Agent Infrastructure Layer (AIL): Design and Operational Semantics v1.0*

L. A. Dennis

January 29, 2007

Abstract

This technical report outlines a proposal for an operational semantics for AIL. AIL is intended to be an agent infrastructure layer that will form a common basis into which BDI-style agent programming languages can be compiled. In particular we hope to provide optimised verification support for AIL in the form of a model-checker.

As well as presenting an operational semantics for AIL this technical report is intended to serve as a design document for an implementation. As such some details about data structures are also included.

1 Introduction

This is a proposed operational semantics for AIL. AIL is intended as an *agent infrastructure layer* that will form a common basis into which BDI-style agent programming languages such as those described in [Rao, 1996, Hindricks et al., 1999, Dastani et al., 2005, Pokahr et al., 2005, Fisher, 2005] can be compiled. In particular we hope to provide optimised verification support for AIL in the form of a model-checker. AIL has been developed as part of the MCAPL project [1].

2 The Reasoning Cycle

AIL adopts an informal reasoning cycle shown in figure 1.

In this cycle an *event/plan pair* (plans are viewed as a stack of "things I need to do") is selected. Using the agent's rulebase a set of potential additions to the plan are generated (P above). From this a single plan addition is selected and joined to the current plan. The top "thing I need to do" in this plan is then handled in some fashion and the set of event/plan pairs updated accordingly. This new intention set may have a non-deterministic head which gets resolved when Applicable plans are generated – this enables some of the message semantics we've been considering. Perception takes place which may extend the intention set (informally the set of event/plan pairs). Any messages are then handled this may also extend the intention set. From this set a new current event is selected.

Each stage in this cycle is tagged **A**, **B**, **C**, **D**, **E** or **F** and agent states are tagged accordingly (this restricts the possible transitions available at each state).

AIL treats belief checking as a multi-stage activity so it is necessary to represent this as a stage. Unfortunately belief checking may happen at several points in the cycle. This being the case we treat $\models$ as sugar for belief checking (see §5).

*University of Liverpool, Department of Computer Science, Technical Report ULCS-07-001

[1] EPSRC linked grants EP/D054688/1 and EP/D052548/1. More details can be found at `http://www.csc.liv.ac.uk/~michael/mcapl06.html`.

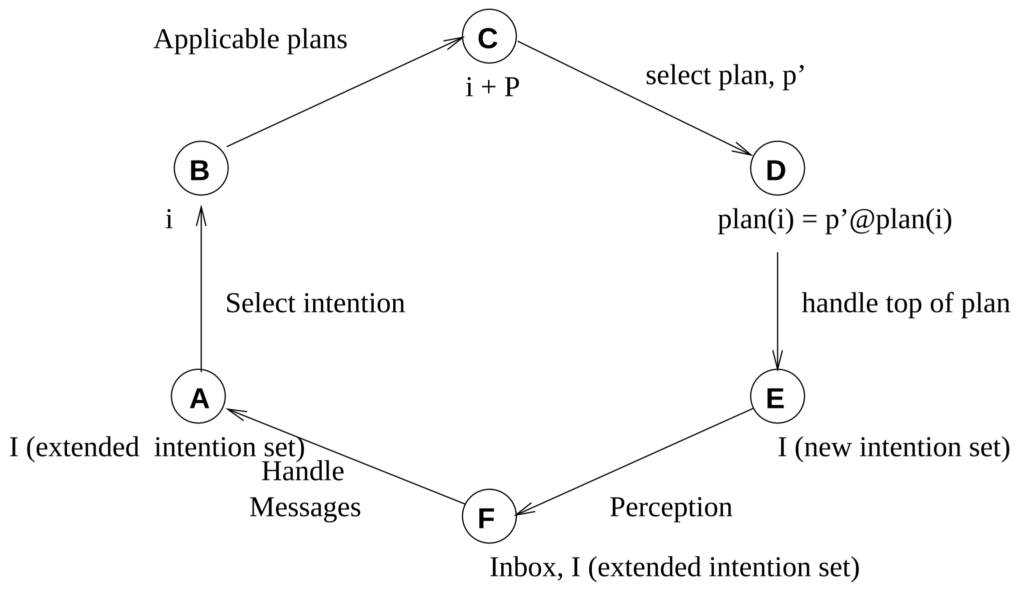

Figure 1: An Informal Presentation of the AIL Reasoning Cycle

3 Syntax

AIL is not intended for programming by humans so a detailed syntax is not strictly necessary. The following represents the syntactic conventions we adopt in this technical report but may not be a reflection of any syntax actually used in the implementation.

3.1 Basic Term Language

var	$\rightarrow$	`string beginning with an upper-case letter`
constant	$\rightarrow$	`string beginning with a lower-case letter`
term	$\rightarrow$	*var*
	$\mid$	*constant*
	$\mid$	$constant(term*)$
literal	$\rightarrow$	*term*
	$\mid$	$\neg term$

3.2 Beliefs

belief type	$\rightarrow$	ex
	$\mid$	br
	$\mid$	c
	$\mid$	p
	$\mid$	cn
	$\mid$	cx

We also have a number of belief cateagories, depending whether there is a belief about the external world (ex) or some aspect of the internal state such as belief rules (br), constraints (c), performatives (p), rules (r), content(cn), context(cx), outbox of sent message (o), intentions (i) or desires (d). This allows common BDI concepts and notations for belief checking and update to be extended to all aspects of the agents states. This provides an elegant way to cover BDI-languages which allow communication of plans (eg. the Jason AgentSpeak interpreter [Bordini and Hübner, 2006] and MetateM)

$$\begin{array}{rcl} \textit{belief} & \rightarrow & \textit{belief}^{\text{belief type}} \\ \textit{belief}^{ex} & \rightarrow & \textit{literal}^{ex} \end{array}$$

$$\begin{array}{rcl} \textit{source} & \rightarrow & \textit{var} \\ & \mid & \textit{constant} \\ \textit{guard} & \rightarrow & \textit{belief}\{\textit{source}\} \\ & \mid & \sim \textit{belief}\{\textit{source}\} \\ & \mid & \textit{guard} \wedge \textit{guard} \\ \textit{belief rule} & \rightarrow & \textit{belief}\ \texttt{:-}\ \textit{belief} * \\ \textit{belief}^{br} & \rightarrow & \textit{belief rule}^{br} \end{array}$$

NB. We are using $\neg$ here to represent "believes not" and $\sim$ to represent "doesn't believe".

3.3 Goals, Actions, Plans etc.

$$\begin{array}{rcl} \textit{goal} & \rightarrow & \textit{literal} \\ \textit{goal type} & \rightarrow & \mathrm{a} \\ & \mid & \mathrm{m} \\ & \mid & \mathrm{p} \\ & \mid & \mathrm{t} \end{array}$$

There are a number of different categories of goal in the literature [Dastani et al., 2006] (achieve, maintain, perform and test) we use subcripts to distinguish between them. We in fact treat most goals as declarative or achieve goals and describe how test and perform goals can be transformed into these. We therefore only provide separate rules for maintain goals.

$$\begin{array}{rcl}
\mathit{action} & \rightarrow & \mathit{term} \\
\mathit{deed} & \rightarrow & +((\mathit{belief}^{\text{belief type}}) * \\
 & | & -(\mathit{belief}^{\text{belief type}}) * \\
 & | & +!_{\mathit{goal\ type}}\mathit{goal} \\
 & | & -!_{\mathit{goal\ type}}\mathit{goal} \\
 & | & \mathit{action} \\
 & | & \epsilon \\
 & | & \texttt{backtrack} \\
\mathit{guard} & \rightarrow & +!_{\mathit{goal\ type}}\mathit{goal}^{i} \\
\mathit{guard} & \rightarrow & \sim +!_{\mathit{goal\ type}}\mathit{goal}^{i} \\
\mathit{guard} & \rightarrow & +!_{\mathit{goal\ type}}\mathit{goal}^{d} \\
\mathit{guard} & \rightarrow & \sim +!_{\mathit{goal\ type}}\mathit{goal}^{d}
\end{array}$$

Deeds are the basic components of plans. They allow for the standard setting of goals, performing actions and updating beliefs. But we have extended them to allow updating of several aspects of the state important for establishing group organisation.

ϵ is a distinguished symbol that is taken to mean "no plan yet". `backtrack` signals that backtracking should take place through the states of the transition system. `backtrack` is a meta-command for which we do not supply a rule in the operational semantics.

3.4 Events

$$\begin{array}{rcl}
\mathit{event} & \rightarrow & +\mathit{belief}\{\mathit{source}\} \\
 & | & -\mathit{belief}\{\mathit{source}\} \\
 & | & +!\mathit{goal}\{\mathit{source}\} \\
 & | & -!\mathit{goal}\{\mathit{source}\} \\
 & | & +?\mathit{goal}\{\mathit{source}\} \\
 & | & -?\mathit{goal}\{\mathit{source}\}
\end{array}$$

The only point where a programmer may directly refer to the source of a belief or goal is as part of an event or a guard. Events are generated by AIL so the programmer can not fake the source of a belief or goal and guards are only used to check beliefs they are not a part of the program state.

3.5 Rules, Messages and Constraints

$$
\begin{array}{rcl}
plan & \rightarrow & event : deed* : guard* : deed * \\
 & | & var : deed* : guard* : deed * \\
belief^{p} & \rightarrow & rule^{p} \\
constraint & \rightarrow & deed\texttt{++}guard \\
belief^{c} & \rightarrow & constraint^{c} \\
message & \rightarrow & < constant, constant, literal, constant > \\
guard & \rightarrow & message^{o} \\
guard & \rightarrow & \sim message^{o}
\end{array}
$$

Plans consist of an event and deed stack against which they are matched to determine relevance. There is then a stack of guards and another deed stack – representing the course of action and checks at each stage of pursuing this course of action.

Constraints specify preconditions that must be true before an action is taken or a deed adopted for planning.

In constraints are to be established dynamically as agents join and leave groups and organisations in the style of METATEM[Fisher, 2005].

Semantics for specific message performatives in the style of the MABLE system [Wooldridge et al., 2002] can be established by grouping constraints on `send` actions and `+received` beliefs and plans associated with these. We should provide some built-in syntax to group such constraints and plans together.

3.6 An Agent

$$
agent\_def \quad \rightarrow \quad belief*, source*, source*, belief\ plan*, goal*, plan*, constraint*
$$

A programmer may specify an agent by providing initial beliefs, content, context, belief rules, goals, plans and constraints.

3.7 Notation

In what follows we generally refer to individual beliefs, goals, actions, events, etc with lower case letters and sets or stacks of beliefs, goals, actions, etc. with uppercase letters (mostly we presume these are stacks but sometimes it is OK to generalise to sets). In general the following letters will be associated with each concept: beliefs (b), goals (g), actions (a), events (e), plans (p), deeds (d) and plans (p).

4 Built-in Functions and Data Structures

4.1 Sets and Stacks

We have generic stack and set data-types with the following associated constructors, destructors and operations:

Stack	
[]	an empty stack
$x;s$	The stack, s, with a new top element, x
$\texttt{hd}(s)$	The top element of a stack, s
$\texttt{last}(s)$	The bottom element of a stack, s
$\texttt{tl}(s)$	The stack, s, without its top element
$\texttt{drop}(n, s)$	Remove the top n elements from the stack, s
$\texttt{prefix}(n, s)$	The top n elements from the stack, s
$s[n]$	The nth element of the stack, s
$s_1@s_2$	s_1 appended to the front of s_2
$\#s$	The number of elements in the stack, s.
$\texttt{empty}(s)$	True if s is empty

Set	
$x \in S$	x is in the set S
$S_1 \cup S_2$	The union of two sets
S_1/S_2	The set S_1 less the set S_2
$in_1(S_1 \times S_2)$	S_1

4.2 Intentions

Definition 4.1 *An* intention *(i) is an abstract data structure which relates a deed stack to a set of triggering events, guards, unifiers and a source. The idea is that an intention has a source (for whom the intention is being performed) and that any deed on the stack can be traced to its* trigger event. *Individual deeds are also associated with unifiers for its free variables and a guard that must be true before it is handled.*

We do not choose here to explicitly state the structure of an intention but any such data structure should support the following operations.

Intention	
$\texttt{deeds}(i)$	returns the intention's deed stack
$\texttt{events}(i)$	returns a stack of the events in the intention with the original goal (or belief update) at the bottom and then subgoals in order generated above that
$\texttt{tr}(n, i)$	returns the event that triggered the placement of the nth deed on deed stack of i.
$\theta(n, i)$	returns the unifier associated with the nth deed in the deed stack
$\theta_e(n, i)$	returns the unifier associated with the nth event on the event stack
$\texttt{gu}(n, i)$	returns the guard associated with the nth deed on the deed stack
$(e, ds, gs, \theta)@_{\texttt{p}}i,$	joins a stack of deeds, ds, to an intention's deed stack such that all deeds in ds are associated with the event, e and the unifier θ. The nth deed in ds is associated with the nth guard in gs
$\texttt{drop}_{\texttt{i}}(n, i)$	removes the top n deeds from the stack. It also performs appropriate garbage collection on events so that $\texttt{events}(i)$ will only list events still relevant to the truncated deed stack. May well also do some general garbage collection of the rest of the intention.
$\texttt{drop}_{\texttt{e}}(n, i)$	removes the top n events from the intention. It also performs appropriate garbage collection on deeds so that $\texttt{deeds}(i)$ will only list deeds still relevant to the truncated event list. May well also do some general garbage collection of the rest of the intention.
$\texttt{new}(e, ds, g, \theta, s)$	creates a new intention with deed stack, ds, associated with the event e, the unifier θ and source, s, such that the nth deed in ds is associated with the nth guard in g.

On top of these can be defined the following.

$\mathtt{hd}_d(i)$	$\mathtt{hd}(\mathtt{deeds}(i))$	returns the head of the deed stack
$\mathtt{hd}_e(i)$	$\mathtt{hd}(\mathtt{events}(i))$	returns the head of the event stack
$\mathtt{hd}_g(i)$	$\mathtt{gu}(1, i)$	returns the head of the guard stack
$\theta^{\mathtt{hd}(i)}$	$\theta(1, i)$	returns the head of the unifier stack
$\mathtt{tl}_i(i)$	$\mathtt{drop_i}(1, i)$	
$(e, d, b, \theta);_{\mathtt{p}}i$	$(e, [d], [g], \theta)@_{\mathtt{p}}i$	join the deed, d, to an intention's deed stack with guard, g and unifier θ
$i\mathrm{U}_\theta\theta$	$(\mathtt{hd}_e(i), \mathtt{hd}_d(i), \mathtt{hd}_g(i), \theta \cup \theta^{\mathtt{hd}(i)});_{\mathtt{p}}\mathtt{tl}_i(i),$	merges the unifier θ with the unifier for the top goal on the deed stack of i.
$i[\theta^{\mathtt{hd}(i)}/\theta]$	$(\mathtt{hd}_e(i), \mathtt{hd}_d(i), \mathtt{hd}_g(i), \theta);_{\mathtt{p}}\mathtt{tl}_i(i)$	replaces the unifier for the top goal on the deed stack of i with θ.
$\mathtt{new}(e, s)$	$\mathtt{new}(e, [\epsilon], [\top], \emptyset, s)$	creates a new intention with an $[\epsilon]$ deed stack, associated with the event e, the guard $\top$, the unifier $\emptyset$ and source, s.
$\mathtt{noplan}(i)$	$\mathtt{deeds}(i) = [\epsilon]$	the top of the intention does not yet have a deed stack.
$\mathtt{empty}_i(i)$	$\mathtt{deeds}(i) = []$	the intention's deed stack is empty (ie. completed)

4.3 Source Syntax

All aspects of an agent's internal state are annotated with sources:

$\mathtt{src}(c)$	returns the source of component c

4.4 Agent State

Definition 4.2 *An* agent state *is a tuple* $< ag, i, I, Pl, B, BR, P, C, In, Out, Cn, Cx, Ann, t >$ *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), 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 t is one of* **A**, **B**, **C**, **D**, **E**, **F**, *indicating the stage from the cycle discussed in §1 currently occupied by the agent. All components of the state are labelled with a source.*

Definition 4.3 *The* initial state *of an agent defined by*

$$bs, gp_1, gp_2, brs, gs, ps, cs$$

is a state

$$< ag_{id}, hd(I), tl(I), \emptyset, B, BR, P, C, \emptyset, \emptyset, Cn, Cx, Ann_i, \mathbf{F} >$$

where

$$\begin{array}{rcl} I & = & map(\lambda g.\ (\mathtt{new}(+!g, \mathtt{self})), gs) \\ B & = & map(\lambda b.\ b\{\mathtt{self}\}, bs) \\ BR & = & map(\lambda br.\ br\{\mathtt{self}\}, brs) \\ P & = & map(\lambda p.\ p\{\mathtt{self}\}, ps) \\ CS & = & map(\lambda c.\ c\{\mathtt{self}\}, cs) \\ Cn & = & map(\lambda gp.\ gp\{\mathtt{self}\}, gp_1) \\ Cx & = & map(\lambda gp.\ gp\{\mathtt{self}\}, gp_2) \end{array}$$

Basically these pair the aspects of the initial state as specified by the programmer with the source `self`

We assume that, given an agent, AG, all parts of its state can be accessed for examination. Moreover we assume that the agent's id, ag, is unique and can be used to reference all other parts of the agent:

For $AG =< ag, i, I, Pl, B, BR, P, C, In, Out, Cn, Cx, Ann, t >$			
ag_{ag}	ag	ag_i	i
ag_I	I	ag_{Pl}	Pl
ag_B	B	ag_{BR}	BR
ag_P	P	ag_C	C
ag_{In}	In	ag_{Out}	Out
ag_{Cn}	Cn	ag_{Cx}	Cx
ag_{Ann}	Ann	ag_t	t
ag_{Is}	$i;I$		

4.5 AIL primitives

We assume additional the following primitive actions

Provided by AIL		
Informal Description	Syntax	Notes
unification	unify(Term, Term)	returns the Most General Unifier of the two terms
standardisation of variables	standardise_apart(Term, Term)	returns a unifier that will rename the variables in the second term so they do not clash with the names in the first term.
apply unifier	$t\theta$	Apply unifier θ to term t

Using these we can define

Informal Description	Syntax	Notes
standardise apart then unify	sunify(Term, Term)	returns the Most General Unifier of the two terms having first standardised apart the first term.

Provided by an Interpreter or Interface		
Informal Description	Syntax	Notes
perception	$\mathbf{perception} = I$	I is a set of intentions
perform action	$\mathbf{do}(a) = \theta$	returns a unifier, *theta* or $\perp$ if action fails
select intention	$\mathcal{S}_i(I) =< i, I' >$	i is an intention in I and I' is a reordering of I.
select plan based on current intention	$\mathcal{S}_p(P, i) = p$	p is a plan in P
initial annotation	Ann_i	
filter unwanted plans	$\mathbf{filter}(P, i) = P'$	Allows the interpreter to filter out plans
relevance of one source to another	$\mathbf{relevant}(s_1, s_2)$	Allows the interpreter to determine when information from one source, s_2, is relevant to s_1.
consistency of beliefs	$\mathbf{consistent}(B)$	This defaults to true but allow an interpreter to check for consistency of beliefs in some specific fashion.

We should probably assume an "update annotation" function for every transition in the semantics which defaults to identity. This would allow additional interpreters to react in specific ways to the selection of particular rules of inference.

5 Belief Checking

$$\frac{b' \in ag_B \quad \text{sunify}(b', b_1\theta_1) = \theta}{< b_1^{ex}; bs_1, \theta_1, ag > \rightarrow_{bc} < bs_1, \theta \cup \theta_1, ag >} \tag{1}$$

Checking for belief rules:

$$\frac{b' \in ag_{BR} \quad \text{sunify}(b', b_1\theta_1) = \theta}{< b_1^{br}; bs_1, \theta_1, ag > \rightarrow_{bc} < bs_1, \theta \cup \theta_1, ag >} \tag{2}$$

Checking for plans:

$$\frac{b' \in ag_P \quad \text{sunify}(b', b_1\theta_1) = \theta}{< b_1^{p}; bs_1, \theta_1, ag > \rightarrow_{bc} < bs_1, \theta \cup \theta_1, ag >} \tag{3}$$

The next rule is for checking "intends" – interpreted as a trigger event which has been committed to (i.e. it was either raised as a sub-goal, or is an original goal (desire) to which planning is being actively applied).

$$\frac{i \in ag_{Is} \quad e \in \texttt{tr}(n, i) \quad \texttt{deeds}(i)[n] \neq \epsilon \vee n < \#\texttt{deeds}(i) \quad \text{sunify}(e, b_1^i\theta_1) = \theta}{< b_1^{i}; bs_1, \theta_1, ag > \rightarrow_{bc} < bs_1, \theta \cup \theta_1, ag >} \tag{4}$$

The next rule is for checking "desires" – interpreted as the original trigger event for some intention.

$$\frac{i \in ag_{Is} \quad \text{sunify}(\texttt{tr}(\#\texttt{deeds}(i), i), b_1^d\theta_1) = \theta}{< b_1^{d}; bs_1, \theta_1, ag > \rightarrow_{bc} < bs_1, \theta \cup \theta_1, ag >} \tag{5}$$

This distinction between intentions and desires allows an interpreter, if it so wishes, to enforce the theoretical distinction that intentions must be consistent.

Checking for constraints:

$$\frac{c \in ag_C \quad \text{sunify}(c, b_1\theta_1) = \theta}{< b_1^{c}; bs_1, \theta_1, ag > \rightarrow_{bc} < bs_1, \theta \cup \theta_1, ag >} \tag{6}$$

Checking the inbox:

$$\frac{in \in ag_{In} \quad \text{sunify}(in, b_1\theta_1) = \theta}{< b_1^{in}; bs_1, \theta_1, ag > \rightarrow_{bc} < bs_1, \theta \cup \theta_1, ag >} \tag{7}$$

Checking the outbox:

$$\frac{out \in ag_{Out} \quad \text{sunify}(out, b_1\theta_1) = \theta}{< b_1^{o}; bs_1, \theta_1, ag > \rightarrow_{bc} < bs_1, \theta \cup \theta_1, ag >} \tag{8}$$

Checking the content:

$$\frac{cn \in ag_{Cn} \quad \text{sunify}(cn, b_1\theta_1) = \theta}{< b_1^{cn}; bs_1, \theta_1, ag > \rightarrow_{bc} < bs_1, \theta \cup \theta_1, ag >} \tag{9}$$

Checking the context:

$$\frac{cx \in ag_{Cx} \quad \text{sunify}(cx, b_1\theta_1) = \theta}{< b_1^{cx}; bs_1, \theta_1, ag > \rightarrow_{bc} < bs_1, \theta \cup \theta_1, ag >} \tag{10}$$

And lastly, actually applying a belief rule.

$$\frac{b \texttt{ :- } bs \in ag_{BR} \quad \text{standardise\_apart}(b_1; bs_1\theta_1, b; bs) = \theta' \quad \text{unify}(b_1\theta_1, b'\theta') = \theta}{< b_1; bs_1, \theta_1, ag > \rightarrow_{bc} < bs(\theta')@bs_1, \theta_1 \cup \theta, ag >} \tag{11}$$

5.1 Sugar

Guards (which get checked) are either beliefs or $\sim belief$ (for "does not believe" – i.e. failure under the closed world assumption) or conjunctions of guards. We provide the following syntax for checking guards in the semantics.

$$ag \models b, \theta \equiv < b, \emptyset, ag_B, ag_{BR} > \rightarrow^*_{bc} < [], \theta, ag_B, ag_{BR} > \quad (12)$$

$$ag \models \sim b, \emptyset \equiv < b, \emptyset, ag_B, ag_{BR} > \not\rightarrow^*_{bc} < [], \theta, ag_B, ag_{BR} > \quad (13)$$

$$ag \models g_1 \wedge g_2, \theta_1 \cup \theta_2 \equiv < g_1, \emptyset, ag_B, ag_{BR} > \rightarrow^*_{bc} < [], \theta_1, ag_B, ag_{BR} > \wedge < g_2\theta_1, \emptyset, ag_B, ag_{BR} > \rightarrow^*_{bc} < [], \theta_2, ag_B, ag_{BR} > \quad (14)$$

$$ag \not\models g \equiv \neg(ag \models g,_) \quad (15)$$

6 Derived Functions

We specify some AIL functions used later in the semantics.

6.1 Applicable Plans

$$appPlans(ag, i) = continue(ag, i) \cup match\_rules(ag, i) \quad (16)$$

Applicable plans produces a set of tuples, these tuples represent an alteration to be made to the intention, i, of the form $<$ trigger, newplan, newguard, length, unifier $>$. To borrow some terminology from 3APL[Hindricks et al., 1999, Dastani et al., 2005], *Plan revision plans* specify that a prefix of the current plan should be dropped and replaced by another while *Goal Planning plans* specify how the current plan should be extended to plan a subgoal. We have unified this idea with the use of events (which allow an agent to pursue multiple intentions at once). So the above tuple specifies a trigger event for the new deed stack section, the new deed stack (that replaces the old prefix), the new guard (that goes with this plan), the length of the prefix to be dropped, and a unifier.

6.2 continue

continue processes intentions with pre-existing deed stacks. ie. where there is no need to start off new sub-goals. However such intentions may also be altered by plan revision plans.

$$continue(ag, i) \quad = \quad \{< \mathtt{hd}_e(i), \mathtt{hd}_d(i), \mathtt{hd}_g(i), 1, \theta \cup \theta^{\mathtt{hd}(i)} > | ag \models \mathtt{hd}_g\theta^{\mathtt{hd}(i)}, \theta \wedge \mathtt{hd}_d(i)\theta\theta^{\mathtt{hd}(i)} \neq \epsilon\} \quad (17)$$

6.3 match_rules

We will talk through match rules a stage at a time.

$$
\begin{aligned}
\mathit{match\_rules}(ag, i) \quad = \quad & \{< p_e, p_d, p_g, \#p_p, \Theta \cup \theta_c > | \qquad (18)\\
& p_e : p_p : p_g : p_d\{p_s\} \in ag_P, \\
& \text{select a plan from the plan library} \\
& \\
& \#p_p > 0 \rightarrow \text{sunify}(\texttt{tr}(\#\texttt{deeds}(i) - \#p_p + 1) : \texttt{prefix}(\#p_p, \texttt{deeds}(i))\theta^{\texttt{hd}(i)}, p_e : p_p) = \theta_e \\
& \text{If this is not a reactive rule } (\#r_i > 0) \\
& \text{unify the trigger of the rule with the trigger of the last deed on the deed stack to be replaced} \\
& \text{This is the top trigger in cases where the prefix is } \epsilon \\
& \text{At the same time unify the rule's plan prefix with the intention's plan prefix} \\
& \\
& ag \models \texttt{hd}(p_g)\theta_e, \theta_b, \; \Theta = \theta^{\texttt{hd}(i)} \cup \theta_e \cup \theta_b \\
& \text{Match the rest of the rule to check the prefix, guards and generate a unifier} \\
& \\
& \mathit{check\_constraints}(ag, p_e, \Theta, \texttt{src}(i)) = \theta_c\}
\end{aligned}
\qquad (19)
$$

This looks complex but it's mostly book-keeping to generate appropriate unifiers.

Note that, in an implementation, Match rules may need to be resource bounded so that it only produces n matches rather than all matches.

6.4 Checking Constraints

Constraint checking follows the inference system: If there is a satisfiable applicable constraint return a success ($\top$) and the relevant plan and unifier.

$$\frac{G = \{gu\theta_c | d' {+}{+} gu \in ag_C \wedge \textbf{relevant}(s, \texttt{src}(c)) \wedge \text{sunify}(d', d\theta) = \theta_c\} \qquad G \neq \emptyset \qquad ag \models \bigwedge G, \theta_p}{\mathit{check\_constraints}(ag, d, \theta, s) = \theta_p} \qquad (20)$$

If there is no suitable constraint also return a success with an empty plan and unifier.

$$\frac{\emptyset = \{gu\theta_c | d' {+}{+} gu \in ag_C \wedge \textbf{relevant}(s, \texttt{src}(c)) \wedge \text{sunify}(d', d\theta) = \theta_c\}}{\mathit{check\_constraints}(ag, d, \theta, s) = \emptyset} \qquad (21)$$

If there is an applicable constraint but it is not satisfied return a failure.

$$\frac{G = \{gu\theta_c | d' {+}{+} gu \in ag_C \wedge \textbf{relevant}(s, \texttt{src}(c)) \wedge \text{sunify}(d', d\theta) = \theta_c\} \qquad G \neq \emptyset \qquad ag \not\models \bigwedge G, \theta_p}{\neg \mathit{check\_constraints}(ag, d, \theta, s)} \qquad (22)$$

7 AIL Operational Semantics

For the time being, in what follows, we omit all parts of the state not effected by a transition for the sake of readability. We also any mention of the interpreter Annotations. It is presumed that any of the interpreter supplied functions (ie. $\mathcal{S}_i$, `relevant` etc. may alter the annotations). It is probably a good idea if we assume an interpreter function for each rule that may update annotations if desired.

Definition 7.1 *A* multi-agent system *is a tuple of n agents and and environment, ξ.*

Standard stuff for updating sets of agents.

$$\frac{\mathcal{A}_i \rightarrow \mathcal{A}'_i}{\{\mathcal{A}_1, ..., \mathcal{A}_i, ..., \mathcal{A}_n, \xi\} \rightarrow \{\mathcal{A}_1, ..., \mathcal{A}'_i, ..., \mathcal{A}_n, \xi\}} \tag{23}$$

7.1 In State A: Intention Selection

Rule (24) is the standard rule for selecting a new intention. Works for all trigger events except the suggestion that a goal be dropped. The last pre-condition is basically a check that the interpreter selection function is behaving as specified and hasn't added or deleted any intentions.

$$\frac{\neg\mathtt{empty}(i) \quad \mathcal{S}_i(I \cup \{i\}) = (i', I') \quad \mathtt{hd}_e(i') \neq -!g \vee \neg\mathtt{noplan}(i') \quad I' = ((I \cup \{i\})/i')}{< ag, i, I, \mathbf{A} > \rightarrow < ag, i', I', \mathbf{B} >} \tag{24}$$

Rule (25) tidies away completed intentions – we stay in state **A** because it's possible the selected intention may be a goal drop.

$$\frac{\mathtt{empty}(i) \quad \mathcal{S}_i(I) = (i', I') \quad I' = (I/i')}{< ag, i, I, \mathbf{A} > \rightarrow < ag, i', I', \mathbf{A} >} \tag{25}$$

Rule (26) is a special rule for those situations where the selected intention is triggered by a drop goal event and has, as yet, no plan. It finds all intentions from the same source which have a matching trigger somewhere in the stack and inserts the goal drop on top. Once again the last two preconditions are checks that the selection function performs as specified.

$$\frac{\begin{array}{c}\mathcal{S}_i(I \cup \{i\}) = (i', I') \quad \mathtt{noplan}(i') \quad \mathtt{hd}_e(i') = -!g \\ I_1 = \{(i_1, \theta) | i_1 \in I' \wedge \exists n.\mathtt{src}(i_1) = \mathtt{src}(i') \wedge \mathrm{sunify}(g, \mathtt{tr}(n, i_1)) = \theta\} \\ I_2 = \{(-!g, \epsilon, \top, \theta);_{\mathtt{p}} i_1, |(i_1, \theta) \in I_1\} \\ \mathcal{S}_i((I'/in_1(I_1)) \cup I_2) = (i'', I'') \quad I' = (I \cup \{i\})/\{i'\} \quad I'' = ((I'/in_1(I_1)) \cup I_2)/\{i''\}\end{array}}{< ag, i, I, \mathbf{A} > \rightarrow < ag, i'', I'', \mathbf{B} >} \tag{26}$$

If none of the rules apply then the interpreter proceeds to step **B** without any other alteration to the internal state. See rule (32) for an example of this sort of rule.

7.2 In State B: Generate applicable plans

Rule (27) uses *appPlans* defined in equation (16) to generate a set of plans deemed applicable by AIL. It then allows the interpreter to filter out some of these.

$$\frac{P' = \mathbf{filter}(appPlans(ag, i)) \quad Pl' \neq \emptyset}{< ag, Pl, \mathbf{B} > \rightarrow < ag, Pl', \mathbf{C} >} \tag{27}$$

Rule (28) applies when there are no applicable plans but the trigger event isn't a request to plan a subgoal. The new plan is essentially an empty plan.

$$\frac{Pl' = \mathbf{filter}(appPlans(ag, i)) \quad Pl' = \emptyset \quad \mathtt{hd}_e(i) \neq +!_t g}{< ag, i, Pl, \mathbf{B} > \rightarrow < ag, i, \{< \mathtt{hd}_e(i), [], [], 0, \emptyset >\}, \mathbf{C} >} \tag{28}$$

Rule (29) applies if there is no applicable plan for some sub-goal. In this case a drop goal event is posted. NB. This does not immediately cause the goal to be dropped. After all some plan might subsequently become applicable. But it gives the agent the option of dropping the goal.

$$\frac{Pl' = \mathbf{filter}(appPlans(ag, i)) \quad Pl' = \emptyset \quad \mathtt{hd}_e(i) = +!_t g}{< ag, i, I, Pl, \mathbf{B} > \rightarrow < ag, i, \mathtt{new}(-!_t g\theta^{\mathtt{hd}(i)}, \mathtt{src}(i)); I, \{< \mathtt{hd}_e(i), [], [], 0, \emptyset >\}, \mathbf{C} >} \tag{29}$$

7.3 In State C: Select a plan

These rules are made complex by the existence of the 3APL plan revision rules. Our applicable plan stage has generated a tuple of a trigger event, e, a deed stack, ds, a guard stack, gs, the length of prefix to be dropped, n, and a unifier, θ. In goal planning the prefix length is just 1 (ie. it will drop the ϵ (no deed yet) marker from the top of the plan and insert the new plan from the rule) but this allows longer prefixes to be dropped in plan revision.

$$\frac{\mathcal{S}_p(Pl, i) = (< e, ds, gs, n, \theta >) \qquad < e, ds, bs, n, \theta > \in Pl \qquad n > 0}{< ag, i, Pl, \mathbf{C} > \rightarrow < ag, (e, ds, gs, \theta) @_{\mathrm{p}} \mathtt{drop}(n, i)[\theta^{\mathtt{hd}(i)}/\theta], [], \mathbf{D} >} \tag{30}$$

Rule (31) handles *reactive plans*. These are plans that a triggered by the state of the agent's beliefs *alone*, not by any particular trigger event or deed stack. We do not want these appended to the current intention, but want them handled as a new intention associated with that belief state coming about.

$$\frac{\mathcal{S}_p(Pl, i) = (< \_, ds, gs, 0, \theta >) \qquad < \_, ds, gs, 0, \theta > \in Pl}{< ag, i, I, Pl, \mathbf{C} > \rightarrow < ag, \mathtt{new}(+\mathtt{state}(\mathtt{hd}(gs)), ds, gs, \theta, \mathtt{self}), i; I, [], \mathbf{D} >} \tag{31}$$

7.4 In State D: Handle top of the Deed Stack

We have a general rule for what to do if the guard on a deed can not be satisfied (nothing).

$$\frac{ag \not\models \mathtt{hd}_g(i)\theta^{\mathtt{hd}(i)}}{< ag, i, \mathbf{D} > \rightarrow < ag, i, \mathbf{E} >} \tag{32}$$

This is an obvious situation where an interpreter might want to change some annotations in order to select a different intention next time round the loop.

7.4.1 Achieve, Perform and Test Goals

Rule (33) handles situations where a goal has already been achieved. When we achieve a goal the top unifier is then transferred to the next goal down in order to preserve instantiations (using the $\mathtt{U}_\theta$ function we defined earlier). We don't check any constraints at this point (since it is already too late!).

$$\frac{ag \models \mathtt{hd}_g(i)\theta^{\mathtt{hd}(i)}, \theta_b \qquad \mathtt{hd}_d(i)\theta^{\mathtt{hd}(i)}\theta_b = +!_t g \qquad t \neq \mathrm{m} \qquad ag \models g, \theta_g}{< ag, i, \mathbf{D} > \rightarrow < ag, \mathtt{tl}_i(i)\mathtt{U}_\theta(\theta^{\mathtt{hd}(i)} \cup \theta_g \cup \theta_b), \mathbf{E} >} \tag{33}$$

Rule (34) sets up a sub goal for planning. It does this by making the sub-goal a new trigger event associated with the "no plan yet" symbol. It leaves $+!_t g$ on the deed stack under ϵ. The idea is that ϵ will be replaced by the deed stack to achieve g and then we test that g has indeed been achieved.

$$\frac{ag \models \mathtt{hd}_g(i)\theta^{\mathtt{hd}(i)}, \theta_b \qquad \mathtt{hd}_d(i)\theta^{\mathtt{hd}(i)}\theta_b = +!_t g \qquad t \neq \mathrm{m} \qquad ag \not\models g}{< ag, i, \mathbf{D} > \rightarrow < ag, (+!_t g, \epsilon, \top, \theta^{\mathtt{hd}(i)});_{\mathrm{p}} i, \mathbf{E} >} \tag{34}$$

7.4.2 Maintaining Goals

Maintain rules have to be treated separately from test, perform and achieve rules. Rule (35) sets up a new plan which will trigger whenever the belief is removed. This plan is a reactive plan $\_ : [] :\sim g : +!g_{\mathrm{a}}$. It fires if g is no longer believed and sets up a new goal to achieve g.

$$\frac{ag \models \mathtt{hd}_g(i)\theta^{\mathtt{hd}(i)}, \theta_b \qquad \mathtt{hd}_d(i)\theta^{\mathtt{hd}(i)}\theta_b = +!_{\mathrm{m}} g}{< ag, i, \mathbf{D} > \rightarrow < ag, (+!_{\mathrm{m}} g, +_{\mathrm{p}}(\_ : [] :\sim g : +!_{\mathrm{a}} g), \mathtt{hd}_g(i), \theta^{\mathtt{hd}(i)} \cup \theta_b;_{\mathrm{p}} i, \mathbf{E} >} \tag{35}$$

This second rule is for dropping maintain goals:

$$\frac{ag \models \mathtt{hd}_g(i)\theta^{\mathtt{hd}(i)}, \theta_b \qquad \mathtt{hd}_d(i)\theta^{\mathtt{hd}(i)}\theta_b = -!_{\mathrm{m}} g}{< ag, i, \mathbf{D} > \rightarrow < ag, (-!_{\mathrm{m}} g, -_{\mathrm{r}}(\_ : [] :\sim g : +!_{\mathrm{a}} g), \mathtt{hd}_g(i), \theta^{\mathtt{hd}(i)} \cup \theta_b);_{\mathrm{p}} i, \mathbf{E} >} \tag{36}$$

It should be noted here that discussion of developing AILite – a stripped down, and therefore more tractable, subset of AIL – include removing reactive plans from the language. In this case, if maintain goals, were preserved these rules would need to be altered so that the new rules had a specific trigger.

7.4.3 Updating Beliefs, Plans, Constraints, Content and Context

We treat updating the belief base, plan library, constraints, etc. the same. We may make several updates at once. An implementation of this would use straightforward list processing and would be simpler than the rather ugly presentation here suggests.

Rule (37) adds beliefs (both internal and external). It also starts new intentions triggered by each "new belief" event. This allows for any AgentSpeak-style belief inference that follows from the change.

$$\frac{\begin{array}{c} ag \models \mathtt{hd}_g(i)\theta^{\mathtt{hd}(i)}, \theta_b \quad \mathtt{hd}_d(i)\theta^{\mathtt{hd}(i)}\theta_b = +BL \quad B' = \{b | b^{ex} \in BL\} \quad \mathbf{consistent} B \cup B' \\ BR' = \{br | br^{br} \in BL\} \quad P' = \{p | p^p \in BL\} \quad C' = \{c | c^c \in BL\} \\ Cn' = \{cn | cn^{cn} \in BL\} \quad Cx' = \{cx | cx^{cx} \in BL\} \quad I' = \{\mathtt{new}(+b, \epsilon, \mathtt{src}(i)) | b \in BL\} \end{array}}{\begin{array}{c} < ag, i, I, B, BR, P, C, Cn, Cx\mathbf{D} > \rightarrow \\ < ag, \mathtt{tl}_i(i)\mathtt{U}_\theta(\theta^{\mathtt{hd}(i)} \cup \theta_b), I'@I, B \cup B', BR \cup BR', P \cup P', C \cup C', Cn \cup Cn', Cx \cup Cx', \mathbf{E} > \end{array}} \quad (37)$$

Rule (38) is for removing beliefs. It is essentially the same as for adding beliefs but is complicated by the need to find a single unifier for all beliefs to be dropped and to ensure that all the beliefs in the list are *actually* dropped (ie. it isn't possible to find a unifier for just some of them and then drop those beliefs).

$$\frac{\begin{array}{c} ag \models \mathtt{hd}_g(i)\theta^{\mathtt{hd}(i)}, \theta_b \quad \mathtt{hd}_d(i)\theta^{\mathtt{hd}(i)}\theta_b = -BL \\ \forall b^{ex} \in BL. \exists b' \in B. b\Theta = b'\Theta \wedge \mathbf{relevant}(\mathtt{src}(b'), \mathtt{src}(b)) \\ \forall br^{br} \in BL. \exists br' \in BR. br\Theta = br'\Theta \wedge \mathbf{relevant}(\mathtt{src}(br'), \mathtt{src}(br)) \\ \forall p^p \in BL. \exists p' \in P. p\Theta = p'\Theta \wedge \mathbf{relevant}(\mathtt{src}(p'), \mathtt{src}(p)) \\ \forall c^c \in BL. \exists c' \in C. c\Theta = c'\Theta \wedge \mathbf{relevant}(\mathtt{src}(c'), \mathtt{src}(c)) \\ B' = \{b' | b^{ex} \in BL \wedge b' \in B \wedge b\Theta = b'\Theta \wedge \mathbf{relevant}(\mathtt{src}(b'), \mathtt{src}(b))\} \\ BR' = \{br' | br^{br} \in BL \wedge br' \in B \wedge br\Theta = br'\Theta \wedge \mathbf{relevant}(\mathtt{src}(br'), \mathtt{src}(br))\} \\ P' = \{p' | p^p \in BL \wedge p' \in P \wedge p\Theta = p'\Theta \wedge \mathbf{relevant}(\mathtt{src}(p'), \mathtt{src}(p))\} \\ C' = \{c' | c^c \in BL \wedge c' \in C \wedge c\Theta = c'\Theta \wedge \mathbf{relevant}(\mathtt{src}(c'), \mathtt{src}(c))\} \\ Cn' = \{cn' | cn^{cn} \in BL \wedge cn' \in Cn \wedge cn\Theta = cn'\Theta\Theta \wedge \mathbf{relevant}(\mathtt{src}(cn'), \mathtt{src}(cn))\} \\ Cx' = \{cx' | cx^{cn} \in BL \wedge cx' \in Cx \wedge cx\Theta = cx'\Theta\Theta \wedge \mathbf{relevant}(\mathtt{src}(cx'), \mathtt{src}(cx))\} \quad I' = \{\mathtt{new}(-b, \epsilon, \mathtt{src}(i)) | b \in BL\} \\ \Theta \text{minimal} \end{array}}{\begin{array}{c} < ag, i, I, B, BR, P, C, Cn, Cx\mathbf{D} > \rightarrow \\ < ag, \mathtt{tl}_i(i)\mathtt{U}_\theta(\theta^{\mathtt{hd}(i)} \cup \theta_b \cup \Theta), I'@I, B/B', BR/BR', P/P', C/C', Cn/Cn', Cx/Cx', \mathbf{E} > \end{array}} \quad (38)$$

7.4.4 Actions

Rule (39) covers generic actions. Another slight abuse of notation here. $a \neq \mathtt{send}$ means $a \neq \mathtt{send}(ag', ilf, \phi.th_{id})$ etc.

$$\frac{\begin{array}{c} ag \models \mathtt{hd}_g(i)\theta^{\mathtt{hd}(i)}, \theta_b \quad \mathtt{hd}_d(i)\theta^{\mathtt{hd}(i)}\theta_b = a, \quad a \neq \mathtt{send} \quad a \neq \mathtt{sjoin} \quad a \neq \mathtt{sadopt} \\ \mathit{check\_constraints}(ag, a, \emptyset, \mathtt{src}(i)) = \theta_c \quad \mathit{do}(a\theta_c) = \theta_a \end{array}}{< ag, i, \mathbf{D} > \rightarrow < ag, \mathtt{tl}_i(i)\mathtt{U}_\theta(\theta^{\mathtt{hd}(i)} \cup \theta_a \cup \theta_b \cup \theta_c), \mathbf{E} >} \quad (39)$$

Sending involves checking and constraints on sending and generating a new message id. It's possible the send refers to an old message id (th_{id} for thread id) if it is a reply.

$$\frac{\begin{array}{c} ag \models \mathtt{hd}_g(i)\theta^{\mathtt{hd}(i)}, \theta_b \quad \mathtt{hd}_d(i)\theta^{\mathtt{hd}(i)}\theta_b = \mathtt{send}(ag', ilf, \phi, th_{id}) \quad \mathcal{M}_{id}(ag, ag') = m_{id} \\ check\_constraints(ag, \mathtt{send}(ag, ag', ilf, \phi, m_{id}, th_{id}), \emptyset) = \theta_m \quad do(\mathtt{send}(ag, ag', ilf, \phi, m_{id}, th_{id})\theta_m) = \theta_a \end{array}}{\begin{array}{c} < ag, i, I, Out, \mathbf{D} > \rightarrow \\ < ag, \mathtt{tl}_i(i)\mathtt{U}_\theta(\theta^{\mathtt{hd}(i)} \cup \theta_a \cup \theta_b \cup \theta_m), \\ \mathtt{new}(+\mathtt{sent}(ag', ilf, \phi, m_{id}, th_{id}), [\epsilon], [\top], \theta^{\mathtt{hd}(i)} \cup \theta_a \cup \theta_b \cup \theta_m, \mathtt{self}); I, Out \cup \{< ag', ilf, \phi, m_{id}, th_{id} >\}, \mathbf{E} > \end{array}} \tag{40}$$

Broadcasting to an agent's entire Content or Context can be implemented on top of send. This will be provided as a built-in function in AIL, but there is no need to treat it separately in the semantics.

Rule (41) allows synchronous joining of groups.

$$\frac{\begin{array}{c} ag \models \mathtt{hd}_g(i)\theta^{\mathtt{hd}(i)}, \theta_b \quad \mathtt{hd}_i(i)\theta^{\mathtt{hd}(i)}\theta_b = \mathtt{sjoin}(ag') \quad check\_constraints(ag, \mathtt{sjoin}(ag'), \emptyset, \mathtt{src}(i)) = \theta_c \\ ag' \models \mathtt{hd}_g(i')\theta^{\mathtt{hd}(i')}, \theta'_b \quad \mathtt{hd}_i(i')\theta^{\mathtt{hd}(i')}\theta'_b = \mathtt{sadopt}(ag) \quad check\_constraints(ag', \mathtt{sadopt}(ag), \emptyset, \mathtt{src}(i')) = \theta'_c \end{array}}{\begin{array}{c} < ag, i, Cx, \mathbf{D} >, < ag', i', Cn', \mathbf{D} > \rightarrow < ag, i\mathtt{U}_\theta(\theta^{\mathtt{hd}(i)} \cup \theta_b \cup \theta_c), Cx \cup \{ag'\theta_c\}, \mathbf{E} >, \\ < ag', i'\mathtt{U}_\theta(\theta^{\mathtt{hd}(i')} \cup \theta'_b \cup \theta'_c), Cn' \cup \{ag\theta'_c\}, \mathbf{E} > \end{array}} \tag{41}$$

(NB. asynchronous joining and leaving of groups can be done via the $+/-_{\mathrm{cn/x}}$ deeds).

Lastly a rule for unsuccessful action attempts:

$$\frac{\begin{array}{c} ag \models \mathtt{hd}_g(i)\theta^{\mathtt{hd}(i)}, \theta_b \quad \mathtt{hd}_d(i)\theta^{\mathtt{hd}(i)}\theta_b = a, \quad \mathtt{hd}_e(i) = +!_t g, \\ check\_constraint(ag, a, \emptyset, \mathtt{src}(i)) = < \top, \theta_c > \quad \neg do(a\theta_c) \end{array}}{< ag, i, I, \mathbf{D} > \rightarrow < ag, i, \mathtt{new}(-!_t g, \theta^{\mathtt{hd}(i)} \cup \theta_b, \mathtt{src}(i)); I, \mathbf{E} >} \tag{42}$$

7.4.5 Dropping Goals

$$\frac{\begin{array}{c} ag \models \mathtt{hd}_g(i)\theta^{\mathtt{hd}(i)}, \theta_b \quad \mathtt{hd}_d(i)\theta^{\mathtt{hd}(i)}\theta_b = -!_t g \quad t \neq \mathrm{m} \\ \mathrm{unify}(+!_t g, \mathtt{events}(i)[n]\theta_e(n, i)) = \theta_e \quad \forall m < n. \mathtt{events}(i)[m]\theta_e(m, i) \neq +!_t g \end{array}}{< ag, i, \mathbf{D} > \rightarrow < ag, \mathtt{drop_e}(n, i), \mathbf{E} >} \tag{43}$$

7.5 In State E: Perception

$$\frac{perception = I', In}{< ag, i, I, [], \mathbf{E} > \rightarrow < ag, i, I'@I, In, \mathbf{F} >} \tag{44}$$

This allows perception to start new intentions *only* and fill the inbox *only*

It may be necessary to add a side-condition (eg. $\forall i \in I'.\ i = (\epsilon, +b, \top, s) \vee = (\epsilon, -b, \top, s)$) – This stops the interpreter performing goal revision (although not belief revision).

7.6 In State F: Message Handling

$$\frac{\begin{array}{c} I' = \{\mathtt{new}(+\mathtt{received}(ag', ilf, \phi, m_{id}, th_{id}), [\epsilon], \top, \theta_m \cup \theta_b, ag') | < ag', ilf, \phi, m_{id}, m'_{id} > \in In \wedge \\ check_constraints(ag, +\mathtt{received}(ag', ilf, \phi, m_{id}, th_{id}), \emptyset = \theta_m\} \end{array}}{< ag, I, In, \mathbf{F} > \rightarrow < ag, I'@I, [], \mathbf{A} >} \tag{45}$$

References

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

[Dastani et al., 2005] Dastani, M., van Riemsdijk, M. B., and Meyer, J.-J. C. (2005). Programming multi-agent systems in 3APL. In Bordini, R. H., Dastani, M., Dix, J., and Seghrouchni, A. E. F., editors, *Multi-Agent Programming: Languages, Platforms and Applications.* Springer.

[Dastani et al., 2006] Dastani, M., van Riemsdijk, M. B., and Meyer, J.-J. C. (2006). Goal types in agent programming. In *Proceedings of the 17th European Conference on Artificial Intelligence*. to appear.

[Fisher, 2005] Fisher, M. (2005). 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.

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

[Pokahr et al., 2005] Pokahr, A., Braubach, L., and Lamersdorf, W. (2005). A flexible bdi architecture supporting extensibility. In Skowron, A., Barthes, J., Jain, L., Sun, R., Morizet-Mahoudeaux, P., and J. Liu, N. Z., editors, *The 2005 IEEE/WIC/ACM International Conference on Intelligent Agent Technology (IAT-2005)*, pages 379–385.

[Rao, 1996] Rao, A. (1996). 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.

[Wooldridge et al., 2002] Wooldridge, M., Fisher, M., Huget, M., and Parsons, S. (2002). Model checking multiagent systems with mable. In *Proceedings of Autonomous Agents and Multi-Agent Systems (AAMAS 02), Bologna, Italy.*