. Ghidini, and B. Hirsch. Organising Computation through Dynamic Grouping. In *Objects, Agents and Features*, volume 2975 of *Lecture Notes in Computer Science*, pages 117–136. Springer-Verlag, 2004.
11. C. Hewitt. Viewing control structures as patterns of passing messages. *Artificial Intelligence*, 8(3):323–364, 1977.
12. K. V. Hindricks, F. S. de Boer, W. van der Hoek, and J.-J. Ch. Meyer. Agent programming in 3APL. *Autonomous Agents and Multi-Agent Systems*, 2(4):357–401, 1999.
13. JADE — Java Agent DEvelopment Framework. `http://jade.tilab.com`.
14. Jason — a Java-based interpreter for an extended version of AgentSpeak. `http://jason.sourceforge.net`.
15. Alexander Pokahr, Lars Braubach, and Winfried Lamersdorf. Jadex: Implementing a BDI-infrastructure for JADE agents. *EXP - In Search of Innovation*, 3(3):76–85, 2003.
16. Anand S. Rao. AgentSpeak(L): BDI Agents Speak Out in a Logical Computable Language. In *Agents Breaking Away — Proc. Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW)*, volume 1038 of *Lecture Notes in Computer Science*, pages 42–55. Springer, 1996.
17. Y. Shoham. Agent-oriented programming. *Artificial Intelligence*, 60:51–92, 1993.