Programming Verifiable Heterogeneous Agent Systems*

Louise A. Dennis and Michael Fisher

Department of Computer Science, University of Liverpool, Liverpool, UK
Contact: L.A.Dennis@liverpool.ac.uk

Abstract. Our overall aim is to provide a verification framework for practical multi-agent systems. To achieve practicality, we must be able to describe and implement *heterogeneous* multi-agent systems. To achieve verifiability, we must define semantics appropriately for use in formal verification. Thus, in this paper, we tackle the problem of implementing heterogeneous multi-agent systems in a semantically clear, and appropriate, way.

1 Introduction

The construction of multi-agent systems has become relatively straightforward as more high-level agent programming frameworks have become available [7, 27, 5, 1]. Quite sophisticated systems have been developed and, in some cases, deployed. However, there still remain a number of problems, particularly regarding the *flexibility* and *reliability* of multi-agent systems. The particular aspect of flexibility we are interested in here concerns systems of *heterogeneous* agents. Thus, while we might construct a multi-agent system in one language, a more realistic scenario is that a practical multi-agent system will comprise agents implemented in a number of languages. Several approaches already exist which support heterogeneous agents, e.g:

- communication at a common level can be achieved if the agents are based on extensions of standard languages such as Java or Java Agent Services [18], or comprise standard distribution mechanisms such as CORBA [19] or .COM [26];
- if all agents are implemented within a common underlying framework [1, 16], then this provides a layer through which communication and ontologies can be handled;
- if the heterogeneous agents are embedded in an appropriate *wrapper* that handles communication and coordination, such as that utilised within the IMPACT framework [24, 12], then effective heterogeneous multi-agent systems can be built; and
- if all agents subscribe to a general interaction protocol, for example the FIPA [14] speech act approach, communicative aspects are taken care of within the protocol.

We are interested in developing systems whose *reliability* and *trustworthiness* can be (automatically) assessed. Specifically, we aim to apply *formal verification* to agents [3], by developing *model checking* [6, 17] techniques. For some of the above, formal analysis has been carried out; for others, no definitive formal semantics is available. However, those systems where formal analysis has been considered, notably IMPACT [24] and FIPA-ACL [20, 21], have only considered interaction aspects of agents, not their full

* Work supported by EPSRC (UK) grant EP/D052548.

internal behaviour. This can cause problems if the mental states of of the agents are inconsistent, even if the communication aspects are homogenised [25].

In our work, we have aimed to develop techniques for analysing implemented multi-agent systems. Beginning with model-checking techniques for AgentSpeak [4], we are extending these now to other languages such as 3APL [7]. Until now we have only considered systems of agents implemented using the same language, i.e. homogeneous multi-agent systems. Our overall aim within this paper is to describe our approach for handling *heterogeneous multi-agent systems*. Within this, we also aim to show:

1. how our Agent Infrastructure Layer (AIL) [11] provides an effective, high-level, basis for implementing operational semantics for BDI-like programming languages;
2. how the AIL supports heterogeneous agent computation, and to provide an example of this — the example is a simplified *Contract Net* [23] and the system comprises three agents, each implemented in a different language[1]; and
3. how formal verification can *potentially* be carried out [2].

2 The AIL

The AIL [11, 9] is a toolkit of Java classes designed to support the implementation of BDI programming languages and the model checking of programs implemented in these languages. Our previous approaches to model checking agent programs showed that encoding agent concepts, such as goals and beliefs, into the state machine of the model checker was a complex and time-consuming task. It was also necessary to adapt the property language of a model checker to express properties in these terms; 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 in their implementation. The AIL therefore consists of data structures for representing agents, beliefs, plans, etc., which can be adapted to the operational semantics of individual languages. A language implemented in the AIL sub-classes the AIL's agent class and then specifies a *reasoning cycle*. The reasoning cycle consists of a transition system which defines a number of stages and specifies the changes to the agent structure that occur as it passes from one stage to the next. The AIL agent data structure contains a place holder for the current stage of the reasoning cycle which is instantiated in interpreted agents. To aid the implementation of interpreters the AIL also contains an *operational semantics package* (`OSRules`) which contains many sample transition rules.

In [11], the AIL is described as a customisable language with its own operational semantics and reasoning cycle. This proved too inflexible to accommodate the language features found in implementations of the major agent programming languages. In particular, the reasoning cycle became an obstacle. Hence our current implementation of the AIL is a *collection* of data structures. The AIL's most complex data structure is that which represents an *intention*. BDI languages use intentions to store the *intended means* for achieving goals – this is generally represented as some form of *deed stack* (a deed

[1] GOAL [8], SAAPL [28] and Gwendolen, a BDI language developed by the first author.
[2] The full model-checker is under development with our partners in Durham [9].

may be an action, a belief update or the commitment to a goal). Intention structures in BDI languages may also maintain information about the (sub-)goal they are intended to achieve or the event that triggered them. In the AIL, we aggregate this information. Most importantly for the discussion here an intention becomes a stack of pairs of an event and a deed, Individual pairs associate a deed with the event that has caused the deed to be placed on the intention. New events are associated with an empty deed, ϵ.

The AIL's basic plan data structure associates some matching information, a guard and a deed stack. The AIL includes operations for using plans to modify intentions. In these the plan is "matched" to the intention; the plan's guard is checked; and the plan's body (a deed stack) is added to the deed stack of the intention. Since some BDI languages trigger plans by matching the prefix of the deed stack and some by matching a *trigger event*, the AIL plan matching information contains both a trigger event (which must match the top event of the intention) and a deed stack prefix (which must match the prefix of the intention's deed stack). The matched prefix is replaced by the plan's body. By way of example, consider the following AIL plan for cleaning rooms.

trigger	**prefix**	**guard**	**body**
$+!_a$`clean()`	ϵ	`dirty(Room)`	$+!_a$`Goto(Room)` $+!_a$`Vacuum(Room)`

We use the syntax $+!_a g$ to indicate the commitment to an *achievement goal*. The AIL allows several different goal types. In this paper we will be interested in *achievement* (or declarative) goals represent a belief the agent desires to have; and *perform goals* (syntax $+!_p g$), which need not lead to a belief. The following shows the operation of the AIL's default planning operation on an intention, given the plan above.

trigger	deed
$+!_a$ `clean()`	ϵ

$\rightarrow$

trigger	deed
$+!_a$`clean()`	$+!_a$`Goto(Room)`
$+!_a$`clean()`	$+!_a$`Vacuum(Room)`

The plan's trigger matched the top event of this intention and its prefix matched the deed stack prefix. We assume the guard was believed. The top row of the intention (matching the prefix) was removed and replaced it with two rows representing the body of the plan. The AIL can also handle *reactive plans* which become applicable whenever the guard is satisfied do not match a specific intention. We have omitted discussion of unification. This is, in general, intuitive but fiddly, and handling unifiers obscures the presentation.

The AIL provides an environment interface which it expects systems implemented using it to satisfy. An environment, ξ, is expected to implement the following: $\mathbf{do}(a)$ executes the action, a. It is assumed that this handles any external effects of an agent's actions. $newpercepts(ag)$ returns any new perceptions from the environment since the agent (ag) last checked and $oldpercepts(ag)$ returns a list of things that can no longer be perceived. $getmessages(ag)$ returns a list of messages. When one of these interface functions is called we write it, for instance, as $\xi.\mathbf{do}(a)$.

3 The AIL as an Environment for Heterogeneous Agents

The AIL view of the world is shown in Figure 1. The AIL is implemented in Java and, in turn, a variety of languages are implemented in the AIL. The AIL comes with

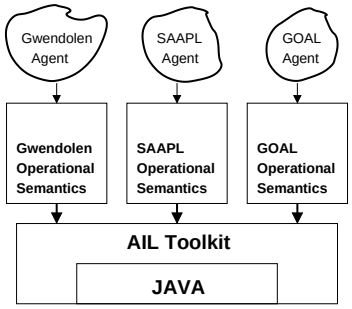

Fig. 1. Architecture of Heterogeneous Multi-Agent System.

interfaces and default classes for composing multi-agent systems, these classes handle agents at the level of the underlying AIL data structures and so can be used with agents in any language that builds upon those structures. This makes it straightforward to build a heterogeneous multi-agent system using the AIL once the implementations of the target semantics are in place. In the subsequent sections, we will describe the three agent languages we use, namely: Gwendolen (Section 4); SAAPL (Section 5); and GOAL (Section 6). Our aim was to use not only the AIL data structures but as many of the prewritten AIL transition rules as possible in order to assess their breadth and utility. We also provide code fragments, both to give the reader some concrete syntax, and to introduce our heterogeneous scenario. Thus, we will code agents in each of these languages to take part in a *Contract Net* scenario [23], a well-known, and widely used, model of coordination in distributed problem-solvers. Essentially, a particular agent (the *manager*) broadcasts tasks (goals) to be achieved, and then agents capable of achieving these tasks bid for the contract. In real scenarios, the bidding, allocation and sub-contracting is complex. However, we consider a *very* simple version: the manager does not broadcast to all the agents in the system at once but instead contacts them in turn; there is no bidding process nor sub-contracting; agents volunteer for a task if, and only if, they can perform it; and the manager simply accepts the first proposal it receives.

4 Gwendolen

Gwendolen [10] is based on the language presented in [11] now abstracted away from the AIL. Gwendolen uses the AIL's planning mechanisms "off the shelf" with both achievement and perform goals, event triggered, reactive and prefix matching plans. In this paper we only look at plans which match the prefix ϵ and so omit that from our plan presentation. We therefore represent Gwendolen's version of the plan

trigger	$+!_a$`clean()`
prefix	[ϵ]
guard	`dirty(Room)`
body	$+!_a$`Goto(Room)` $+!_a$`Vacuum(Room)`

as $+!_a clean() : dirty(Room)$ <–
$+!_a Goto(Room); +!_a Vacuum(Room)$

Throughout this paper, we will use ';' to indicate concatenation of deeds on a stack and $\uparrow^a\ m$ to indicate the sending of a message m to agent a and $\downarrow^a\ m$ to indicate the receipt of a message m from an agent a (as in [28]). Gwendolen has two negation symbols which can be used in plan guards. $\neg gu$ succeeds if the agent believes $\neg gu$ (stong negation), $\sim gu$ success if the agent does not believe gu (weak negation).

Contract Net Code. The code for our Contract Net written in Gwendolen is as follows:

$$+!_a g : cando(g)\ \ <-\ \ a \qquad +!_a g' : cando(g')\ \ <-\ \ a'$$
$$+!_a g : \neg cando(g)\ \ <-\ \ +!_p cfp(g) \qquad +!_a g' : \neg cando(g')\ \ <-\ \ +!_p cfp(g')$$

These are four basic plans for achieving the goals g and g' either by performing an action or committing to performing a "call for proposals".

Our Contract Net protocol assumes a message semantics consisting of a performative and a ground formula. The **perform** performative expects the agent to perform an action and the **tell** performative expects the receiving agent to update its belief base. So, Gwendolen has a plan for asking an agent to respond to a request to perform a goal, together with a number of plans for how to perform a 'respond' action and how to act if an agent has a proposal or is awarded a contract. $ag(A)$ is the belief that A is the name of another agent and $name(N)$ is the belief that N is the agent's own name.

$$+!_p cfp(T) : ag(A) \wedge name(N) \wedge \sim \uparrow^A (\textbf{perform}, respond(T, N))\ \ <- \\ \uparrow^A (\textbf{perform}, respond(T, N)); \texttt{wait}$$

$$+!_p cfp(T) : proposal(T, A)\ \ <-\ \ \texttt{wait}$$
$$+!_p respond(T, A) : cando(T) \wedge name(N)\ \ <-\ \ \uparrow^A (\textbf{tell}, proposal(T, N))$$
$$+!_p respond(T, A) : \neg cando(T) \wedge name(N)\ \ <-\ \ \uparrow^A (\textbf{tell}, sorry(T, N))$$
$$+proposal(P, A) : \top\ \ <-\ \ \uparrow^A (\textbf{tell}, award(P))$$
$$+award(T) : \top\ \ <-\ \ +!_a T$$

5 SAAPL

SAAPL (Simple Abstract Agent Programming Language) [28] is intended as a simple abstraction of languages such as Jason [5], 3APL [7], and CAN [29]. In [28] SAAPL is used to drive the discussion of *commitment machines*. We ignore this issue and focus instead on SAAPL's semantics as a simple, yet typical, language. The semantics of SAAPL, as presented in [28], are shown in Figure 2 where Q is the environment (a message queue), N is name of the agent, B the belief base, Γ the intentions, Π the plan base, and Δ the applicable plans.

SAAPL *implemented with AIL.* SAAPL's semantics has three stages: Basic (which acts on a single intention); Agent which acts on a set of intentions; and MAS which acts on a set of Agents. SAAPL's semantics handles transitions between stages by treating a transition in one stage as a precondition to a rule for the next (e.g., rule (8) in Figure 2),

$$\frac{}{\langle Q, N, B, +b\rangle \xrightarrow{\textsf{Basic}} \langle Q, N, B \cup \{b\}, \epsilon\rangle} \tag{1}$$

$$\frac{}{\langle Q, N, B, -b\rangle \xrightarrow{\textsf{Basic}} \langle Q, N, B \backslash \{b\}, \epsilon\rangle} \tag{2}$$

$$\frac{\Delta = \{P_i\theta \mid (t_i : c_i \leftarrow P_i) \in \Pi \wedge t_i\theta = e \wedge B \models c_i\theta\}}{\langle Q, N, B, e\rangle \xrightarrow{\textsf{Basic}} \langle Q, N, B, \mathcal{S}_{\mathcal{O}}(\Delta)\rangle} \tag{3}$$

$$\frac{\langle Q, N, B, P_1\rangle \xrightarrow{\textsf{Basic}} \langle Q', N, B', P'\rangle}{\langle Q, N, B, P_1; P_2\rangle \xrightarrow{\textsf{Basic}} \langle Q, N, B, P_1; P_2\rangle} \tag{4}$$

$$\frac{}{\langle Q, N, B, \epsilon; P\rangle \xrightarrow{\textsf{Basic}} \langle Q, N, B, P\rangle} \tag{5}$$

$$\frac{}{\langle Q, N, B, \uparrow^{N_B} m\rangle \xrightarrow{\textsf{Basic}} \langle Q + N : N_B : m, N, B, \epsilon\rangle} \tag{6}$$

$$\frac{Q = N_A : N : m + Q'}{\langle Q, N, B, \Gamma\rangle \xrightarrow{\textsf{Agent}} \langle Q', N, B, \Gamma \cup \{\downarrow^{N_A} m\}\rangle} \tag{7}$$

$$\frac{P = \mathcal{S}_{\mathcal{I}}(\Gamma) \qquad \langle Q, N, B, P\rangle \xrightarrow{\textsf{Basic}} \langle Q', N, B', P'\rangle}{\langle Q, N, B, \Gamma\rangle \xrightarrow{\textsf{Agent}} \langle Q', N, B', (\Gamma \backslash \{P\}) \cup \{P'\}\rangle} \tag{8}$$

$$\frac{P = \mathcal{S}_{\mathcal{I}}(\Gamma) \qquad P = \epsilon}{\langle Q, N, B, \Gamma\rangle \xrightarrow{\textsf{Agent}} \langle Q', N, B', \Gamma \backslash \{P\}\rangle} \tag{9}$$

$$\frac{\langle N, B, \Gamma\rangle = \mathcal{S}_{\mathcal{A}}(As) \qquad \langle Q, N, B, \Gamma\rangle \xrightarrow{\textsf{Agent}} \langle Q', N, B', \Gamma'\rangle}{\langle Q, As\rangle \xrightarrow{\textsf{MAS}} \langle Q', (As \cup \{\langle N, B', \Gamma'\rangle\}) \backslash \{\langle N, B, \Gamma\rangle\}\rangle} \tag{10}$$

Fig. 2. Operational Semantics for SAAPL

while the AIL expects a "chaining" style where an agent's reasoning cycle decides when to change stage. We include the current stage as part of the agent data structure.

Let us consider the Basic stage first. Although our agent data structure contains all the intentions, it also distinguishes a "current intention" so in the Basic stage we work on this intention. SAAPL's intentions are stacks of belief modifications, send message actions or events, while the AIL has a more complex structure of events and deeds. In general we will be interested in the AIL's deed stack. Only when planning will we be interested in the AIL's event stack. We will look at rule (1) in detail. We discovered, while performing this case study, that many of the operational rules in `OSRules` were *over complex* and still specialised towards what is now the Gwendolen language. In particular they contained pre-conditions and effects that were unnecessary in many cases. As a case in point, there is a belief addition rule in `OSRules` which, excluding unifiers and irrelevant parts of the Agent data structure, was:

$$\frac{\mathbf{consistent}(B \cup \{b\})}{< ag, (E, +b); i, I, B, \textbf{?} > \rightarrow < ag, i, [(+b, \epsilon)]; I, B \cup \{b\}, \textbf{?} >} \tag{11}$$

In this transition rule, and in all others in the paper, we will include in the agent data-structure $< \ldots >$ only those parts affected by the rule. Throughout this paper we will

use ; to indicate concatenation of the rows in our intention data structure so $(E, +b); i$ is the intention whose top row has event, E and deed, $+b$. **?** indicates the placeholder for the stage of the agent's reasoning cycle. We replaced (11) with

$$\frac{\mathbf{consistent}(B \cup \{b\})}{< ag, (E, +b); i, I, B, \textbf{?} > \rightarrow < ag, i, I, B \cup \{b\}, \textbf{?} >} \tag{12}$$

which does not issue a new intention $(+b, \epsilon)$. **consistent** defaults to $\top$ in the AIL, but can be over-ridden in language implementations. We keep the default and again that pre-condition is trivial. Changing the presentation of the rule to use SAAPL syntax (B for belief base, P the current intention, Γ the intentions, N agent name) gives:

$$\frac{}{< N, (E, +b); P, \Gamma, B, \textbf{Basic} > \rightarrow < N, P, \Gamma, B \cup \{b\}, \textbf{Agent} >} \tag{13}$$

which is in most respects identical to (1) and (4), then returning to the **Agent** stage as specified by (8). The AIL's default "drop belief" rule (combining (2) and (4)) becomes:

$$\frac{}{< N, (E, -b); P, \Gamma, B, \textbf{Basic} > \rightarrow < N, P, \Gamma, B \backslash \{b'\}, \textbf{Agent} >} \tag{14}$$

We now look at plan selection; rule (3). This rule combines two operations that are separated in `OSRules`. Firstly a set, Δ, of applicable plans is determined and then one of these is selected $\mathcal{S}_{\mathcal{O}}(\Delta)$ using a selection function. Since we had two rules to represent this we introduced a new stage **BasicPlanning** to chain them together:

$$\frac{\Delta = \{(t_i, P_i) \mid t_i : c_i \leftarrow P_i \in \Pi \wedge (t_i\theta = e) \wedge B \models c_i\theta\} \quad \Delta \neq \emptyset}{< N, [], (e, \epsilon); P, B, \textbf{Basic} > \rightarrow < N, \Delta, (e, \epsilon); P, B, \textbf{BasicPlanning} >} \tag{15}$$

$$\frac{\mathcal{S}_{\mathcal{O}}(\Delta, i) = (e, P_i)}{< N, (e, \epsilon); P, \Delta, \textbf{BasicPlanning} > \rightarrow < N, (e, P_i); P, [], \textbf{Agent} >} \tag{16}$$

$(e, P_i); P$ is a shorthand for adding a row (e, p_i) to the intention P for each deed, $p_i \in P_i$.

SAAPL has a mechanism for posting events, e. These are placed on the intention stack and picked up immediately for planning. In the AIL data structures they get placed on the deed stack and need to be moved to the event stack before planning can take place. This step requires us to introduce a new rule from `OSRules` into SAAPL:

$$\frac{}{< N, (E, +!_p g); P, \textbf{Agent} > \rightarrow < N, (+!_p g, \epsilon); P, \textbf{Agent} >} \tag{17}$$

The SAAPL semantics requires the use of a new distinguished symbol 'ϵ' to represent a "done" update. Since we have integrated (4) into our rules for individual steps we no longer need this marker nor (5) to handle it.

We now look at sending messages. SAAPL assumes a message queue, while AIL assumes a more adhoc arrangement where an agent can access all its messages at once. Therefore we implemented two new rules for (6) and (7):

$$\frac{\xi.enqueue(N : N_B : m)}{< N, \xi, \uparrow^{N_B} m; P, B, \textbf{Basic} > \rightarrow < N, \xi, P, B \cup \{\uparrow^{N_B} m\} \textbf{Agent} >} \tag{18}$$

$$\frac{N_A : N : m = \xi.dequeue}{< N, \Gamma, \textbf{Agent} > \rightarrow < N, (\downarrow^{N_A} m, \epsilon); \Gamma, \textbf{Agent} >} \quad (19)$$

ξ is the agent environment. The SAAPL implementation therefore specifies two operations ($enqueue$ and $dequeue$) that any environment in which it runs must implement. We treat the receipt of a new message as the acquisition of a belief that the message has been received (so in the AIL this is modelled as a belief change event).

Rules (8) and (9) handle the selection of intentions. Although we are not using the SAAPL "do nothing" symbol we can have empty intentions all of whose deeds have been performed. There were select intention rules within the AIL but, again, they proved too complex and simplified rules were developed.

$$\frac{P = \mathcal{S}_{\mathcal{I}}(\Gamma \cup \{P_o\}) \quad \neg \texttt{empty}(P)}{< N, P_o, \Gamma, \textbf{Agent} > \rightarrow < N, P, \Gamma \backslash \{P\} \cup \{P_o\}, \textbf{Basic} >} \quad (20)$$

$\mathcal{S}_{\mathcal{I}}$ is a function for selecting intentions. Since AIL's default rules expect a separate distinguished current intention the equivalent of Γ in rules (8) and (9) is $\Gamma \cup P_o$ in AIL, where P_o is the "old" current intention. This isn't a complete representation of (8) we have to assume the correct operation of the **Basic** stage to complete the rule.

$$\frac{P = \mathcal{S}_{\mathcal{I}}(\Gamma \cup \{P_o\}) \quad \texttt{empty}(P)}{< N, P_o, \Gamma, \textbf{Agent} > \rightarrow < N, null, \Gamma \backslash \{P\} \cup \{P_o\}, \textbf{Agent}\} >} \quad (21)$$

We also introduced a rule that put an agent's thread to sleep should its intention set Γ become empty (as opposed to letting it continuously run, checking for an applicable rule). Rule (10) implicitly assumes a single threaded environment. In a multi-threaded Java implementation it seemed sensible not to implement this rule in the semantics but allow the Java scheduling algorithm to handle interleaving of agent execution.

Since we were interested in an example which required agents to perform actions beyond simply sending messages, we also introduced one further rule from `OSRules`:

$$\frac{a \neq \uparrow^{N_A} m \quad \xi.\mathbf{do}(a)}{< N, \xi, (E, a); P, \textbf{Basic} > \rightarrow < N, \xi, P, \textbf{Agent} >} \quad (22)$$

Recall that $\mathbf{do}$ is an interface requirement for all environments that support the AIL. It is assumed that an environment for SAAPL would fulfil the basic AIL requirements (implementing $\mathbf{do}$ etc.) as well as those specific to SAAPL ($enqueue$ and $dequeue$).

Faithfulness of the Implementation. Any claim to have implemented the operational semantics of a language is faced with correctness issues involved in transferring a transition system to, in this case, a set of Java classes. Verifying implementations is a complex undertaking. Such a verification effort would be a significant task and falls outside the scope of this work. However, that aside, it is also the case that we have not directly implemented the transition system presented in [28] but the one shown above and so the question arises "Are these two transition systems equivalent"? In fact they are not. For instance we have included a new rule for action execution and have interleaved agent execution. But nevertheless it would clearly be desirable to produce a theorem demonstrating the extent to which the two transition systems match and so providing

a clear idea of the extent to which we can claim to have implemented SAAPL with the AIL. We have offered above an informal discussion of the relationship between the semantics but leave a proper (ideally formal) proof to further work[3]. We anticipate that the production of such a proof will refine the implementation of SAAPL in the AIL.

Contract Net Code. The contract net code in SAAPL is similar to that for Gwendolen. The major difference is the inability to trigger plans by general belief updates. We create a special "react" event, r, used to trigger reactions to messages. The react event is posted when **tell** messages are recieved (see section 7).

$$g : cando(g) \leftarrow a \qquad g' : cando(g') \leftarrow a'$$
$$g : \neg cando(g) \leftarrow cfp(g) \qquad g' : \neg cando(g') \leftarrow cfp(g')$$

$$cfp(T) : ag(A) \wedge name(N) \wedge \sim\uparrow^A (\textbf{perform}, respond(T, N)) \leftarrow \uparrow^A (\textbf{perform}, respond(T, N)); \texttt{wait}$$

$$respond(T, A) : cando(T) \wedge name(N) \leftarrow \uparrow^A (\textbf{tell}, proposal(T, N))$$
$$respond(T, A) : \neg cando(T) \wedge name(N) \leftarrow \uparrow^A (\textbf{tell}, sorry(T, N))$$

$$r : proposal(P, Ag) \leftarrow \uparrow^{Ag} (\textbf{tell}, award(P)) \qquad r : award(T) \leftarrow T$$

6 GOAL

GOAL [8] is a BDI language introduced by de Boer et. al to illustrate the use of purely declarative goals in agent programming. It is clearly a BDI language but is quite different in style to many other agent languages. In particular it does not use the concepts of event or intention explicitly in its semantics. An agent is defined by its mental state: two sets of formulas Σ for the agent's beliefs and Γ for the agent's goals. In this sense, it is closer in style to the original AOP proposal [22] or to MetateM [15]. GOAL assumes an underlying logic on its formula language, $\mathcal{L}$, with an entailment relation $\models_C$. Its semantics then defines entailment for mental states as follows:

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

- $\langle \Sigma, \Gamma \rangle \models_M \mathbf{B}\phi$ *iff* $\Sigma \models_C \phi$,
- $\langle \Sigma, \Gamma \rangle \models_M \mathbf{G}\psi$ *iff* $\psi \in \Gamma$,
- $\langle \Sigma, \Gamma \rangle \models_M \neg\phi$ *iff* $\langle \Sigma, \Gamma \rangle \not\models_M \phi$,
- $\langle \Sigma, \Gamma \rangle \models_M \phi_1 \wedge \phi_2$ *iff* $\langle \Sigma, \Gamma \rangle \models_M \phi_1$ *and* $\langle \Sigma, \Gamma \rangle \models_M \phi_2$.

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

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

[3] Initial steps in this direction are given in [13].

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

$$\mathcal{M}(\boldsymbol{a}, \langle \Sigma, \Gamma \rangle) = \begin{cases} \langle \mathcal{T}(\boldsymbol{a}, \Sigma), & \text{if } \mathcal{T}(\boldsymbol{a}, \Sigma) \\ \quad \Gamma \backslash \{\psi \in \Gamma \mid \mathcal{T}(\boldsymbol{a}, \Sigma) \models_C \psi\} \rangle & \quad \text{is defined,} \\ \text{is undefined for } \boldsymbol{a} \in Bcap & \text{if } \mathcal{T}(\boldsymbol{a}, \Sigma) \\ & \quad \text{is undefined} \end{cases} \tag{23}$$

$$\mathcal{M}(\textbf{\textit{drop}}(\phi), \langle \Sigma, \Gamma \rangle) = \langle \Sigma, \Gamma \backslash \{\psi \in \Gamma \mid \psi \models_C \phi\} \rangle \tag{24}$$

$$\mathcal{M}(\textbf{\textit{adopt}}(\phi), \langle \Sigma, \Gamma \rangle) = \begin{cases} \langle \Sigma, & \text{if } \not\models_C \neg\phi \text{ and} \\ \quad \Gamma \cup \{\phi' \mid \Sigma \not\models_M \phi', \models_C \phi \rightarrow \phi'\} \rangle & \quad \Sigma \not\models_C \phi \\ \text{is undefined} & \text{if } \Sigma \models_C \neg\phi \text{ or} \\ & \quad \models_C \neg\phi \end{cases} \tag{25}$$

Lastly, conditional actions and a *commitment strategy* provide a mechanism for selecting which capability to apply next.

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

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

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

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

GOAL *implemented with AIL.* To model GOAL's mental states we treated the AIL belief base as the GOAL belief base, Σ. AIL already had an operation to extract the "goals" of a agent – interpreted as the set of AIL achieve goals appearing in the event stacks of intentions. GOAL's goal set, Γ, became the AIL's goal set.

Implementation of $\models_M$ was simple. The forumulas $\mathbf{B}(\phi)$ etc. are equivalent to the AIL's guard formulas and the AIL logical consequence relation, $\models$, is allowed to inspect not just AIL's belief base but also its intentions, mailboxes, plans [4] and goals. The AIL interpreted $\mathbf{G}(\phi)$ as $\phi \in \Gamma$ as required by GOAL. Therefore the AIL's $\models$ relation was equivalent to GOAL's $\models_M$ except that the current implementation of $\models$, in the AIL, only allows for unification with the belief base. This therefore limits reasoning about GOAL mental states. (We intend to build in Prolog style reasoning in the future.)

Next we turn to capabilities. Inherent in the description of a capability is the idea that the agent performs an action associated with the capability. Also inherent in the description and in the semantics of mental state transformers is the idea that all the belief updates associated with a capability are performed before the agent does anything else

[4] This allows us to model communication performatives such as *Jason*'s askHow.

(like planning a different intention). The AIL's pre-existing transition rules only allowed for one belief update at a time. There was nothing to prevent us from writing a new rule that would perform all the tasks in $\mathcal{T}(\mathbf{a}, \Sigma)$ at once, but since we were interested in re-using the AIL's pre-existing rules where possible we assigned a reasoning cycle stage, **Capability**, for performing all the updates required by $\mathcal{T}(\mathbf{a}, \Sigma)$.

We treat capabilities as *perform* goals because they function as steps/sub-goals an agent *should* perform yet they are not declarative. The AIL requires the execution of actions to be triggered explicitly so we decided to treat $\mathcal{T}(\mathbf{a}, \Sigma)$ as a function on the belief base paired with an optional action. We write this as $\mathcal{T}(\mathbf{a}, \Sigma) = \mathbf{do}(a) + f(\Sigma)$ and represent it in the AIL as a plan, where the range of f is a deed stack of belief updates. The enabledness of a capability is governed by the plan guard. When $\mathcal{T}$ is executed it first performs the action (if appropriate) and then modifies the belief base. Lastly, it removes any achieved goals. This breaks down the execution of $\mathcal{T}$ into several transition rules. First we modify the deed stack of the intention in accordance with $\mathcal{T}$

$$\frac{\Delta = \{< \mathbf{a}, a'; f'(\Sigma) > | \ \mathbf{a} \in Bcap \wedge enabled(\mathbf{a}) \wedge \mathcal{T}(\mathbf{a}, \Sigma) = \mathbf{do}(a') + f'(\Sigma)\} \quad \mathcal{S}_{\text{plan}}(\Delta) =< \mathbf{a}, a; f(\Sigma) >}{< ag, (\mathbf{a}, \epsilon); i, I, \textbf{Main} > \rightarrow < ag, (\mathbf{a}, a; f(\Sigma)); i', I \backslash \{i'\} \cup \{i\}, \textbf{Capability} >} \tag{26}$$

where $\mathcal{S}_{\text{plan}}$ is an application specific function for selecting a plan from a set. This was a new rule but made use of pre-existing AIL operations, particularly the built-in functions for matching plans to intentions. After applying this rule the system is in the **Capability** stage which ensures that all the changes associated with $\mathcal{T}$ take place before the agent does anything else. We used four pre-existing transitions to handle most of $\mathcal{T}$, three of which (13), (14) and (22) we have already shown leaving us only to provide a special case for when the action to be performed involves sending a message:

$$\frac{\xi.\mathbf{do}(\uparrow^{ag'} m)}{< ag, (E, \uparrow^{ag'} m); i, Out, \textbf{Capability} > \rightarrow < ag, i, Out \cup \{\uparrow^{ag'} m\}, \textbf{Capability} >} \tag{27}$$

Note here how $\uparrow^{ag'} m$ is added to the agent's outbox, Out. This functions as part of the belief base, from now on the agent will believe it has sent the message.

Only deeds associated with a capability have a perform goal as their event. Since a capability can not invoke another capability there will never be two consecutive capabilities on the event stack of an intention. So we trigger the end of capability processing by detecting the event is no longer a perform goal. At this point we need to remove any goals already achieved.

$$\frac{e \neq \mathbf{a} \quad \mathcal{G} = \{g \in \Gamma \mid B \models g\} \quad i' = map(\lambda g.\, \mathbf{drop}(g, (e, d); i), \mathcal{G}) \quad I' = \{i \mid i = map(\lambda g.\, \mathbf{drop}(g, i')) \wedge i' \in I\}}{< ag, (e, d); i, I, \textbf{Capability} > \rightarrow < ag, i', I', \textbf{Perception} >} \tag{28}$$

Just as our implementation of $\models$ does not include anything equivalent to Prolog style reasoning, this rule also avoids dropping goals which follow from goals already achieved. $\mathbf{drop}$ is a built in AIL operation on intentions which removes a goal from the event stack and all subgoals subsequently placed there. GOAL had no semantics for perception or message handling which we needed for our example scenario. We assumed that

these should directly update an agent's belief base. We therefore introduced two new stages to control these with simple new rules (which again are now a part of `OSRules`) using AIL's environment interface:

$$\frac{B_1 = \xi.newPercepts(ag) \qquad B_2 = \xi.oldPercepts(ag)}{< ag, \xi, \Sigma, \textbf{Perception} > \rightarrow < ag, \xi, \Sigma \backslash \{B_2\} \cup \{B_1\}, \textbf{Messages} >} \tag{29}$$

$$\frac{M = \xi.getMessages(ag) \qquad B' = \{\downarrow^A m \mid m \in M\}}{< ag, \xi, \Sigma, \textbf{Messages} > \rightarrow < ag, \xi, \Sigma \cup B', \textbf{Main} >} \tag{30}$$

We now need to deal with the selection and application of conditional actions and capabilities, and the mental state transformers for **adopt** and **drop**.

GOAL has no concept of multiple intentions (using multiple goals instead) however AIL has these goals distributed among intentions. Part of the process of planning with a conditional action must therefore include selecting the appropriate intention. We chose to first find all the plans that were applicable, no matter which intention, and then chose one of those plans, implicitly choosing (or creating) a new intention in the process. Naturally we chose to represent GOAL's conditional actions as AIL plans. We had a choice here as the AIL can have plans which are not triggered by an event but it was more natural within the AIL to use event triggers. We decided therefore that where a condition of a conditional action referred to a goal this would be treated as the event trigger for the plan[5]. We created two new rules for reactive plans and triggered plans:

$$\frac{\Delta = \{ap \mid ap = \phi \rhd \mathbf{do}(a) \wedge ag \models \phi\} \qquad \mathcal{S}_{\text{plan}}(\Delta) = \phi' \rhd \mathbf{do}(a') \qquad \mathbf{G}(\phi'') \notin \phi'}{< ag, i, I, \textbf{Main} > \rightarrow < ag, (+\phi', a), i \cup I, \textbf{Goal}\} >} \tag{31}$$

We use the absense of any goals in the plan's mental condition ($\mathbf{G}(\phi'') \notin \phi'$) to tell that this is a reactive plan. This rule starts a new intention when a reactive plan applies.

$$\frac{\begin{array}{c}\Delta = \{ap \mid ap = \phi \rhd \mathbf{do}(a) \wedge ag \models \phi\} \qquad \mathcal{S}_{\text{plan}}(\Delta) = \phi' \rhd \mathbf{do}(a') \qquad \mathbf{G}(\phi'') \in \phi' \\ (\textbf{adopt}(\phi'), \epsilon); i' \in \{I \cup i\}\end{array}}{< ag, i, I, \textbf{Main} > \rightarrow < ag, (\phi'', a'); i', I \backslash \{i'\} \cup \{i\}, \textbf{Goal}\} >} \tag{32}$$

For triggered plans we modify the intention that triggered the plan. We overrode the AIL's default $\mathcal{S}_{\text{plan}}$ function to prevent continuous firing of reactive rules once they became applicable. This was done by keeping track of how many times a conditional action had been used and, where there was a choice, opting for ones less frequently applied. We also needed to amend $\mathcal{S}_{\text{plan}}$ to distinguish between plans representing capabilities and plans representing conditional actions and to ensure that the correct type of plan was used with the correct transition rule.

We have added a new **Goal** stage because the AIL needs to shift goals between deeds and events. We need rules for both perform and achieve goals. The rule for perform goals was a GOAL equivalent of (17). Similar rules were used to move **adopt**

[5] Although we never used a plan with multiple goals, if we had this would have had to be represented either as a reactive plan (no trigger) or as multiple plans each with one of the goals as a trigger and the other goals as part of the plan context.

deeds and **drop** deeds to the event stack. Once back to the **Main** stage, we either plan with a capability, if that is now the event (as above), or handle the **adopt** (moving an achieve goal to the event stack, above) or **drop** event:

$$\frac{i' = \textbf{drop}(\phi, i) \qquad I' = \{i'_1 \mid i_1 \in I \wedge i'_1 = \textbf{drop}(\phi, i_1)\}}{< ag, (\textbf{drop}(\phi), \epsilon); i, I, \textbf{Main} > \rightarrow < ag, i', I', \textbf{Main} >} \tag{33}$$

We also added a rule to the language to sleep the agent when it had nothing to do.

Faithfulness of the Implementation. We have implemented GOAL with a commitment strategy based on planning *recent* goals (i.e., those goals at the top of an intention's event stack). Our semantics for **drop** are different because AIL's 'drop' function removes subgoals (which may not be explicitly represented as deducible from the dropped goal) – in retrospect an easy way to avoid this would have been to arrange for **adopt** deeds to start new intentions rather than being stacked as sub-goals on existing intentions. We would like to re-implement the example in this fashion before attempting a correctness proof of the equivalence of the two operational semantics.

Contract Net Code. We needed to split our plans for the contract net between capabilities and conditional actions. The requirement that *all* goals be declarative has caused the introduction of capabilities whose purpose is, once they are applicable, to add the belief that a goal is achieved.

Conditional Actions:

$$\begin{array}{rr} \mathbf{G}(g) \wedge \mathbf{B}(cando(g)) \rhd g & \mathbf{G}(g) \wedge \neg\mathbf{B}(cando(g)) \rhd \textbf{adopt}(cfp(g) \\ \mathbf{G}(g') \wedge \mathbf{B}(cando(g')) \rhd g' & \mathbf{G}(g') \wedge \neg\mathbf{B}(cando(g')) \rhd \textbf{adopt}(cfp(g') \end{array}$$

$$\mathbf{G}(cfp(T)) \wedge \mathbf{B}(ag(A)) \wedge \mathbf{B}(name(N)) \wedge \neg\mathbf{B}(send_p(A, respond(T, N))) \rhd \mathbf{do}(\textbf{adopt}(send_p(A, respond(T, N))))$$

$$\mathbf{G}(cfp(T)) \wedge \mathbf{B}(ag(A)) \wedge \mathbf{B}(name(N)) \wedge \mathbf{B}(send_p(A, respond(T, N))) \rhd \mathbf{do}(cfp\_done(T))$$

$$\begin{array}{r} \mathbf{B}(respond(T, A)) \wedge \neg\mathbf{B}(cando(T)) \wedge \mathbf{B}(name(N)) \rhd \mathbf{do}(send_t(A, sorry(T, N))) \\ \mathbf{B}(respond(T, A)) \wedge \mathbf{B}(name(N)) \rhd send_t(A, proposal(T, N) \\ \mathbf{B}(respond(T, A)) \wedge \mathbf{B}(send_t(A, sorry(T, N))) \rhd believe(respond(T, A)) \\ \mathbf{B}(proposal(P, Ag)) \rhd send_t(Ag, award(P)) \\ \mathbf{B}(award(T) \wedge \neg T \rhd \textbf{adopt}(T) \end{array}$$

Capabilities:

$$\begin{array}{rr} \mathcal{T}(g, \Sigma) = \mathbf{do}(a) + \Sigma & \mathcal{T}(g', \Sigma) = \mathbf{do}(a') + \Sigma \\ \mathcal{T}(cfp\_done(T), \Sigma) = \Sigma \cup (cfp(T)) & \mathcal{T}(cfp\_done(T), \Sigma) = \Sigma \cup (cfp(T)) \end{array}$$

$$\begin{array}{l} \mathcal{T}(send_p(A, G), \Sigma) = \mathbf{do}(\uparrow^A (\textbf{perform}, G)) + \Sigma \cup send_p(A, G) \\ \mathcal{T}(send_t(A, G), \Sigma) = \mathbf{do}(\uparrow^A (\textbf{tell}, G)) + \Sigma \cup send_t(A, G) \end{array}$$

7 Execution of the Contract Net Scenario

The Environment. For the Gwendolen and GOAL agents we were able to use the default environment for the AIL, but for SAAPL we needed to implement a message queue on top of this. This involved sub-classing the default environment and implementing $enqueue$ and $dequeue$. We also needed to define **do** for sending messages (already present in the AIL's default environment) and the actions a, a' and `wait`. For `wait` we simply slept the agent for a short time. a and a' introduce new perceptions g and g' into the environment which agents would subsequently detect.

Communication Semantics. Our Contract Net assumed an agreed communication semantics between the agents. In this semantics messages were paired with a performative (either **perform** or **tell**). The semantics of **tell** was simple: on receiving a **tell** message an agent was to update its belief base with the body of the message. The semantics of **perform** was more complex, especially since GOAL did not have perform goals. Its semantics was simply that the agent should commit to that goal (irrespective of whether it was a perform or achieve goal). In the AIL the semantics of communication is supposed to be determined by plans triggered by the receipt (or, where applicable, sending) of messages. This gave us the following additional plans in our environment.

Gwendolen Code

$+\!\downarrow^{Ag}(\textbf{perform}, G) : \top \ \texttt{<-} \ +!_p G$ $\qquad$ $+\!\downarrow^{Ag}(\textbf{tell}, B) : \top \ \texttt{<-} \ +B$

SAAPL Code

$\downarrow^{Ag}(\textbf{perform}, G) : \top \leftarrow G$ $\qquad$ $\downarrow^{Ag}(\textbf{tell}, B) : \top \leftarrow +B : r$

GOAL Code

$$\mathbf{B}(\downarrow^{Ag}(\textbf{perform}, G)) \wedge \neg\mathbf{B}(G) \rhd \textbf{adopt}(G)$$
$$\mathbf{B}(\downarrow^{Ag}(\textbf{tell}, B)) \wedge \neg\mathbf{B}(B) \rhd believe(B)$$
$$\mathcal{T}(believe(B), \Sigma) = \Sigma \cup \{B\}$$

Execution. Figure 3 shows the message sequence (running from top to bottom) for a typical run of our scenario. The Gwendolen agent acts as the manager and GOAL and SAAPL agents bid for contracts. We show the more important goals and beliefs as they are added to, and removed from (represented by striking through) the agents' structure. The Gwendolen agent has two goals g and g'. It sends a message to the SAAPL agent with the content $respond(g')$. This becomes a goal of the SAAPL agent who responds with $sorry(g')$ which at some point later becomes a belief of the Gwendolen agent. Interaction with the environment (actions a and a') and perception are shown by arrows terminating or originating outside of an agent.

8 Conclusions

One of clearest conclusions we drew from our work was that the AIL data structures are sufficiently expressive to represent the concepts in (at least) the languages we had chosen. However our package of pre-existing transition rules was not well designed. This was not perhaps surprising since the transitions in this package were originally intended for a catch-all language. Fortunately, from a model checking perspective, the

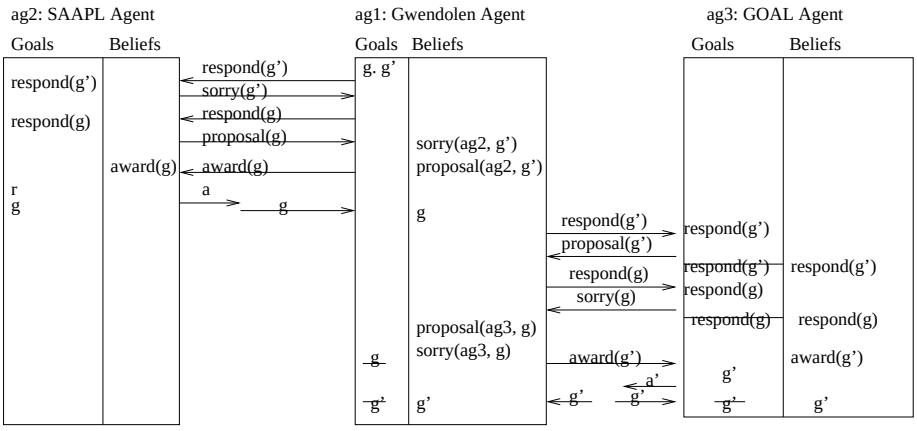

Fig. 3. Typical Execution of the Contract Net Scenario.

data structures are the crucial element needed for reasoning about an agent and we have been able to model check simple programs in our three languages even though the AIL classes are not yet optimised for the model checker. The transition rules were provided as a convenience, and in the hope that some might prove amenable for optimisation. However, it became clear, that the transition rule package needed to be redesigned to provide simple rules with better options for customisation. Yet, even with the burden of more customisation of the transition rules than we had originally anticipated we were able to implement interpreters for SAAPL and GOAL with relative ease. The SAAPL interpreter took about a week to implement and debug while the GOAL interpreter took about two weeks. Once correctly implemented it was simple to incorporate and run a heterogeneous multi-agent system. This, together with the (designed) ability to implement languages such as 3APL and AgentSpeak [11] confirms that the AIL provides a suitable level for implementing most BDI programming language semantics,

In the immediate future we intend to revisit the AIL implementation, in particular `OSRules` and rework it in the light of this work. We also intend to improve the model checking aspects of the framework and to address larger, more complex languages, in particular the *Jason* implementation of AgentSpeak and 3APL and provide full AIL based implementations of these languages.

References

1. F. L. Bellifemine, G. Caire, and D. Greenwood. *Developing MultiAgent Systems with JADE*. 2007.
2. R. H. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni, editors. *Multi-Agent Programming: Languages, Platforms and Applications*. Springer-Verlag, 2005.
3. R. H. Bordini, M. Fisher, W. Visser, and M. Wooldridge. Model Checking Rational Agents. *IEEE Intelligent Systems*, 19(5):46–52, 2004.
4. R. H. Bordini, M. Fisher, W. Visser, and M. Wooldridge. Verifying Multi-Agent Programs by Model Checking. *J. Autonomous Agents and Multi-Agent Systems*, 12(2):239–256, 2006.

5. R. H. Bordini, J. F. Hübner, and M. Wooldridge. *Programming Multi-agent Systems in AgentSpeak Using Jason*. Wiley, 2007.
6. E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, December 1999.
7. M. Dastani, M. B. van Riemsdijk, and J.-J. C. Meyer. Programming Multi-Agent Systems in 3APL. In Bordini et al. [2], chapter 2, pages 39–67.
8. F. S. de Boer, K. V. Hindriks, W. van der Hoek, and J.-J. C. Meyer. A Verification Framework for Agent Programming with Declarative Goals. *J. Applied Logic*, 5(2):277–302, 2007.
9. L. A. Dennis, B. Farwer, R. H. Bordini, and M. Fisher. A Flexible Framework for Verifying Agent Programs. In *Proc. Int. Conf. Autonomous Agents and Multiagent Systems*. ACM, 2008. (Short paper.).
10. L. A. Dennis. Gwendolen: A BDI Language for Verifiable Agents In *Logic and the Simulation of Interaction and Reasoning*. AISB Convention 2008, University of Aberdeen, 2008.
11. L. A. Dennis, B. Farwer, R. H. Bordini, M. Fisher, and M. Wooldridge. A Common Semantic Basis for BDI Languages. In *Proc. 7th Int. Workshop on Programming Multiagent Systems (ProMAS)*, 2007.
12. J. Dix and Y. Zhang. IMPACT: A Multi-Agent Framework with Declarative Semantics. In Bordini et al. [2], chapter 3, pages 69–94.
13. B. Farwer and L. A. Dennis. Translating into an Intermediate Agent Layer: A prototype in Maude. In *Proc. Concurrency, Specification, and Programming (CS&P)*, 2007.
14. FIPA: Foundation for Intelligent Physical Agents. *http://www.fipa.org*.
15. M. Fisher. METATEM: The Story so Far. In *Proc. 3rd International Workshop on Programming Multiagent Systems (ProMAS)*, pages 3-22. Volume 3862 of *LNAI*, Springer, 2005.
16. I. Gungui and V. Mascardi. Integrating tuProlog into DCaseLP to Engineer Heterogeneous Agent Systems. In *Proc. Italian Conf. Computational Logic (CILC)*. University of Parma, Italy, 2004.
17. G. J. Holzmann. *The Spin Model Checker*. Addison-Wesley, November 2003.
18. Java Agent Services. *http://www.java-agent.org*.
19. OMG: Object Management Group. *http://www.omg.org*.
20. S. Paurobally, J. Cunningham, and N. Jennings. Verifying the Contract Net Protocol: A Case Study in Interaction Protocol and Agent Communication Language Semantics. In *Proc. 2nd Int. Workshop on Logic and Communication in Multi-Agent Systems (LCMAS)*, 2004.
21. S. Paurobally, J. Cunningham, and N. R. Jennings. A Formal Framework for Agent Interaction Semantics. In *Proc. 4th Int. Conf. Autonomous Agents and Multiagent Systems*, pages 91–98. ACM, 2005.
22. Y. Shoham. Agent-Oriented Programming. *Artificial Intelligence* 60(1):51–92, 1993.
23. R. G. Smith and R. Davis. Frameworks for Cooperation in Distributed Problem Solving. *IEEE Transactions on Systems, Man, and Cybernetics*, 11(1), 1980.
24. V. S. Subrahmanian, P. Bonatti, J. Dix, T. Eiter, S. Kraus, F. Ozcan, and R. Ross. *Heterogeneous Agent Systems: Theory and Implementation*. MIT Press, 2000.
25. H. Suguri, E. Kodama, and M. Miyazaki. Assuring Interoperability in Heterogeneous, Autonomous and Decentralized Multi-Agent Systems. In *Proc. 6th Int. Symposium on Autonomous Decentralized Systems (ISADS)*, pages 17–24. IEEE, 2003.
26. X. Tan and S. Wang. Implementation of Multi-Agent System Based on CORBA and COM. In *Proc. 6th Int. Conf. Computer Supported Coop. Work in Design*, pages 299–302, 2001.
27. M. Winikoff. JACK$^{\text{TM}}$ Intelligent Agents. In Bordini et al. [2], chapter 7, pages 175–193.
28. M. Winikoff. Implementing Commitment-Based Interactions. In *Proc. 6th Int. Conf. Autonomous Agents and Multiagent Systems*, pages 1–8, ACM, 2007.
29. M. Winikoff, L. Padgham, J. Harland, and J. Thangarajah. Declarative and Procedural Goals in Intelligent Agent Systems. In *Proc. Int. Conf. Principles of Knowledge Representation and Reasoning*, 2002.