A Common Semantic Basis for BDI Languages[*]

Louise A. Dennis[1], Berndt Farwer[2], Rafael H. Bordini[2],
Michael Fisher[1], and Michael Wooldridge[1]

[1] Department of Computer Science, University of Liverpool, UK
[2] Department of Computer Science, University of Durham, UK
Contact: lad@csc.liv.ac.uk

Abstract. We describe the design of an intermediate language (AIL) for BDI-style programming languages. AIL is not intended as yet another programming language, but is meant to provide a common semantic basis for a number of BDI programming languages in order to support both formal verification and the transfer of concepts and developments. We examine some of the key features of AIL, unifying a wide variety of structures appearing in the operational semantics of BDI programming languages. In particular, we highlight issues in the treatment of events, goals, and intentions, which are central to the design of these languages.

1 Introduction

As the concept of an "agent" becomes more popular, so the variety of programming languages based upon this concept increases. These *agent-oriented* programming languages range from minimal extensions of JAVA through to logic-based languages for "intelligent" agents [1, 14]. In our work, we are particularly concerned (at least initially) with approaches based on *rational agent theories* [28], primarily the *BDI theory* developed by Rao and Georgeff [23]. Such languages not only incorporate the autonomous behaviour required for the agent concept, but also provide sophisticated mechanisms for instigating, controlling, and reasoning about such behaviours.

Although programming languages based on the BDI approach (let us call these *BDI languages*) are increasingly popular, there are several problems, for example:

1. there are *too* many languages – consider all the varieties described in [1];
2. many of the languages are similar, yet subtly different – this makes it difficult for developers to learn more than one language, as they are not based on agreed notions/definitions; further, such differences make it difficult to identify precisely the general mechanisms and to transfer new techniques between languages; and
3. despite the fact that many BDI languages have logical semantics and utilise logical mechanisms, formal verification tools are rare.

This last aspect is particularly important, since BDI approaches are increasingly used in complex, critical applications such as space exploration [20, 5, 24].

In our work[1] we aim to design an intermediate language (called AIL– *Agent Infrastructure Layer*) for BDI-style programming languages. There are several motivations for this, including:

---

[1] See http://www.csc.liv.ac.uk/~michael/mcapl06 for details.

– providing a common semantic basis for a number of BDI languages, thus clarifying issues and aiding further programming language development;
– supporting formal verification by developing a *model-checker* optimised for checking AIL programs – existing BDI languages can have language-specific compilers for AIL so as to take advantage of its associated model-checker; and
– providing, potentially, a high-level virtual machine for efficient and portable implementation.

Rather than attempting to cover all BDI languages from the start, we have initially tackled some of the most popular. Thus, we have principally referred to the variant of AgentSpeak [22] used in *Jason* [3] and 3APL [18, 8] when designing the semantics for the AIL, but have also taken Jadex [21] and (Concurrent) METATEM [13] into account. However, we expect that a significant proportion of the existing programming languages for multi-agent systems will have mappings into AIL in the future.

The current design for AIL, in the form of an extensive operational semantics, can be found in [10]. For the sake of space, in this paper we only discuss the main aspects of AIL and introduce only the most important rules of the operational semantics. In order to model a particular language in AIL it will be necessary to create a custom AIL compiler for that language. It may also be necessary to provide some custom JAVA classes for the language although these will, in general, be specific to a particular *interpreter* for the language rather than the language itself. We intend to provide such classes and compilers for AgentSpeak and 3APL, though this work remains to be done. The correctness of these compilers will then also need to be addressed. One of the reasons why AIL is to be implemented as a JAVA library is that we aim to use JPF [2] [26] as a target model checker for programs written in various BDI languages.

Sometimes it will prove possible to map only fragments of a given language into AIL. Our expectation is that large and useful fragments of most BDI-style agent programming languages will be translatable. In order to accommodate the main features of the primary BDI languages, AIL has some components with overlapping functionality.

The structure of the remainder of this paper is as follows. In Section 2, we will describe the key similarities in the programming languages considered, which will in turn provide the basis for AIL. Section 3 then describes the core features and operational semantics of AIL. Within AIL, certain language design decisions are required; those related to plan revision in particular are highlighted in Section 4. Finally, in Section 5, we provide concluding remarks, outline future work, and point to aspects of AIL not covered in this paper.

2 General Similarities

There are some general concepts that are found in many BDI languages. We will review these similarities and discuss the design implications for AIL.

Formula Representation. 3APL, AgentSpeak, and METATEM all use minor variations on first order literals for the representation of beliefs, goals, actions, etc. Jadex uses an internal JAVA representation but fragments of this can be mapped into first order logic [4]. Therefore we have chosen first order literals as the basic representation.

---
[2] http://javapathfinder.sf.net

Beliefs. All these languages have the concept of a *belief base*, generally considered as a set of (belief) formulæ. A formula is considered to be believed if it is (unifiable with) a formula in this set. In some languages there is extra reasoning machinery on top of this. In both AgentSpeak and 3APL this additional machinery is a Prolog-style reasoning engine which we have therefore adopted for AIL.

Goals. All the BDI languages have the concept of goals – states of the world the agent is trying to bring about. The precise internal representation of goals differs but all the languages we have considered maintain sets of *outstanding goals*. In general, the languages (with the exception of METATEM) also maintain a stack (or set of stacks) of *deeds*[3] to be performed in order to achieve these goals – these deeds may include committing to the achievement of further sub-goals. Informally, an agent's reasoning cycle involves either adding new deeds to this stack (triggered by the creation of a sub-goal) or removing deeds from the stack (as actions are performed and goals achieved).

In [9] goals are categorised into four types: *achieve*, *perform*, *maintain* and *query*. When an achieve goal appears in a deed stack it must be believed before it can be removed. This contrasts with a perform goal which is removed as soon as new deeds are added to the stack as a result of generating an intention from a suitable plan. Query goals are used to query the belief base, usually in order to obtain instantiations for variables. Maintain goals only trigger plan execution if they cease to be believed.

Terminology and semantics in this area is quite subtle, sometimes also referring to *events* (AgentSpeak, 2APL [6]). In AgentSpeak, events refer both to commitment to achieving goals and changes perceived in the environment. There are also many ways of managing the relationship between (outstanding) goals, sub-goals, and the deeds associated with achieving them. Outstanding goals are those to which the agent has committed but not yet achieved. This places a design burden on AIL, as it must:

– allow outstanding goals to be identified;
– be possible to link a given outstanding goal with the sequence of deeds currently being pursued in order to achieve it;
– maintain sequences of deeds to be performed (including committing to new goals).

Actions. Actions are performed by an agent in the "outside world", i.e., the environment where the agent is situated. The only effect an action has on the working of AIL is that it may return a unifier for some variables (as this is allowed in some of the languages, but not all) and, of course, it may be deemed to have succeeded or failed. In some languages, actions have specific effects on the belief base (e.g., 3APL *capabilities*[4]); such actions can be modelled as plans (see next point).

Plans. The word 'plan' is overloaded among BDI languages and can be used to represent either something that a programmer writes to describe how particular goals should be tackled, or an agent's internal deed stack of pending actions. We have chosen to use *plans* for the first of these, and *deed stack* for the second.

---

[3] The term "deed" has not been used in the agent programming language literature, to our knowledge, but we have adopted it as a way to refer to the various types of formula one can typically have in the body of plans.

[4] A 3APL capability is an "internal" action which alters an agent's beliefs about the world.

BDI languages have plans which are triggered according to aspects of the agent's state, typically the existence of an outstanding goal. Such plans are of the form

(trigger,guard,body)

where the guard is some set of literals that should be believed for the plan to be deemed applicable. If a plan is selected, the plan body is placed on the relevant deed stack.

Jason also allows plans (and therefore deed stacks) to include belief update information and so this is also permitted in AIL. This allows us to model actions with side effects (and specifically 3APL capabilities) within our definition of plans.

It should be noted that we do not intend humans to write native AIL code, so we are able to ignore features which help a programmer conceptually differentiate between aspects of the language, as is the case with 3APL plans and capabilities.

As well as having plans triggered by outstanding goals, AgentSpeak allows plans to be triggered by changes in the belief base. 3APL allows *plan revision* rules/plans which match the prefix of the deed stack and replace it with some alternative. Jadex and METATEM have *constraint* rules/plans which are triggered by some specific configuration of the belief base alone. In order to represent these different types of plans within AIL, we need to make a number of generalisations. We assume a set of *intentions*, each composed, among other things, of a stack of *events* (such as outstanding goals and sub-goals or information about belief updates) and a stack of deeds. The structure of intentions will be further explained in Section 3.1 and later exemplified in Section 3.4. In this set, what some languages (such as AgentSpeak) call an "event" can be represented as an AIL intention with an empty deed stack. We assume that a *current intention*, thus also a *current event* and a *current deed stack*, has been selected from this set.

Each plan in the agent's plan library is represented by a tuple consisting of a trigger event, a deed stack (called the *prefix*), a stack of belief expressions (called the *guard stack*), and a (second) deed stack (called the *body*). The trigger must match the current event, and the prefix must match the prefix of the current deed stack for the plan to be deemed relevant. The belief expression at the top of the guard stack must also be believed by the agent. When this happens, the prefix is dropped from the current deed stack and replaced with the body. Each new deed is paired with the corresponding guard (i.e., belief expression) from the guard stack. Through the use variables in triggers and empty prefixes, this structure allows us to model many different types of plan.

We use a guard stack in order to model the different semantics for guards. Some languages (e.g., Jadex) have *invariant expressions* that must be checked at every stage during the execution of a deed stack, while others (e.g., AgentSpeak and 3APL) check guards only when a plan is to be adopted. When a plan does have an invariant expression, that expression is paired with every deed on the stack. For normal plans (i.e., those with only a guard and no invariants), only the first deed is paired with the guard expression; the remaining deeds are simply paired with $\top$ ('true', denoting an empty guard). Once again, since humans are not expected to write native AIL plans, the tedium of repeating the guard multiple times in order to represent a Jadex invariant is not an issue.

Applicable Plans. Most of the BDI languages employ the concept of generating an *applicable* plan for achieving an outstanding goal. This is based on matching the plan's trigger, or prefix (to generate *relevant plans*) and then checking the guard (to generate *applicable plans*). These BDI languages rely on user-defined methods used by the inter-
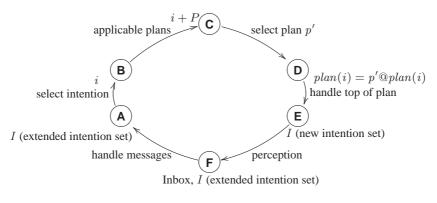
Fig. 1. AIL's deliberation cycle

preter to select *one* of the appropriate plans from the plan library, which then is used to generate a new deed stack. However, METATEM generates all possible next states (deed stacks). In particular, it instantiates all the relevant plans, in some cases generating several potential new deed stacks for a single plan, and then chooses between these based, among other things, on how many outstanding goals are achieved by each option. We adopt this as a more general solution.

3 Agent Infrastructure Layer

AIL's reasoning cycle may informally be viewed as shown in Figure 1.

In this cycle, starting at stage **A**, an *intention* – which includes a deed stack – is selected, leading to stage **B**. Using the agent's plan library and belief base, a set of applicable plans (P in Figure 1) is generated (stage **C**). From this, a single applicable plan is selected and its deed stack joined to the current deed stack (**D**). The top *deed* in this stack is then handled in the appropriate way (depending on the type of formula) and the set of intentions updated accordingly (**E**). Next, perception takes place, posting new events (i.e., belief updates), leading to stage **F**. At this final stage, agent communication messages are handled and the reasoning cycles restarts.

When events are generated from perception of the environment (from **E** to **F**), they are treated as intentions with empty plans. Agents have a message "inbox" where agent messages are placed. Any messages received during the last cycle are handled just before another reasoning cycle starts; this may also extend the intention set.

Since AIL is designed as a basis for efficient verification and not as a programming language to be used in developing agent-based systems, some parts of AIL programs are essentially syntax-less (e.g., plans are represented directly as data structures in AIL). We summarise AIL in the following sections.

3.1 Intentions: Events, Goals and Deed Stacks

The concept of an *intention* is common in BDI languages and is used to represent the *intended means* for achieving goals – intentions include what we call a deed stack, but

may also maintain information about the sub-goal they are intended to achieve or the event that triggered them.

In AIL, we treat intentions as a complex abstract data structure. This data structure aggregates the information about events, outstanding goals, and deed stacks used by the various BDI languages we have considered. As suggested above, we use the idea of events to represent outstanding goals (as, e.g., in AgentSpeak).

AIL intentions may most simply be viewed as a matrix structure consisting of four columns in which we record events, guards, deeds, and unifiers (respectively). These columns form an event stack, a guard stack, a deed stack, and a unifier stack. There are as many rows in the matrix as there are deeds (in the bodies of the plan instances that have become intentions) and events that have not been dealt with yet. Individual rows in the intention associate a particular deed with the event that has caused the deed to be placed on the intention, a guard, and a unifier; new events are associated with an empty deed. An example of the use of this data structure can be found in Section 3.4. The actual implementation of intentions is likely to be more compact than this – for instance the commitment to achieving a goal (i.e., an event) will generally cause a stack of deeds (the plan body) to be joined to the intention's deed stack, all of which will get associated with the same event; that is, each new deed generates a new row in the matrix and the event is repeated in all those rows. Information about outstanding goals can be extracted from the event stacks of all intentions, which record the agent's existing goals and the sequence of unachieved sub-goals generated in pursuit of these.

3.2 Interpreter Specifics

We already noted that many interpreters for BDI languages delegate plan selection to user-defined methods. *Jason* also defers intention selection to such methods. We have chosen to provide simple defaults for such functions (in each case the default selects the top of the stack) but allow them to be overridden. In some cases, such as METATEM, which has specific phases in which only certain plans are applicable, it will be necessary to override these defaults when theoretically modelling the language.

3.3 Operational Semantics

In this section we present a simplified outline of the operational semantics for AIL. The full semantics is available as a technical report (see [10]); we here focus on key issues and semantic rules.

We view an agent as a tuple consisting of an identifier ag, intentions (including a current intention), applicable plans, a belief base, plan library, and a tag indicating the current stage of the agent's reasoning cycle. For presentation reasons we will only show those parts of the state directly relevant to a rule.

Suppose we have already selected an intention (i.e., we are at stage **B** in Figure 1). We use the following rule to generate all applicable plans.

$$\frac{P' = \mathbf{filter}(appPlans(ag)) \quad P' \neq \emptyset}{< ag, P, \mathbf{B} > \rightarrow < ag, P', \mathbf{C} >} \quad (1)$$

filter is an AIL function that, by default, is the identity mapping, but can be overridden by a particular interpreter to remove some of the plans which AIL considers applicable (e.g., ones which have already been attempted).

The AIL function $appPlans$ generates the union of two sets.

$$appPlans(ag) = match\_plans(ag) \cup continue(ag) \tag{2}$$

Informally $match\_plans(ag)$ produces all the plans applicable to the current intention by inspection of the plan library. In contrast, $continue(ag)$ produces the plans which result from continuing to process the current deed stack.

The plans generated by $appPlans$ are tuples consisting of the event, deed stack, guard stack, and unifier to be added to the current intention. However, they also include a number (n), representing a number of rows to be dropped from the current intention before this plan is added (typically, this number would be 1, to remove the ϵ "no plan yet" marker; see semantic rule (4)). The need for this is discussed further in Section 4.1.

The interpreter then selects one of these applicable plans, drops the specified number of rows from the current intention, and replaces them with a new row for each deed in the plan's deed stack (paired with the event, unifier, and appropriate guard as supplied by the plan). In the next semantic rule, the selection function '$\mathcal{S}_{\text{plan}}$' defaults to selecting the top plan in the stack but may be overridden if required by a particular application. We use 'i' to denote the current intention.

$$\frac{\mathcal{S}_{\text{plan}}(P) = (<e, ds, gs, \theta, n>)}{<ag, i, P, \mathbf{C}> \rightarrow <ag, (e, ds, gs, \theta) \ @ \ \texttt{drop}(n, i)[\theta^{\text{hd}}/\theta], [], \mathbf{D}>} \tag{3}$$

The top n rows (as specified in the plans generated from $appPlans$) are dropped from the intention stack ($\texttt{drop}(n, i)$), the top unifier on the unifier stack of this new intention[5] is replaced by θ ($[\theta^{\text{hd}}/\theta]$) and the new intention segment (e, p, gs, θ) is joined to the front of the intention stack (@). The set of applicable plans is emptied. The plans provided to the agent by the programmer remain in its plan library.

Then the top of the plan is handled by a variety of rules. The following rule shows how to handle an (achieve) sub-goal not yet achieved. Recall from our discussion of $appPlans$ that 'ϵ' is a special symbol used to denote "no plan yet". In our semantics we use the syntax $+!_a g$ to signify the adoption of an achieve goal (a for "achieve"), g. This is a deed when it appears in the deed stack of an intention and an event when it appears in the event stack – its type is determined by context. When $+!_a g$ appears in the event stack of an intention we may say that the agent has committed to achieving the goal. The full syntax for AIL can be found in [10].

$$\frac{ag \models gu, \quad ag \not\models g}{<ag, (e, +!_a g, gu, \theta); i, \mathbf{D}> \rightarrow <ag, (+!_a g, \epsilon, \top, \theta); (e, +!_a g, gu, \theta); i, \mathbf{E}>} \tag{4}$$

The rule pushes "no plan yet" on top of the intention's deed stack. This is associated with the event $+!_a g$ (i.e., the commitment to achieving g) and an empty guard. Note that '$\models$' is used to represent the AIL belief checking process. Thus, "$ag \models gu$" asserts

---

[5] see Section 3.1 for a description of the intention data structure.

that the agent believes the guard to be true, while "$ag \not\models g$" asserts that the agent does not believe g, which can be interpreted as the agent not believing the goal has been achieved. Belief checking may cause the instantiation of variables.

If a goal is achieved, then we remove it from the top of the intention

$$\frac{ag \models gu, \quad ag \models g}{< ag, (e, +!_a g, gu, \theta); i, \mathbf{D} > \rightarrow < ag, i[\theta^{\mathrm{hd}(i)}/\theta \cup \theta^{\mathrm{hd}(i)}], \mathbf{E} >} \quad (5)$$

Because we want to preserve any decisions about unifiers, the unifier associated with $+!_a g$ is merged with the top unifier of i (the tail of the intention).

It is worth noting here that AIL does not distinguish between *achieve* and *query* goals. Query goals are easily handled by (5), since '$\models$' instantiates variables. AgentSpeak even allows query goals to act as trigger events and match plans if they do not succeed, so (4) is also used. *Perform* goals can be handled by a simple modification to (4) which does not leave $+!_p g$ on the stack. Only *maintain* goals need to be treated entirely separately. In AIL, maintain goals insert a new constraint plan in the library which fires whenever the goal is no longer believed (details of this can be found in [10]).

Beliefs. All the languages we considered allow new beliefs to be inserted into, and removed from, the agent's belief base. However, some (e.g., AgentSpeak) also allow new plans to be placed in the plan library. We have therefore generalised the concept of belief to include many aspects of an agent's internal state, such as the plan library. Belief updates (i.e., the addition of new beliefs, or the deletion of old beliefs) are tagged by the relevant part of the state (e.g., $+b^{bb}$ is an instruction to add b to the belief base, while $+p^{pl}$ is an instruction to add p to the plan library). We have found by this mechanism that all such updates can be handled essentially by the same rule, the only difference being the state component that is selected. Rule (6) shows the special version of this general rule for adding a belief to the belief base[6].

$$\frac{ag \models gu}{\begin{array}{c} < ag, (e, +b^{bb}, gu, \theta); i, I, B, \mathbf{D} > \rightarrow \\ < ag, i[\theta^{\mathrm{hd}(i)}/\theta \cup \theta^{\mathrm{hd}(i)}], (+b^{bb}, [\epsilon], \top, \emptyset); I, B \cup \{b\}, \mathbf{E} > \end{array}} \quad (6)$$

This causes the top of the current intention to be removed as in rule (5), and also causes b to be added to the belief base. However, since a belief update may be a trigger for a plan, we also place a new intention on the intention stack $(+b^{bb}, [\epsilon], \top, \emptyset)$, which has a "no plan yet" deed stack. This, recall, is how events are represented in AIL.

3.4 Example

We now illustrate the operation of an AIL agent via a simple example. This is loosely based on a 3APL example available in its user guide[7]. A robot has a goal to clean rooms. When the robot believes a room is dirty, the plan is to go to that room and vacuum clean

---

[6] AIL's actual semantics allows multiple belief updates of mixed types at once resulting in a rather complex rule but (6) captures the key idea applied to a single update.
[7] http://www.cs.uu.nl/3apl/download/java/userguide.pdf

it. There is insufficient space here to discuss a translation from 3APL to AIL although we will briefly touch on some of the more interesting issues.

The robot possesses the following plans for cleaning rooms and changing locations.

PLAN 1:	
trigger	`+!`$_a$`clean()`
prefix	`[`ϵ`]`
guard stack	`dirty(Room)`
	`TRUE`
body	`+!`$_a$`Goto(Room)`
	`+!`$_a$`Vacuum(Room)`

PLAN 2:	
trigger	`+!Goto(R)`
prefix	`[`ϵ`]`
guard stack	`pos(P)`
	`TRUE`
	`TRUE`
body	`-pos(P)`
	`+pos(R)`
	`+Goto(R)`

We represent the components plans in table form showing the components introduced in Section 2. Since 3APL guards are only checked once the guard is only associated with the top deed. Note that 3APL goals such as `+!`$_a$`clean()` are 'achieve' goals and it is expected that the truth of `clean()` will be established during execution. (In the sequel we assume all goals to be achieve goals and drop the subscripts.)

Plan 2 is derived from a 3APL capability. The semantics of capabilities is given in a Hoare-triple like format, for example: `{pos(P)} Goto(R) {NOT pos(P), pos(R)}`. The plan shows how this is can be transformed into AIL.

Let us consider the execution of an AIL agent which starts out with the goal `clean()` and the beliefs `pos(room3)` and `dirty(room1)`. We examine the intention stack since this is of the most interest in understanding the execution of an AIL agent. We represent individual intentions as a matrix with four columns as discussed in Section 3.1.

Initially there is one intention, and this has one row to achieve the goal `+!clean()`. The event is the start of the program. The guard and unifier stacks are initially empty (left-hand table below). Since the agent does not believe `clean()`, ϵ is placed on top of the plan according to rule (5) with the trigger event noting the need to achieve `clean()`.

trigger	deed	guard	unifier
start	`+!clean()`	$\top$	$\emptyset$

$\xrightarrow{(5)}$

trigger	deed	guard	unifier
`+!clean()`	ϵ	$\top$	$\emptyset$
start	`+!clean()`	$\top$	$\emptyset$

Plan 1 now matches the intention. The 'ϵ' is dropped (since it matches the prefix) and the plan's new deed stack is joined to the intention's remaining deed stack. All the deeds in this new stack are associated with the plan's trigger.

trigger	deed	guard	unifier
`+!clean()`	`+!Goto(Room)`	`dirty(Room)`	`Room = room1`
`+!clean()`	`+!Vacuum(Room)`	$\top$	`Room = room1`
start	`+!clean()`	$\top$	$\emptyset$

This process then repeats to plan `Goto`:

trigger	deed	guard	unifier
`+!Goto(R)`	`-pos(P)`	`pos(P)`	`Room = room1, P = room3, R = room1`
`+!Goto(R)`	`+pos(room1)`	$\top$	`Room = room1, P = room3, R = room1`
`+!Goto(R)`	`+Goto(room1)`	$\top$	`Room = room1, P = room3, R = room1`
`+!clean()`	`+!Goto(Room)`	`dirty(Room)`	`Room = room1`
⋮	⋮	⋮	⋮

AIL now performs the belief updates on the deed stack. These generate new intentions according to rule (6); let us assume these are not prioritised by the intention selection process so the intention stack becomes:

trigger	deed	guard	unifier
`+!Goto(R)`	`+Goto(R)`	$\top$	`Room = room1, P = room3, R = room1`
`+!clean()`	`+!Goto(Room)`	`dirty(Room)`	`Room = room1`
`+!clean()`	`+!Vacuum(Room)`	$\top$	`Room = room1`
start	`+!clean()`	$\top$	$\emptyset$
`-pos(room3)`	ϵ	$\top$	$\emptyset$
`+pos(room1)`	ϵ	$\top$	$\emptyset$

When we handle this last belief update the unifier is merged into the one for the top of the first sub-plan, preserving any unifications obtained.

trigger	deed	guard	unifier
`+!clean()`	`+!Goto(Room)`	`dirty(Room)`	`Room = room1, P = room3, R = room1`
`+!clean()`	`+!Vacuum(Room)`	$\top$	`Room = room1`
start	`+!clean()`	$\top$	$\emptyset$
⋮	⋮	⋮	⋮

For lack of space we cannot expound on this example any further.

4 Plan Failure and Plan Revision

In most BDI languages it is assumed, in general, that once an agent has committed to a goal, the goal is not abandoned. However, in reality, it is sometimes necessary to reconsider intentions. Unfortunately, the literature on agent programming languages is mostly vague about this process.

4.1 Plan Revision

3APL uses *plan revision rules* to replace the prefix of whole intentions with revisions. This influenced the design of AIL plans.

Consider an intention to give Jane a present, which has formed the deed stack: check what is in the Harrods department store, go to London, and purchase the gift. So our intention stack (ignoring guards) is represented as follows:

trigger	deed	unifier
`+!give(X1, Y1)`	`+!in_harrods(Y1)`	`X1 = jane, Y1 = X`
`+!give(X1, Y1)`	`gotolondon`	`X1 = jane, Y1 = X`
`+!give(X1, Y1)`	`purchase(Y1, harrods)`	`X1 = jane, Y1 = X`
start	`+!give(jane, X)`	$\emptyset$

Let us suppose that achieving `+!in_harrods(Y1)` instantiates `Y1` to 'computer' yielding the new intention stack:

trigger	deed	unifier
`+!give(X1, Y1)`	`gotolondon`	`X1 = jane, Y1 = X, Y1 = computer`
`+!give(X1, Y1)`	`purchase(Y1, harrods)`	`X1 = jane, Y1 = X`
start	`+!give(jane, X)`	$\emptyset$

Suppose also that we have a plan revision rule that says that instead of going to London and buying a computer in Harrods we should, instead, purchase it from Dell:

PLAN 3:	
trigger	`Any`
prefix	`gotolondon`
	`purchase(computer, harrods)`
guard stack	TRUE
body	`purchase(computer, dell)`

The prefix is of length 2 so we drop two items from our intention. The last trigger of the dropped section is `+!give(X1, Y1)` so that unifies with `Any` and we replace the dropped parts of the stack with the new deed stack:

trigger	deed	unifier
`Any`	`purchase(Y1, dell)`	`X1 = jane, Y1 = X, Y1 = computer, Any = +!give(X1, Y1)`
start	`+!give(jane, X)`	$\emptyset$

This has preserved the unifications already decided upon (e.g., that `Y1 = computer`)[8].

4.2 Plan Failure

The original AgentSpeak specification [23] includes a "drop goal" construct in its syntax but its semantics has never been made clear and therefore it is often ignored in attempts to model the language. For instance [17], which embeds AgentSpeak in an early version of 3APL, ignores this aspect of the AgentSpeak semantics. The *Jason* interpreter [2] for AgentSpeak posts drop goal ($-!g$) events when a plan fails [19]. There are no default rules for handling these events but it is possible to write handlers as a plan, for instance:

$$-!g:true <- +!g$$

which forces backtracking[9], or other plans for handling failure. While there is no default backtracking behaviour in either AgentSpeak or 3APL, METATEM uses backtracking as a default revision procedure.

It seemed necessary to provide a mechanism by which the designer of an AIL compiler may define plan failure handling behaviour without providing unnecessary additions to the language. This meant that plan failure had to be defined by plans. We therefore needed to introduce a distinguished symbol 'backtrack' into our deed syntax which, if used, causes the execution of the AIL operational semantics rules to systematically retrace their steps attempting different instantiations and rules, as in traditional backtracking.

We adopt the *Jason* idea of posting a drop goal event when applicable plans cannot be found or actions fail. When this happens the current trigger event is selected and posted as a drop goal. A particular AIL interpreter need never select such events for handling. However, if a drop goal event *is* selected, then it is matched with *all* outstanding

---

[8] This does mean that incautious use of plan revision can preserve unexpected unifications.

[9] Note that this backtracking only retries the goal – the programmer must enforce the use of a different plan or this could potentially cycle.

intentions to see if it unifies with an event (i.e., one of the goals or sub-goals to which the intention has committed). If it does, 'ϵ' is placed on top of the plan for that intention with $-!g$ as its trigger. We plan to extend the semantics to allow the option of unifying with just one intention. This allows us to model 3APL's drop goal constructs[10].

Any plans available for dropping goals can then be applied at the applicable plan stage. Possible plans include:

PLAN: Actually drop a goal	
trigger	$-!g$
prefix	ϵ
guard stack	TRUE
body	$-!g$

PLAN: Retry a goal	
trigger	$-!g$
prefix	ϵ
guard stack	TRUE
body	$+!g$

PLAN: Traditional backtracking	
trigger	$-!g$
prefix	ϵ
guard stack	TRUE
body	backtrack

The first of these will place $-!g$ on top of the deed stack. We have specified the handling of a $-!g$ deed in AIL as dropping everything on the goal stack after that goal was first committed to. This also drops all unifiers allowing different rules to be used.

$$\frac{ag \models gu \qquad +!g = \texttt{events}(i)[n] \qquad \forall m > n.\neg(+!g = \texttt{events}(i)[m])}{< ag, (e, -!g, gu, \theta); i, \mathbf{D} > \rightarrow < ag, \texttt{drop}_e(n, i), \mathbf{E} >} \quad (7)$$

where $\texttt{events}(i)[n]$ is the nth trigger event on the intention stack.

Let us reconsider purchasing the present for Jane, suppose we are unable to get to London. The failure of the action `gotolondon` will post a new "drop goal" intention:

trigger	deed	unifier
`-!give(X1, Y1)`	ϵ	`X1 = jane, Y1 = X, Y1 = computer`
`+!give(X1, Y1)`	`gotolondon`	`X1 = jane, Y1 = X, Y1 = computer`
`+!give(X1, Y1)`	`purchase(Y1, harrods)`	`X1 = jane, Y1 = X`
start	`+!give(jane, X)`	$\emptyset$

Assuming this intention is selected a new merged intention is generated

trigger	deed	unifier
`-!give(X1, Y1)`	ϵ	`X1 = jane, Y1 = X, Y1 = computer`
`+!give(X1, Y1)`	`gotolondon`	`X1 = jane, Y1 = X, Y1 = computer`
`+!give(X1, Y1)`	`purchase(Y1, harrods)`	`X1 = jane, Y1 = X`
start	`+!give(jane, X)`	$\emptyset$

Upon using the plan "Actually drop a goal" above, we arrive at:

trigger	deed	unifier
`-!give(X1, Y1)`	`-!give(X1, Y1)`	`X1 = jane, Y1 = X, Y1 = computer`
`+!give(X1, Y1)`	`gotolondon`	`X1 = jane, Y1 = X, Y1 = computer`
`+!give(X1, Y1)`	`purchase(Y1, harrods)`	`X1 = jane, Y1 = X`
start	`+!give(jane, X)`	$\emptyset$

Now, (7) causes us to drop back to the first appearance of `+!give(X1, Y1)`:

trigger	plan	unifier
start	`+!give(jane, X)`	$\emptyset$

We have lost our commitment to giving Jane a computer (since it is such commitments that may have caused failure).

---

[10] Mehdi Dastani, Personal Communication.

5 Concluding Remarks

This paper provides an overview of our *Agent Infrastructure Layer* (AIL), capturing all major features of common BDI languages. The main purpose of AIL is to provide a common (operational) semantics for large fragments of these languages in order to aid the transfer of new ideas and techniques and to allow the development of common verification tools and technologies. The development of AIL has highlighted several subtle language design decisions, which we have described in the paper. In this way, AIL serves a valuable role in clarifying and formalising BDI language semantics.

In order to provide this semantics, we needed to characterise the shared concepts of beliefs, goals, actions, and plans as well as accounting for common variations such as the use of events and deed stacks. Thus, our semantics develops a complex data structure to represent intentions associating events (which include outstanding goals) with stacks of deeds (which include belief updates) to be performed. A generalised notion of a plan is developed to be used in this data structure which captures many of the notions of plans available in the literature.

While we have described aspects relating to goals, beliefs, plans, etc. *within* agents, AIL itself covers much more than we addressed in this paper [10]. Three important aspects that are omitted are mentioned briefly below.

Constraints. An additional construct within the agent's state is actually provided within AIL. *Constraints* describe pre-conditions that must hold before a given action may be performed or a goal may be adopted. These preconditions are checked just like the guards of plans and it is here that we particularly expect the extended notion of belief to become useful (so constraints may express that the agent has particular goals or particular plans in its library). It is important to note that whereas an agent selects *only one* applicable plan it must satisfy *all* relevant constraints. The generalised notion of constraint allows us to express a wide variety of permissions and prohibitions. If an action is *prohibited*, the pre-condition is simply $\perp$ (false), so it always fails and the action is never taken (or the goal never adopted). If certain actions are only permitted in certain situations, or to agents who have adopted certain roles, these can also easily be modelled (e.g., an agent can check if it is performing the appropriate role). The operational semantics of AIL, therefore, forces an agent to check if there are any constraints and, if so, to see that they hold before it takes an action or selects a plan.

Communication. Armed with constraints, we are able to describe a wide range of communication protocols. A common concept among BDI languages is that messages should contain both *content* and a *performative* (which determines what should be done with the content). Communication protocols are established by agreeing on constraints associated with these messages (e.g., which performatives can be used in a given stage of a communication protocol) and associating particular plans to be enacted on their receipt. Variations on these basic ideas are present in [27, 3, 12].

In this model a communication protocol would consist of a selection of plans and constraints on `send` actions and `received` events. Sending messages is treated as an action by AIL, and constraints are checked in the same way as they are for any action. The

last phase of the AIL reasoning cycle is dedicated to handling the receipt of messages.

$$I' = \{(+\texttt{received}(ag', ilf, \phi), [\epsilon], \top, \emptyset)|$$
$$< ag', ilf, \phi > \in In \ \land \ check\_constraints(+\texttt{received}(ag', ilf, \phi)\}$$
$$\overline{< ag, I, In, \mathbf{F} > \rightarrow < ag, I'@I, [], \mathbf{A} >} \tag{8}$$

In this rule the intention stack, I is extended with a set of $+\texttt{received}$ events, one for each message in the inbox whose relevant constraints are satisfied. These events can then trigger appropriate plans for reacting to the message. The use of constraints allows AIL to filter out certain messages; this allows AIL to handle concepts such as the social acceptability of messages which are important, for example, in *Jason* [3].

Organisational Structures. We have designed AIL aiming, in future work, not only to be able to accommodate a variety of languages but also to account for future developments of the existing languages. For example, most languages currently concentrate on individual agents, so it is likely that those languages will be extended to include constructs to support the social level of multi-agent systems, particularly the notion of "organisations" [25]. Important common concepts in this area are the ability for agents to form groups which possess and communicate goals, plans, permissions, and prohibitions. Furthermore groups of agents need to be able to organise themselves into organisations, with specific roles within those organisations and specific relationships between roles. All of this implies that such groups adhere to certain communication protocols; [11, 25, 7] all describe variants which rely on these basic constructs as building blocks. Clearly any machinery for organisation and communication within AIL needed, at a minimum, to be able to express these notions and preferably needed to be customisable to allow variations on their basic forms.

AIL is therefore being designed with simple constructs which allow it to model many of the most obvious developments in this area. Of the BDI languages we have examined, only METATEM has any primitives for describing social organisations of agents (all other languages have messaging constructs and many are investigating frameworks for describing organisational structures). AIL's social organisations are currently based on METATEM's groups which flexibly allow the concepts of organisation and role to be captured [16, 15]. In order to properly express permissions and prohibitions it was necessary to provide AIL with constraints as described above. We also needed to annotate aspects of the agent's internal state with sources of information/goals and define a concept of the relevance of a constraint or plan to a situation. The treatment of groups of agents as agents in their own right also provides a natural mechanism for introducing concepts of modularity into agent programs.

Space restrictions preclude further discussion of this important item, but we note that it forms a key part of our future work.

Future Work

As mentioned above, a key aim of this work is to provide a basis for the *formal verification* of programs written in BDI-based programming languages. AIL itself still requires refinement, in particular in the communication and organisation aspects mentioned above. Thus, deeper analysis of these aspects will be carried out, and appropriate high-level primitives will be developed.

Also in the short term, planned work revolves around the implementation of AIL (in JAVA) and the provision of compilers for, at least, significant fragments of AgentSpeak and 3APL. In the longer term, the correctness of these compilers needs to be addressed and verification tools for AIL developed. In particular, we aim to extend JPF [26] so that AIL classes are treated as internal classes of JPF, which should provide for efficient verification of agent programs written in various BDI languages.

An additional aim, within our future work, is to develop a subset of AIL, currently called AIL$^-$, which: captures most *reasonable* BDI programs, has a very clear and straightforward semantics, and is easily implementable. Currently, AIL$^-$ is conceived, in particular, as reducing the number of goal types available and the mechanisms for handling plan failure. It will also eliminate some of the flexibility of the current group structuring mechanisms. AIL$^-$ would then provide the basis for a *lightweight*, *efficient*, and *verifiable* agent programming language.

References

1. R. H. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni, editors. *Multi-Agent Programming: Languages, Platforms and Applications*. Number 15 in Multiagent Systems, Artificial Societies, and Simulated Organizations. Springer-Verlag, 2005.

2. R. H. Bordini and J. F. Hübner. *Jason: A Java-based interperter for an extended version of AgentSpeak*, 2006. Available from `http://jason.sourceforge.net`.

3. R. H. Bordini, J. F. Hübner, and R. Vieira. *Jason* and the golden fleece of agent-oriented programming. In Bordini et al. [1], chapter 1, pages 3–37.

4. L. Braubach, A. Pokahr, and B. Farwer. On Formalising Jadex. Personal Communication, January 2007.

5. W. Clancey, M. Sierhuis, C. Kaskiris, and R. van Hoof. Advantages of Brahms for Specifying and Implementing a Multiagent Human-Robotic Exploration System. In *Proc. 16th International Florida Artificial Intelligence Research Society Conference (FLAIRS)*, pages 7–11. AAAI Press, 2003.

6. M. Dastani. 2APL: A Practical Agent Programming Language. Slides to be Presented at PLDT-MAS Tutorial at AAMAS conference, 2007.

7. M. Dastani, V. Dignum, and F. Dignum. Role-Assignment in Open Agent Societies. In *Proc. 2nd International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*. ACM Press, 2003.

8. M. Dastani, M. B. van Riemsdijk, and J.-J. C. Meyer. Programming multi-agent systems in 3APL. In Bordini et al. [1], chapter 2, pages 39–67.

9. M. Dastani, M. B. van Riemsdijk, and J.-J. C. Meyer. Goal Types in Agent Programming. In *Proc. 17th European Conference on Artificial Intelligence (ECAI)*, 2006.

10. L. A. Dennis. Agent Infrastructure Layer (AIL): Design and Operational Semantics v1.0. Technical Report ULCS-07-001, Department of Computer Science, University of Liverpool, 2007. Available from `http://www.csc.liv.ac.uk/research/techreports/`.

11. J. Ferber, O. Gutknecht, and F. Michel. From Agents to Organizations: An Organizational View of Multi-agent Systems. In *Proc. 4th International Workshop on Agent-Oriented Software Engineering (AOSE)*, volume 2935 of *LNCS*, pages 214–230. Springer, 2003.

12. FIPA. FIPA Communicative Act Library Specification. Technical Report FIPA00037, Foundation for Intelligent Physical Agents, 2002.

13. M. Fisher. METATEM: The story so far. In *Proc. 3rd International Workshop on Programming Multiagent Systems (ProMAS)*, volume 3862 of *LNAI*, pages 3–22. Springer, 2005.

14. M. Fisher, R. H. Bordini, B. Hirsch, and P. Torroni. Computational Logics and Agents — A Roadmap of Current Technologies and Future Trends. *Computational Intelligence*, in press.

15. M. Fisher, C. Ghidini, and B. Hirsch. Programming Groups of Rational Agents. In *Proc. 4th International Conference on Computational Logic in Multi-Agent Systems (CLIMA)*, volume 3259 of *LNCS*, pages 849–856. Springer, November 2004.

16. M. Fisher and T. Kakoudakis. Flexible Agent Grouping in Executable Temporal Logic. In *Proc. 12th International Symposium on Languages for Intensional Programming (ISLIP)*. World Scientific Press, 1999.

17. K. V. Hindricks, F. S. Boer, W. van der Hoek, and J.-J. C. Meyer. A Formal Embedding of AgentSpeak(L) in 3APL. In *Advanced Topics in Artificial Intelligence*, volume 1502 of *LNAI*, pages 155–166. Springer, 1998.

18. K. V. Hindricks, F. S. de Boer, W. van der Hoek, and J.-J. C. Meyer. Agent Programming in 3APL. *Autonomous Agents and Multi-Agent Systems*, 2(4):357–401, 1999.

19. J. F. Hübner, R. H. Bordini, and M. Wooldridge. Programming Declarative Goals using Plan Patterns. In *Proc. 4th International Workshop on Declarative Agent Languages and Technologies (DALT)*, pages 65–81, Hakodate, Japan, May 2006.

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

21. A. Pokahr, L. Braubach, and W. Lamersdorf. A Flexible BDI Architecture Supporting Extensibility. In *Proc. IEEE/WIC/ACM International Conference on Intelligent Agent Technology (IAT)*, pages 379–385, 9 2005.

22. A. Rao. 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)*, volume 1038 of *LNCS*, pages 42–55. Springer, 1996.

23. A. S. Rao and M. Georgeff. BDI Agents: from theory to practice. In *Proc. 1st International Conference on Multi-Agent Systems (ICMAS)*, pages 312–319, San Francisco, CA, June 1995.

24. M. Sierhuis. Multiagent Modeling and Simulation in Human-Robot Mission Operations. (See http://ic.arc.nasa.gov/ic/publications), 2006.

25. J. Vázquez-Salceda, V. Dignum, and F. Dignum. Organizing multiagent systems. Technical Report UU-CS-2004-015, Institute of Information and Computing Sciences, Utrecht University, 2004.

26. W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *Proceedings of the Fifteenth International Conference on Automated Software Engineering (ASE'00), 11-15 September, Grenoble, France*, pages 3–12. IEEE Computer Society, 2000.

27. M. Wooldridge, M. Fisher, M. Huget, and S. Parsons. Model Checking Multiagent Systems with MABLE. In *Proc. 1st International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS)*, July 2002.

28. M. Wooldridge and A. Rao, editors. *Foundations of Rational Agency*. Applied Logic Series. Kluwer Academic Publishers, Mar. 1999.