Gwendolen: A BDI Language for Verifiable Agents

Louise A. Dennis [1] [2] and **Berndt Farwer** [3] [4]

Abstract. We describe the Gwendolen BDI (Belief, Desires and Intentions) agent programming language. Gwendolen is implemented in the *Agent Infrastructure Layer* (AIL), a collection of JAVA classes intended for use in model checking agent programs in a variety of languages. The Gwendolen language was developed to test key features of the AIL and its integration with the model checker, JPF, and also to provide a *default semantics* for the AIL classes.

1 Introduction

The AIL (Agent Infrastructure Layer) [Dennis et al., 2008] is a toolkit of JAVA classes designed to support the implementation of BDI (Belief, Desire and Intention) programming languages [Rao and Georgeff, 1995, Wooldridge and Rao, 1999] and the model checking of programs implemented in these languages. Previous approaches to model checking agent programs (e.g., [Bordini et al., 2006]) showed that encoding agent concepts, such as goal and belief, into the state machine of the model checker was a complex and time-consuming task. Similarly it was necessary to adapt the property specification language of a model checker in order to express properties in terms of belief and goals; the natural terminology for reasoning about an agent-based program. Our approach is to encode the relevant concepts from the AIL into the model checker just once and then allow multiple languages to benefit from the encoding by utilising the AIL classes for belief, goal etc., in their implementation. The AIL therefore consists of data structures for representing agents, beliefs, plans, goals and intentions which can be adapted to the operational semantics of individual languages. It has a common agent class intended to function as the base for agent classes in individual languages.

This paper documents the Gwendolen language used during the development of the AIL. It also reports on a simple logic for reasoning about Gwendolen programs using JPF (Java PathFinder) [Visser et al., 2003]. Gwendolen is designed to exhibit many typical features of BDI languages. Programs are presented as a library of plans. Plans are enabled when an agent has certain beliefs and goals and suggest a sequence of deeds to be performed in order to attain a goal. Plans may also be triggered by events, such as changes in belief following perception or the commitment to goals. We term such plans *triggered plans* whereas plans which depend on an agent's internal state alone are *untriggered plans*. Gwendolen agents also distinguish two sorts of goals. *Achievement goals* make statements about beliefs the agent wishes to hold. They remain goals until the agent gains an appropriate belief. *Perform goals* simply state a sequence of deeds to be performed and cease to be a goal as soon as that sequence is complete.

Gwendolen is not intended for use as an actual programming language. The point of the AIL is for many languages to be implemented using it. Of particular interest are the *Jason* [Bordini et al., 2007] implementation of AgentSpeak [Rao, 1996] and 3APL [Dastani et al., 2005], and work is underway to implement these within the AIL. Two existing languages, SAAPL [Winikoff, 2007] and GOAL [de Boer et al., 2007], have already been implemented. Gwendolen does however serve two purposes. Firstly it provides a *default semantics* for the AIL data structures. Any language implemented using the AIL classes will have its own operational semantics and so could, in principle, use the AIL's data structure for perform goals as achievement goals and vice versa. Our intention is that such languages, where possible, should use the rules provided for Gwendolen (the default AIL semantics) only altering these where necessary. Furthermore, where languages change the default semantics it becomes possible to reason about the effects of the changes on model checking. Secondly Gwendolen served a practical purpose as part of the development and debugging of the AIL and has been used in some simple case studies – such as reported in [Dennis and Fisher, 2008].

Gwendolen's agent data structure and the workings of some of the transition rules contain elements which are there purely to assist reasoning about the programs. We will highlight these elements as we encounter them. These suggest adaptations to the semantics of existing agent languages that may be useful to enable reasoning about multi-agent systems.

[1] Dept. of Computer Science, University of Liverpool, Liverpool, UK, L.A.Dennis@liverpool.ac.uk
[2] Work supported by EPSRC grant EP/D052548.
[3] Dept. of Computer Science, Durham University, Durham, UK, berndt.farwer@durham.ac.uk
[4] Work supported by EPSRC grant EP/D054788.

2 Syntax

$$
\begin{aligned}
var &\rightarrow \text{string beginning with an upper-case letter} \\
const &\rightarrow \text{string beginning with a lower-case letter} \\
term &\rightarrow var \\
&\mid const \\
&\mid const(term*) \\
literal &\rightarrow term \\
&\mid \neg term \\
source &\rightarrow const \\
&\mid var \\
belief &\rightarrow literal \\
belief_s &\rightarrow belief\{source\} \\
\tau_g &\rightarrow a \\
&\mid p \\
goalexp &\rightarrow literal \\
action &\rightarrow term \\
&\mid null \\
message &\rightarrow (const, term)^{source}_{const,const}
\end{aligned}
$$

Messages consist of a pair of a constant and a term (an illocutionary force, and the message content), a source representing the sender of the message, and then two constants for a message id and a thread id.

$$
\begin{aligned}
sbelief &\rightarrow belief_s \\
&\mid !_{\tau_g} goalexp \\
&\mid message
\end{aligned}
$$

An *sbelief* (State Belief) is a statement about the agent's internal state. This includes whether the agent has a particular belief, a particular goal, or has sent a particular message.

$$
\begin{aligned}
guard &\rightarrow var \\
&\mid sbelief \\
&\mid \sim sbelief \\
&\mid guard \wedge guard
\end{aligned}
$$

A *guard* is a formulae that must be believed by an agent before a plan is applicable. NB. We are using $\neg$ here to represent "believes not" and $\sim$ to represent "doesn't believe".

$$
\begin{aligned}
deed &\rightarrow action \\
&\mid +belief \\
&\mid -belief \\
&\mid +!_{\tau_g} goalexp \\
&\mid -!_{\tau_g} goalexp \\
&\mid \epsilon
\end{aligned}
$$

Deeds represent the statements that may appear in the bodies of plan. So an agent may execute an action, add or remove beliefs and add or remove goals. ϵ is a distinguished symbol that is taken to mean "no plan yet".

$$
\begin{aligned}
event &\rightarrow var \\
&\mid +belief \\
&\mid -belief \\
&\mid +!_{\tau_g} goalexp \\
&\mid \times!_{\tau_g} goalexp \\
&\mid \texttt{start} \\
event_s &\rightarrow event\{source\}
\end{aligned}
$$

Events are statements that may trigger a plan. So again this includes changes in belief, commitment to goals, and discovering there is a problem with achieving some goal ($\times!_{\tau_g} goalexp$).

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 in a plan. This is for matching plans to intentions. A programmer has no control over the assignment of sources to events, beliefs, and so forth.

$$
\begin{aligned}
plan &\rightarrow (event, deed*) : guard \ \texttt{<-} \ deed* \\
plan_s &\rightarrow plan\{source\}
\end{aligned}
$$

Plans consist of an event and deed stack which are matched against the current intention when determining applicability. There is a guard which must succeed for the plan to be applicable and another deed stack, the body of the plan, representing the course of action to be taken.

The initial state of a Gwendolen agent, as specified by a programmer is:

Name:	*const*	**Environment:**	*const*
Beliefs:	*belief*$*$		
Goals:	*goalexp*$*$		
Plans:	*plan*$*$		

2.1 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 upper-case letters (mostly we presume these are stacks but sometimes we 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 guards (gu).

3 Functions and Data Structures

3.1 Sets and Stacks

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

Stack	
$[]$	an empty stack
$x;s$	stack s, with a new top element x
$\texttt{hd}(s)$	top element of stack s
$\texttt{drop}(n, s)$	remove the top n elements from stack s
$\texttt{prefix}(n, s)$	the top n elements of the stack s
$s[n]$	the n-th element of the stack s
$s_1 \mathbin{@} s_2$	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 sets S_1 and S_2
$S_1 \setminus S_2$	the set S_1 less the set S_2
$in_1(S_1 \times S_2)$	S_1

3.2 Intentions

Definition 1 *An* intention *(i) is an abstract data structure which relates a deed stack to a set of triggering events, 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* triggering event*. Individual deeds are also associated with unifiers for its free variables.*

We do not go into the details of the data structure here but note that it supports the following operations.

Intention	
Δ_i	the deed stack of intention i.
$\mathcal{E}_i$	the event stack of intention i.
$\texttt{tr}(n, i)$	returns the event that triggered the placement of the n-th deed on i
$\theta(n, i)$	returns the unifier associated with the n-th deed in the deed stack
$(e, ds, \theta)@_{\texttt{p}} i,$	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.
$\texttt{drop}_{\texttt{p}}(n, i)$	removes the top n deeds from the stack and also performs appropriate garbage collection on events so that $\mathcal{E}_i$ will only list events still relevant to the truncated deed stack
$\texttt{drop}_{\texttt{e}}(n, i)$	removes the top n events from the intention and also performs appropriate garbage collection on deeds so that Δ_i will only list deeds still relevant to the truncated event list
$[(e, ds, \theta)]\{s\}$	A new intention with deed stack, ds, associated with the event e, the unifier θ and source, s

On top of these can be defined the following.

$\texttt{hd}_{\text{deed}}(i)$	$\texttt{hd}(\Delta_i)$
$\texttt{hd}_{\text{ev}}(i)$	$\texttt{hd}(\mathcal{E}_i)$
$\theta^{\texttt{hd}(i)}$	$\theta(1, i)$
$\texttt{tl}_{\text{int}}(i)$	$\texttt{drop}_{\text{p}}(1, i)$
$(e, d, \theta);_p i$	$(e, [d], \theta)@_{\text{p}} i$
$i \texttt{U}_\theta \theta$	$(\texttt{hd}_{\text{ev}}(i), \texttt{hd}_{\text{deed}}(i), \theta \cup \theta^{\texttt{hd}(i)});_p \texttt{tl}_{\text{int}}(i),$
$i[\theta^{\texttt{hd}(i)}/\theta]$	$(\texttt{hd}_{\text{ev}}(i), \texttt{hd}_{\text{deed}}(i), \theta);_p \texttt{tl}_{\text{int}}(i)$
$[(e)]\{s\}$	$\begin{cases} [(e, [\epsilon], \emptyset)]\{s\} & \text{if } e \text{ is a belief update} \\ [(e, e, \emptyset)]\{s\} & \text{otherwise} \end{cases}$
$\texttt{empty}_{\text{int}}(i)$	$\Delta_i = []$

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

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

3.3 Agent State

Definition 2 *An* agent state *is a tuple* $\langle ag, \xi, i, I, Pl, A, B, P, In, Out, RC\rangle$ *where ag is a unique identifier for the agent,* ξ *an environment, i is the current intention, I denotes all extant intentions, Pl are the currently applicable plans (only used in one phase of the cycle), A are actions executed, B denotes the agent's beliefs, P are the agent's plans, In is the agent's inbox, Out is the agent's outbox, and RC is the current stage in the agent's reasoning cycle. All components of the state are labelled with a source.*

Definition 3 *The* initial state *of an agent defined by*

Name:	ag	**Environment:**	ξ
Beliefs:	bs		
Goals:	gs		
Plans:	ps		

is a state

$$\langle ag, \xi, hd(I), tl(I), \emptyset, \emptyset, B, P, \emptyset, \emptyset, \mathbf{A}\rangle$$

where

$$\begin{aligned} I &= map(\lambda g.\, ([(\texttt{start}, +!_{\tau_g} g, \emptyset)]\{\texttt{self}\}), gs) \\ B &= map(\lambda b.\, b\{\texttt{self}\}bs) \\ P &= map(\lambda p.\, p\{\texttt{self}\}ps) \end{aligned}$$

These pair the aspects of the initial state as specified by the programmer with the source `self`*.*

For notational convenience we will sometimes use the agent identifier, ag, in an agent state $\langle ag, \xi, i, I, appPlans, A, B, P, In, Out, RC\rangle$ to refer to the whole agent and will refer to individual components as ag_B for the belief base B; ag_{Out} for the outbox Out; ag_A for the actions A; and ag_{Is} for the combined set of all intentions $i;I$. The action stack A is one of the data structures included in a Gwendolen agent entirely to assist in reasoning about agents. It represents actions the agent has performed and can be used to verify that, at the least, an agent believes it has done something.

4 Application and Environment

We assume some functions that applications may override

Overridable functions	
Description	Syntax
select intention	$\mathcal{S}_{\text{int}}(I) = \langle i, I' \rangle$
select plan	$\mathcal{S}_{\text{plan}}(P, i) = p$
relevance of sources	$\textbf{relevant}(s_1, s_2)$

$\mathcal{S}_{\text{int}}$ and $\mathcal{S}_{\text{plan}}$ default to selecting the first intention/plan in the list (I or P, respectively). The function **relevant** defaults to true.

We also assume that the environment in which the agents run supports several functions:

Environment functions	
Description	Syntax
perform action	$\textbf{do}(a) = \begin{cases} \theta & \text{if action } a \text{ succeeds} \\ \bot & \text{if action } a \text{ fails} \end{cases}$
get new percepts	$\mathit{newpercepts}(ag)$
percepts to remove	$\mathit{oldpercepts}(ag)$
get messages	$\mathit{getmessages}(ag)$

Where **do** performs an action, *newpercepts* returns any new perceptions since the agent last checked, *oldpercepts* returns anything the agent can no longer perceive and *getmessages* returns any new messages for the agent.

5 State Checking

Gwendolen implements a logical consequence relation $\models$ on guards. This relation relates an agent ag with a pair of a guard and a unifier (if the guard is not ground, the unifier returned by $\models$ is a unifier that makes the statement true). This is the default logical consequence relation provided by the AIL and it is a key component in model checking systems using the AIL. In general the model checking process verifies that an agent satisfies a particular property (such as having a belief, or a goal) if the logical consequence relation succeeds.

The semantics of $\models$ is based on an underlying transition system $\rightarrow_{bc}$. This is more elaborate than required at present but we intend to extend $\models$ with Prolog style reasoning.

$$ag \models \mathit{sbelief}, \theta \equiv \langle \mathit{sbelief}, \emptyset, ag \rangle \rightarrow_{bc}{}^{*} \langle [], \theta, ag \rangle \quad (1)$$

$$ag \models\!\sim gu \equiv \neg(ag \models gu, \_) \quad (2)$$

$$ag \models gu_1 \wedge gu_2, \theta_1 \cup \theta_2 \equiv ag \models gu_1, \theta_1 \wedge ag \models gu_2\theta_1, \theta_2 \quad (3)$$

NB. we are using the notation $t\theta$ to represent the application of a unifier θ to a term t.

Checking Beliefs:

$$\frac{b' \in ag_B \qquad \text{unify}(b', b_1\theta_1) = \theta}{\langle b_1; bs_1, \theta_1, ag \rangle \rightarrow_{bc} \langle bs_1, \theta \cup \theta_1, ag \rangle} \quad (4)$$

Checking for Goals:

$$\frac{i \in ag_{Is} \qquad !_{\tau_g} g' \in \mathcal{E}_i \qquad \text{unify}(g', g\theta_1) = \theta}{\langle !_{\tau_g} g; bs_1, \theta_1, ag \rangle \rightarrow_{bc} \langle bs_1, \theta \cup \theta_1, ag \rangle} \quad (5)$$

Checking the outbox:

$$\frac{m \in ag_{Out} \qquad \text{unify}(m, m_1\theta_1) = \theta}{\langle m_1; bs_1, \theta_1, ag \rangle \rightarrow_{bc} \langle bs_1, \theta \cup \theta_1, ag \rangle} \quad (6)$$

6 Planning

Gwendolen has two options during the planning phase of a reasoning cycle. Either the current intention continues to be processed, or it needs new planning. This is represented by the function *appPlans*, yielding the currently applicable plans:

$$\mathit{appPlans}(ag, i) = \mathit{continue}(ag, i) \cup \mathit{match\_plans}(ag, i) \quad (7)$$

The applicable plans are a set of tuples, each representing an alteration to be made to the intention i, of the form $\langle \text{trigger}, \text{newplan}, \text{guard}, \text{length}, \text{unifier} \rangle$. 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 sub-goal. 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 triggering event for the new deed stack section, the new deed stack (that replaces the old prefix), the guard for the plan (we want to keep a record of this when using untriggered plans), the length of the prefix to be dropped, and a unifier.

6.1 *continue*

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

$$\mathit{continue}(ag, i) = \{\langle \mathtt{hd}_{\text{ev}}(i), \mathtt{hd}_{\text{deed}}(i), \top, 1, \theta^{\mathtt{hd}(i)} \rangle | \mathtt{hd}_{\text{deed}}(i)\theta^{\mathtt{hd}(i)} \neq \epsilon\} \quad (8)$$

6.2 *match_plans*

$$\mathit{match\_plans}(ag, i) = \mathit{match\_plans}_1(ag, i) \cup \mathit{match\_plans}_2(ag, i) \quad (9)$$

The two sets that make up *match_plans* consist of those generated from all the triggered plans in the agent's plan library:

$$\mathit{match\_plans}_1(ag, i) = \{\langle p_e, p_d, p_{gu}, \#p_p, \theta \rangle | (p_e, p_p) : p_{gu} \texttt{ <- } p_d\{p_s\} \in ag_P \wedge \#p_p > 0 \wedge \text{unify}(\mathtt{tr}(\#p_p, i) : \mathtt{prefix}(\#p_p, \Delta_i)\theta^{\mathtt{hd}(i)}, p_e : p_p) = \theta_e \wedge ag \models p_{gu}\theta_e, \theta_b \wedge \theta = \theta^{\mathtt{hd}(i)} \cup \theta_e \cup \theta_b\} \quad (10)$$

and all the untriggered plans in the plan library.

$$\mathit{match\_plans}_1(ag, i) = \{\langle p_e, p_d, p_{gu}, \#\Delta_i + 1, \theta \rangle | (p_e, p_p) : p_{gu} \texttt{ <- } p_d\{p_s\} \in ag_P \wedge \#p_p = 0 \wedge \wedge ag \models p_{gu}, \theta_b \wedge \theta = \theta^{\mathtt{hd}(i)} \cup \theta_b\} \quad (11)$$

Although this looks complex, it is the standard BDI planning mechanism for matching triggers and/or guards. There is a fair amount of book-keeping to keep track of unifiers and the added requirement. Because of the variety of different plan types Gwendolen allows to generate a number for the lines to be removed from the current intention.

7 Gwendolen Operational Semantics

Gwendolen has a reasoning cycle based on 6 stages: **A**, **B**, **C**, **D**, **E**, and **F**. In what follows, we omit all parts of the state not affected by a transition for the sake of readability.

Definition 4 *A* multi-agent system *is a tuple of* n *agents* $\mathcal{A}_i (1 \leq i \leq n)$ *and an environment* ξ.

$$\frac{\mathcal{A}_i \rightarrow \mathcal{A}_i'}{\{\mathcal{A}_1, ..., \mathcal{A}_i, ..., \mathcal{A}_n, \xi\} \rightarrow \{\mathcal{A}_1, ..., \mathcal{A}_i', ..., \mathcal{A}_n, \xi\}} \quad (12)$$

7.1 Stage A: Intention Selection

Rule (13) is the standard rule for selecting a new intention. This covers all intentions except empty intentions:

$$\frac{\neg \mathtt{empty}_{\mathrm{int}}(i) \qquad \mathcal{S}_{\mathrm{int}}(I \cup \{i\}) = (i', I')}{\langle ag, i, I, \mathbf{A}\rangle \rightarrow \langle ag, i', I', \mathbf{B}\rangle} \quad (13)$$

Rule (14) tidies away completed intentions – we stay in stage **A** because it is possible the selected intention is also empty.

$$\frac{\mathtt{empty}_{\mathrm{int}}(i) \qquad \mathcal{S}_{\mathrm{int}}(I) = (i', I')}{\langle ag, i, I, \mathbf{A}\rangle \rightarrow \langle ag, i', I', \mathbf{A}\rangle} \quad (14)$$

7.2 Stage B: Generate applicable plans

Rule (15) uses *appPlans* defined in equation (7) to generate a set of plans deemed applicable by Gwendolen.

$$\frac{appPlans(ag, i) \neq \emptyset}{\langle ag, Pl, \mathbf{B}\rangle \rightarrow \langle ag, Pl', \mathbf{C}\rangle} \quad (15)$$
$$\text{where } Pl' = appPlans(ag, i)$$

Rule (16) applies when there are no applicable plans but the triggering event isn't a goal. The new applicable plan is an empty plan.

$$\frac{appPlans(ag, i) = \emptyset \qquad \mathtt{hd}_{\mathrm{ev}}(i) \neq +!_{\tau_g} g}{\langle ag, i, Pl, \mathbf{B}\rangle \rightarrow \langle ag, i, \{\langle \mathtt{hd}_{\mathrm{ev}}(i), [], 1, \emptyset\rangle\}, \mathbf{C}\rangle} \quad (16)$$

Rule (17) applies if there is no applicable plan for some sub-goal. In this case a problem goal event is posted. NB. This does not immediately cause the goal to be dropped. A response to a problem goal has to be handled by a plan. An obvious default plan is $(\times!_{\tau_g} g, \epsilon) : \top$ `<-` $-!_{\tau_g} g$ which will cause problem goals to be dropped (see (24) and (25)) but this mechanism allows for other responses to problem goals.

$$\frac{appPlans(ag, i) = \emptyset \qquad \mathtt{hd}_{\mathrm{ev}}(i) = +!_{\tau_g} g}{\langle ag, i, I, Pl, \mathbf{B}\rangle \rightarrow \langle ag, i, I, \{\langle \times!_{\tau_g} g, [], 0, \theta^{\mathtt{hd}(i)}\rangle\}, \mathbf{C}\rangle} \quad (17)$$

7.3 Stage C: Select a plan

Plan revision rules may wish to alter the prefix of the deeds on a deed stack. For this reason our applicable plan stage has generated a tuple of a triggering event e, a deed stack, ds, a guard, gu, the length of prefix to be dropped, n, and a unifier, θ. In goal planning the prefix length is just 1 (i.e. it will drop the ϵ (no plan 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.

$$\frac{\mathcal{S}_{\mathrm{plan}}(Pl, i) = (\langle e, ds, gu, n, \theta\rangle) \qquad n > 0}{\langle ag, i, Pl, \mathbf{C}\rangle \rightarrow \langle ag, (e, ds, \theta)@_{\mathtt{p}}\mathtt{drop}(n, i)[\theta^{\mathtt{hd}(i)}/\theta], [], \mathbf{D}\rangle} \quad (18)$$

Rule (19) handles untriggered plans. These are plans that are triggered by the agent's state *alone*, not by any particular triggering event or deed stack. We do not want these appended to the current intention. We treat them as a new intention associated with that agent state coming about.

$$\frac{\mathcal{S}_{\mathrm{plan}}(Pl, i) = (\langle e, ds, gu, 0, \theta\rangle)}{\langle ag, i, I, Pl, \mathbf{C}\rangle \rightarrow \langle ag, [(+\mathtt{st}(gu), ds, \theta)]\{\mathtt{self}\}, i; I, [], \mathbf{D}\rangle} \quad (19)$$

7.4 Stage D: Handle top of the Deed Stack

We start with a rule for handling an empty deed stack. This simply proceeds to the next stage:

$$\frac{\Delta_i = []}{\langle ag, i, \mathbf{D}\rangle \rightarrow \langle ag, i, \mathbf{E}\rangle} \quad (20)$$

7.4.1 Achievement Goals

Rule (21) handles situations where a goal has already been achieved. When we achieve a goal the top unifier is transferred to the next goal down in order to preserve instantiations (using the $\mathtt{U}_\theta$ function we defined earlier).

$$\frac{\mathtt{hd}_{\mathrm{deed}}(i)\theta^{\mathtt{hd}(i)} = +!_a g \qquad ag \models g, \theta_g}{\langle ag, i, \mathbf{D}\rangle \rightarrow \langle ag, \mathtt{tl}_{\mathrm{int}}(i)\mathtt{U}_\theta(\theta^{\mathtt{hd}(i)} \cup \theta_g), \mathbf{E}\rangle} \quad (21)$$

Rule (22) sets up a new achieve sub-goal for planning. It does this by making the sub-goal a new triggering event associated with the "no plan yet" symbol. It leaves $+!_a g$ 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.

$$\frac{\mathtt{hd}_{\mathrm{deed}}(i)\theta^{\mathtt{hd}(i)} = +!_a g \qquad ag \models\!\sim g}{\langle ag, i, \mathbf{D}\rangle \rightarrow \langle ag, (+!_{\tau_g} g, \epsilon, \theta^{\mathtt{hd}(i)});_p i, \mathbf{E}\rangle} \quad (22)$$

7.4.2 Perform Goals

Rule (23) sets up a new perform sub-goal for planning, as for (22).

$$\frac{\mathtt{hd}_{\mathrm{deed}}(i)\theta^{\mathtt{hd}(i)} = +!_p g}{\langle ag, i, \mathbf{D}\rangle \rightarrow \langle ag, (+!_p g, \epsilon, \theta^{\mathtt{hd}(i)});_p (\mathtt{hd}_{\mathrm{ev}}(i), null, \theta^{\mathtt{hd}(i)});_p \mathtt{tl}_{\mathrm{int}}(i), \mathbf{E}\rangle} \quad (23)$$

The *null action* (equivalent to "do nothing"), represented by the placeholder *null*, is required to ensure that we do not lose the record of the event that placed the perform goal on the event stack. This is an example of an operational rule governed by the need to reason about the agent's state. Since we use the event stack to, for instance, reason about the goals of the system we do not want to lose a record of these goals. We remove the perform goal deed from the deed stack, $\mathtt{tl}_{\mathrm{int}}(i)$, but we do not want to lose the the event $\mathtt{hd}_{\mathrm{ev}}(i)$ from the event stack. Since if it only occurs once we lose the record that the agent committed to it (if it was a goal).

7.4.3 Dropping Goals

When dropping a goal we remove all the events on the event stack after we committed to the goal:

$$\frac{\mathtt{hd}_{\mathrm{deed}}(i)\theta^{\mathtt{hd}(i)} = -!_{\tau_g} g \qquad \mathrm{unify}(+!_{\tau_g} g, \mathcal{E}_i[n]\theta_e(n,i)) = \theta_e \qquad \forall m < n.\ \mathcal{E}_i[m]\theta_e(m,i) \neq +!_{\tau_g} g}{\langle ag, i, \mathbf{D}\rangle \rightarrow \langle ag, \mathtt{drop}_{\mathtt{e}}(n,i), \mathbf{E}\rangle} \tag{24}$$

$$\frac{\mathtt{hd}_{\mathrm{deed}}(i)\theta^{\mathtt{hd}(i)} = -!_{\tau_g} g \qquad \neg\mathrm{unify}(+!_{\tau_g} g, \mathcal{E}_i[n]\theta_e(n,i))}{\langle ag, i, \mathbf{D}\rangle \rightarrow \langle ag, \mathtt{tl}_{\mathrm{int}}(i), \mathbf{E}\rangle} \tag{25}$$

7.4.4 Updating Beliefs

Rule (26) adds beliefs. It also starts a new intention triggered by the "new belief" event. This allows for any belief inference that follows from the change.

$$\frac{\mathtt{hd}_{\mathrm{deed}}(i)\theta^{\mathtt{hd}(i)} = +b}{\langle ag, i, I, B, \mathbf{D}\rangle \rightarrow \langle ag, \mathtt{tl}_{\mathrm{int}}(i)\mathtt{U}_\theta\theta^{\mathtt{hd}(i)}, [(+b)]\{\mathtt{src}(i)\});I, B \cup \{b\}, \mathbf{E}\rangle} \tag{26}$$

Rule (27) is for removing beliefs:

$$\frac{\mathtt{hd}_{\mathrm{deed}}(i)\theta^{\mathtt{hd}(i)} = -b \qquad \exists b' \in B.\mathrm{unify}(b', b) = \theta \qquad \mathbf{relevant}(\mathtt{src}(b'), \mathtt{src}(b))}{\langle ag, i, I, B, \mathbf{D}\rangle \rightarrow \langle ag, \mathtt{tl}_{\mathrm{int}}(i)\mathtt{U}_\theta(\theta^{\mathtt{hd}(i)} \cup \theta), [(-b)]\{\mathtt{src}(i)\};I, B \setminus \{b'\}, \mathbf{E}\rangle} \tag{27}$$

7.4.5 Actions

Rule (28) covers generic actions. We use $\uparrow^{ag} m$ for the action of sending a message m to an agent ag. For presentational reasons $a \neq \uparrow$ means $a \neq \uparrow^{ag} (ilf, m)^{ag'}_{m_{\mathrm{id}}, th_{\mathrm{id}}}$

$$\frac{\mathtt{hd}_{\mathrm{deed}}(i)\theta^{\mathtt{hd}(i)} = a \quad a \neq \uparrow \quad a \neq null \quad \mathbf{do}(a) = \theta_a}{\langle ag, i, A, \mathbf{D}\rangle \rightarrow \langle ag, \mathtt{tl}_{\mathrm{int}}(i)\mathtt{U}_\theta(\theta^{\mathtt{hd}(i)} \cup \theta_a), a;A, \mathbf{E}\rangle} \tag{28}$$

Note how we place the action a on the action stack A to record its execution for reasoning purposes.

Sending involves generating a new message id. It is possible the send refers to an old message id (th_{id} for thread id) if it is a reply.

$$\frac{\mathtt{hd}_{\mathrm{deed}}(i)\theta^{\mathtt{hd}(i)} = \uparrow^{ag} (ilf, \phi)^{ag'}_{,th_{\mathrm{id}}{}'} \qquad m_{\mathrm{id}} = \mathcal{M}_{\mathrm{id}}(ag, ag') \qquad (th_{\mathrm{id}}{}' = null \wedge th_{\mathrm{id}} = \mathcal{TH}_{\mathrm{id}}(ag, ag')) \vee th_{\mathrm{id}} = th_{\mathrm{id}}{}' \qquad \xi.\mathbf{do}(\uparrow^{ag'} (ilf, \phi)^{ag}_{m_{\mathrm{id}}, th_{\mathrm{id}}}) = \theta_a}{\begin{array}{c}\langle ag, \xi, i, I, A, Out, \mathbf{D}\rangle \rightarrow \\ \langle ag, \xi, \mathtt{tl}_{\mathrm{int}}(i)\mathtt{U}_\theta(\theta^{\mathtt{hd}(i)} \cup \theta_a), \\ [(\uparrow^{ag'} (ilf, \phi)^{ag}_{m_{\mathrm{id}}, th_{\mathrm{id}}}, \epsilon, \theta^{\mathtt{hd}(i)} \cup \theta_a)]\{\mathtt{self}\};I, \\ \uparrow^{ag'} (ilf, \phi)^{ag}_{m_{\mathrm{id}}, th_{\mathrm{id}}};A, Out \cup \{\uparrow^{ag'} (ilf, \phi)^{ag}_{m_{\mathrm{id}}, th_{\mathrm{id}}}\}, \mathbf{E}\rangle\end{array}} \tag{29}$$

where $\mathcal{M}_{\mathrm{id}}$ and $\mathcal{TH}_{\mathrm{id}}$ are functions that generate fresh id numbers for messages and threads respectively.

There is a rule for null actions:

$$\frac{\mathtt{hd}_{\mathrm{deed}}(i)\theta^{\mathtt{hd}(i)} = null,}{\langle ag, i, \mathbf{D}\rangle \rightarrow \langle ag, \mathtt{tl}_{\mathrm{int}}(i)\mathtt{U}_\theta\theta^{\mathtt{hd}(i)}, \mathbf{E}\rangle} \tag{30}$$

Lastly we have two rules for unsuccessful actions:

$$\frac{\mathtt{hd}_{\mathrm{deed}}(i)\theta^{\mathtt{hd}(i)} = a, \qquad \mathtt{hd}_{\mathrm{ev}}(i) = +!_{\tau_g} g \qquad \neg\xi.\mathbf{do}(a)}{\langle ag, \xi, i, I, \mathbf{D}\rangle \rightarrow \langle ag, \xi, (\times!_{\tau_g} g, \epsilon, \theta^{\mathtt{hd}(i)});_p i, I, \mathbf{E}\rangle} \tag{31}$$

$$\frac{\mathtt{hd}_{\mathrm{deed}}(i)\theta^{\mathtt{hd}(i)} = a, \qquad \mathtt{hd}_{\mathrm{ev}}(i) \neq +!_{\tau_g} g \qquad \neg\xi.\mathbf{do}(a)}{\langle ag, \xi, i, I, \mathbf{D}\rangle \rightarrow \langle ag, \xi, i, I, \mathbf{E}\rangle} \tag{32}$$

7.5 Stage E: Perception

$$\overline{\langle ag, \xi, i, I, [], \mathbf{E}\rangle \rightarrow \langle ag, \xi, i, I' \mathbin{@} I'' \mathbin{@} I, In, \mathbf{F}\rangle} \tag{33}$$

where $In = \xi.getmessages(ag)$,

$I' = \{[((+b, +b, \emptyset))]\{\mathtt{percept}\} \mid b \in \xi.newpercepts(ag)\}$,

and $I'' = \{[((-b, -b, \emptyset)]\{\mathtt{percept}\} \mid b \in \xi.oldpercepts(ag)\}$

Perception does not directly act on the belief base, instead it sets up intentions to alter the belief base – this allows for planning based on, for instance, the reliability of the information.

7.6 Stage F: Message Handling

$$\overline{\langle ag, i, In, \mathbf{F}\rangle \rightarrow \langle ag, I' \mathbin{@} I, [], \mathbf{A}\rangle} \tag{34}$$

where $I' = \{[(+ \downarrow^{ag} (ilf, \phi)^{ag'}_{m_{\mathrm{id}}, th_{\mathrm{id}}})]\{ag'\} | (ilf, \phi)^{ag'}_{m_{\mathrm{id}}, th_{\mathrm{id}}} \in In\}$

We use $\downarrow^{ag} m$ to represent the fact that agent, ag has received message, m. Again the receipt of messages doesn't directly act on the belief base, but causes new intentions to be set up noting the receipt of the message.

8 Communication Semantics

Gwendolen has no fixed semantics for communication instead both receiving and sending messages are treated as belief update events and start new intentions. This allows a semantics of communication for any particular program to be established using plans. For instance a semantics for **perform** messages can be established with the following plan:

$$(+ \downarrow^{A_1} (\mathbf{perform}, Goal)^{A_2}_{M_{\mathrm{id}}, Th_{\mathrm{id}}}, \epsilon) : \top \ \texttt{<-}\ +!_p Goal$$

This specifies that the content of a message with a **perform** illocutionary force should be established as a perform goal.

We also intend to extend the AIL and Gwendolen with grouping mechanisms such as described in [Dennis et al., 2007] with the intention that these can be used to program a range of multi-agent coordination models such as organisations and roles [Ferber et al., 2003], joint intentions [Cohen and Levesque, 1991] and institutions [Esteva et al., 2001].

9 Model Checking

The AIL includes a set of interfaces which allow a dedicated set of classes, referred to as the *MCAPL (Model Checking Agent Programming Languages) interface* to model check programs. The MCAPL interface allows a user to specify simple properties in a temporal and modal logic and then check that these properties hold using the JPF (Java PathFinder) model checker. At present we can verify properties of simple programs written in Gwendolen using properties expressed in the following *MCAPL property specification language*:

$$\begin{array}{ll} a ::= & \text{constant} \\ f ::= & \text{ground first order formula} \\ \phi ::= & \mathbf{B}(ag, f) \mid \mathbf{G}(ag, f) \mid \mathbf{A}(ag, f) \mid \phi \wedge \phi \mid \phi \vee \phi \mid \neg\phi \\ & \mid \phi\, \mathbf{U}\, \phi \mid \phi\, \mathbf{R}\, \phi \end{array} \tag{35}$$

Some of these formulas have a semantics determined by the implementation of the MCAPL interface. The AIL implements the formulas $\mathbf{B}(ag, f)$ and $\mathbf{G}(ag, f)$ as *sbelief*s. Consider a program P for a multi-agent system and let *MAS* be the state of the multi-agent system at one point in the run of the program. Consider an agent, $ag \in MAS$, at this point in the program execution, $MAS \models_{MC} \mathbf{B}(ag, f)$ iff $ag \models f, \theta$ (for some θ). Similarly AIL interprets $\mathbf{G}(ag, f)$ to mean that $MAS \models_{MC} \mathbf{G}(ag, f)$ iff $ag \models !_a f, \theta$ (for some θ). We use the syntax f : *belief* and f : *action* here to show how the logical consequence relation distinguishes between the formulae based on their type. By contrast reasoning about executed actions inspects the action stack so $MAS \models_{MC} \mathbf{A}(ag, f)$ iff $f \in ag_A$.

The other formulas in the MCAPL property specification language have an LTL (Linear Temporal Logic)-based semantics defined by the MCAPL interface and including the operators until ($\mathbf{U}$) and release ($\mathbf{R}$).

$$MAS \models_{MC} \phi_1 \wedge \phi_2 \text{ iff } MAS \models_{MC} \phi_1 \text{ and } MAS \models_{MC} \phi_2$$
$$MAS \models_{MC} \phi_1 \vee \phi_2 \text{ iff } MAS \models_{MC} \phi_1 \text{ or } MAS \models_{MC} \phi_2$$
$$MAS \models_{MC} \neg\phi \text{ iff } MAS \not\models_{MC} \phi.$$

The temporal formulas hold true of runs of the programs in the JPF model checker. A run consists of a sequence of program states $MAS_i, 0 \leq i \leq n$ where MAS_0 is the initial state of the program and MAS_n is the final state in the run. Let P be a multi-agent program $P \models_{MC} \phi_1 U \phi_2$ iff in all runs of the program there exists a program state MAS_j such that $MAS_i \models_{MC} \phi_1$ for all $0 \leq i < j$ and $MAS_j \models_{MC} \phi_2$. Similarly $P \models_{MC} \phi_1\, \mathbf{R}\, \phi_2$ iff either $MAS_i \models_{MC} \phi_1$ for all, i or exists MAS_j such that $MAS_i \models_{MC} \phi_1$ for all, $0 \leq i \leq j$ and $MAS_j \models_{MC} \phi_1 \wedge \phi_2$. The more commonly recognised temporal operators $\Diamond$ (eventually) and $\Box$ (always) are special cases of $\mathbf{U}$ and $\mathbf{R}$.

10 Current Status

We have written a number of simple programs based on "Blocks World"-type examples where agents form goals to pick up blocks, or to get other agents to pick up blocks. Two such programs are shown below:

10.1 Program 1

This is a single agent program. The agent believes a block to be available and that its hands are empty. The has a single achievement goal – to pick something up. It has a single plan, triggered by the desire to pick something up and guarded by something being available to pick up. It then adds a belief that it has picked up the object and removes the belief that its hands are empty.

$$\begin{array}{rl} \textbf{Name} & : ag1 \\ \textbf{Beliefs} & : empty \\ & \quad available(block) \\ \textbf{Goals} & : !_a pickup(X) \\ \textbf{Plans} & : (+!_a pickup(Y), \epsilon) : empty \wedge available(Y) \\ & \qquad\qquad \texttt{<-} \;\; +pickup(Y); \\ & \qquad\qquad\qquad -empty \end{array}$$

We can verify properties of this such as $\Diamond(\mathbf{B}(ag1, pickup(block)))$. It is also possible to establish the falsity of expressions such as $\Box(\neg(\mathbf{B}(ag1, pickup(block)) \wedge \mathbf{B}(ag1, empty)))$. This property is violated because the agent first adds the belief it has picked up the block *before* it removes the belief that its hands are empty.

10.2 Program 2

This program consists of two agents, a master and a slave. The master has a goal, to pick something up. To achieve this it sends a message to the slave with the illocutionary force, **achieve**, telling the slave to pick something up.

The slave has two plans. The first of these performs a pick up action if it wants to pick something up, while the second establishes the semantics for **achieve**, namely that the slave establishes an achievement goal when asked to achieve something.

$$\begin{array}{rl} \textbf{Name} & : master \\ \textbf{Beliefs} & : check \\ \textbf{Goals} & : !_a pickup \\ \textbf{Plans} & : (+!_a pickup, \epsilon) : \top \\ & \qquad \texttt{<-} \;\; \uparrow^{slave} (\textbf{achieve}, pickup)^{master}_{M_{\mathrm{id}}, Th_{\mathrm{id}}}; \\ & \qquad\qquad\qquad wait; \\ & \qquad\qquad\qquad +!_a check \\ \textbf{Name} & : slave \\ \textbf{Plans} & : (+!_a pickup, \epsilon) : \top \;\; \texttt{<-} \;\; pickup \\ & \qquad (+ \downarrow^{slave} (\textbf{achieve}, Goal)^{master}_{M_{\mathrm{id}}, Th_{\mathrm{id}}}, \epsilon) : \top \;\; \texttt{<-} \;\; +!_a Goal \end{array}$$

We can verify some properties of this program, such as $\Diamond(\mathbf{B}(master, check))$. But, because JPF does not assume

a fair scheduler, it is impossible to verify, for instance, $\Diamond(\mathbf{B}(master, pickup))$ because there is a run where the slave agent never gets an opportunity to do anything. We are currently working to overcome these limitations and to customise JPF to both improve its efficiency in an agent setting and provide a flexible set of appropriate configurations for checking agent programs under a range of assumptions.

11 Conclusions

We have presented here a prototype language, Gwendolen, used in developing the AIL, which is a collection of classes intended to assist in model checking agent programs written in a variety of languages. We have discussed how the MCAPL interface allows us to inspect a Gwendolen agent in order to deduce properties of that agent for use in model checking.

Gwendolen provides a default semantics for the AIL classes. This allows an implementation of an existing language using the AIL to (potentially) prove that it has preserved the semantics of the AIL data structures sufficiently to generate sound results from the model checking process.

The construction of the Gwendolen language has also revealed places where language semantics need to be specialised if we intend to reason about agents written in that language.

12 Further Work

In future we intend to add basic constructs for forming groups to Gwendolen based on a framework outlined in [Dennis et al., 2007] and at the same time to extend the MCAPL property specification language with appropriate primitives for discussing groups of agents.

We also hope to customise JPF for model-checking with agent languages. For instance, if we assume that the program states of interest in the model-checking process are those that occur after every rule application (for instance) rather than every Java state executed then we can increase the complexity of programs that can be verified within a reasonable time.

Lastly we intend to implement a number of existing BDI languages in the AIL (two implementations, SAAPL [Winikoff, 2007] and GOAL [de Boer et al., 2007] are already complete with others, in particular AgentSpeak [Bordini et al., 2007] underway) and investigate correctness issues.

REFERENCES

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

[Bordini et al., 2007] Bordini, R. H., Hübner, J. F., and Wooldridge, M. (2007). *Programming Multi-agent Systems in AgentSpeak Using Jason*. Wiley.

[Cohen and Levesque, 1991] Cohen, P. R. and Levesque, H. J. (1991). Teamwork. Technical Report 504, SRI International, California, USA.

[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*, pages 39–67. Springer.

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

[Dennis et al., 2008] Dennis, L. A., Farwer, B., Bordini, R. H., and Fisher, M. (2008). A flexible framework for verifying agent programs (short paper). In Padgham, Parkes, Muller, and Parsons, editors, *Proc. of the 7th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2008)*. ACM. To Appear.

[Dennis and Fisher, 2008] Dennis, L. A. and Fisher, M. (2008). Programming Verifiable Heterogeneous Agent Systems. In *Proc. Eighth International Workshop on Programming Multiagent Systems (ProMAS)*. To Appear.

[Dennis et al., 2007] Dennis, L. A., Fisher, M., and Hepple, A. (2007). Foundations of flexible multi-agent programming. In *Eighth Workshop on Computational Logic in Multi-Agent Systems (CLIMA-VIII)*.

[Esteva et al., 2001] Esteva, M., Rodríguez-Aguilar, J. A., Sierra, C., Garcia, P., and Arcos, J. L. (2001). *Agent Mediated Electronic Commerce. The European AgentLink Perspective*, chapter On the Formal Specificaion of Electronic Institutions, pages 126–147. Number 1991 in LNAI. Springer.

[Ferber et al., 2003] Ferber, J., Gutknecht, O., and Michel, F. (2003). 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.

[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.

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

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

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

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

[Wooldridge and Rao, 1999] Wooldridge, M. and Rao, A., editors (1999). *Foundations of Rational Agency*. Applied Logic Series. Kluwer Academic Publishers.