Verifying Heterogeneous Multi-Agent Programs

Thu Trang Doan Yuan Yao Natasha Alechina Brian Logan

School of Computer Science
University of Nottingham
Nottingham NG8 1BB, UK
{ttd,yvy,nza,bsl}@cs.nott.ac.uk

ABSTRACT

We present a new approach to verifying *heterogeneous multi-agent programs* — multi-agent systems in which the agents are implemented in different (BDI-based) agent programming languages. Our approach is based on meta-APL, a BDI-based agent programming language that allows both an agent's plans and its deliberation strategy to be encoded as part of the agent program. The agent programs comprising a heterogeneous multi-agent program are first translated into meta-APL, and the resulting system is then verified using the Maude term rewriting system. We prove correctness of translations of *Jason* and 3APL programs and deliberation strategies into meta-APL. Preliminary experimental results indicate that our approach can significantly out-perform previous approaches to verification of heterogeneous multi-agent programs.

Categories and Subject Descriptors

I.2 [**Artificial Intelligence**]: Programming Languages and Software

Keywords

Agent programming languages; Agent programs; Verification

1. INTRODUCTION

Multi-agent systems (MAS) offer a promising approach to the development of large, distributed, intelligent systems. In a multi-agent system, the agents interact via message passing and/or by performing actions in a shared environment. The agents in a MAS are typically loosely coupled, and a key advantage of multi-agent systems is that the individual agents comprising the system can be developed independently by different developers using different programming languages. We call a multi-agent system in which the agents are implemented in different agent programming languages a *heterogeneous multi-agent program*. In this paper, we focus on heterogeneous multi-agent programs in which the individual agents are written in a language in the *Belief-Desire-Intention* (BDI) family of languages, as BDI is arguably the dominant agent programming paradigm [13].

A key challenge in developing a multi-agent system is verifying that it meets its design requirements. This is particularly important as MAS are increasingly being used for safety critical applications. There has been considerable work on the verification of individual agent programs and homogeneous multi-agent programs (where the individual agents are all implemented in the same agent programming language), e.g., [21, 1, 5, 2, 18]. However, with the exception of the work on the Agent Infrastructure Layer (AIL) [4, 9, 10], there has been relatively little work on verification of heterogeneous multi-agent programs. AIL is a collection of Java classes abstracting capabilities of BDI agent programming languages. The interpreters of each language in a heterogeneous multi-agent program are reimplemented using AIL and programs verified using the AJPF model checker.

In this paper, we present a new approach to verifying heterogeneous multi-agent programs based on meta-APL. Meta-APL is a BDI-based agent programming language that allows both an agent's plans and its deliberation strategy to be encoded as part of the agent program. In our approach, the agent programs comprising a heterogeneous multi-agent program are first translated into meta-APL, and the resulting system is then verified using Maude [6]. A compact, declarative representation of deliberation strategies (rather than, e.g., reimplementing the agent interpreter using special purpose libraries as in AIL) makes it easier to ensure that the operational semantics of the target language has been faithfully encoded for verification. While there has been work on expressing deliberation strategies in an agent programming language, e.g., [12, 15, 7], to the best of our knowledge, such an approach has not previously been used in the verification of heterogenous multi-agent systems.

The main contributions of this paper are the definition meta-APL and its operational semantics; provably correct translations of Jason and 3APL programs into meta-APL; a verification framework for meta-APL multi-agent programs based on Maude; and preliminary experimental results which indicate that our approach requires significantly less time to verify properties compared to [10].

2. Meta-APL

In this section, we briefly introduce meta-APL.[1] A meta-APL agent consists of an *agent program* and the *agent state* which is queried and manipulated by the program. The agent's state consists of two main components: the *mental state*, which is a collection of *atom instances*, and the *plan state* which consists of a collection of *plan instances* and their properties. Atom instances are used to represent beliefs, goals, events etc. Plan instances play a role similar to relevant, applicable plans in conventional BDI agent programming languages.

2.1 Meta-APL Syntax

The syntax of Meta-APL is built from atoms, plans, clauses, macros, object rules, and meta-rules, and a small number of primi-

[1] A preliminary version of meta-APL was presented in [22].

tive operations for querying and updating the mental state and plan state of an agent.

Atoms Atoms are built of terms. Terms are defined using the following disjoint sets of symbols: *IDs* which is a non-empty totally ordered set of ids, *Pred* which is a non-empty set of predicate symbols, *Func* which is a non-empty set of function symbols, and *Vars* which is a non-empty set of variables.

The syntax of terms t and atoms a is given by:

$$\begin{array}{lll} t & =_{def} & x \mid f(t_1, \ldots, t_m) \\ a & =_{def} & p(t_1, \ldots, t_n) \end{array}$$

where $f \in Func$, $p \in Pred$, $x \in Vars \cup IDs$, $n \geq 0$, and $m \geq 0$. To distinguish between different instances of syntactically identical atoms (e.g., two instances of the same event), each atom instance is associated with a unique $id \in IDs$.

The atom instances comprising the agent's mental state can be queried and updated using the following primitive operations:

- $\mathsf{atom}(i, a)$: an instance of the atom a has *id* i
- $\mathsf{add\text{-}atom}(i, a)$: create a new instance of the atom a and bind its *id* to i
- $\mathsf{delete\text{-}atom}(i)$: delete the atom instance with *id* i

For brevity, queries may be expressed in terms of atoms rather than instances where the *id* is not important, i.e., the query a is true if the query $\mathsf{atom}(\_, a)$ is true.

Plans A *plan* is a textual representation of a sequence of actions the agent can execute in order to change its environment or its mental state. Plans are built of external actions, mental state tests, reified mental state actions and subgoals composed with the sequence operator ';'. A plan π is defined as:

$$\pi \quad =_{def} \quad \epsilon \mid (ea \mid mt \mid ma \mid sg)\,;\pi$$

where ϵ denotes the empty plan, ea is an external action of the form $\mathrm{e}(t_1, \ldots, t_n)$, $\mathrm{e} \in$ *ActionNames* and $t_1, \ldots t_n$, $n \geq 0$ are ground terms, mt is a mental state test of the form $?\,q$ where q is a (primitive or defined) mental state query, ma is a (primitive or defined) mental state action, and sg is a subgoal of the form $!\,g(u_1, \ldots, u_m)$ where $g(u_1, \ldots, u_m)$ is an atom and $u_1, \ldots u_m$, $m \geq 0$ are (possibly non-ground) terms.

Meta-APL distinguishes between generic plans, which are a static part of the agent program, and plan instances — specific substitutions of generic plans generated during the execution of the program. The plan state of a meta-APL agent may contain multiple instances of the same plan (e.g., if a plan is used to achieve different subgoals). Each plan instance has a unique *id*, a current *suffix* (the part of the instance still to be executed), one or more justifications, a substitution and (at most) one active subgoal. A *justification* is an atom instance *id*. Informally a justification is a 'reason' for executing (this instance of) the plan, e.g., an atom representing a belief or goal. In general, a plan instance may have multiple justifications, and a justification may be the reason for adopting multiple plan instances. The *substitution* $\theta = \{x_1/t_1, \ldots, x_k/t_k\}$ specifies the current binding of variables in the plan instance to terms. A *subgoal* is created by the execution of a subgoal step $!\,g(u_1, \ldots, u_m)$, and is an instance of the atom $g(u_1, \ldots, u_m)$ which shares variables with the subgoal in the plan instance. Each plan instance also has a set of *execution state flags* σ. σ is subset of a set of flags *Flags* which includes at least `intended`, `scheduled`, `stepped` and `failed`, and may contain additional user-defined flags, e.g., some deliberation strategies may require a `suspended` execution state.

The plan instances comprising the plan state of an agent can be queried and updated using the following primitive operations:

- $\mathsf{plan}(i, \pi)$: i is the *id* of an instance of the plan π
- $\mathsf{plan\text{-}remainder}(i, \pi)$: π is the textual representation of the (unexecuted) suffix of the plan instance with *id* i
- $\mathsf{justification}(i, j)$: the plan instance with *id* i has the atom instance with *id* j as a justification
- $\mathsf{substitution}(i, \theta)$: the plan instance with *id* i has substitution θ
- $\mathsf{subgoal}(i, j)$: j is the *id* of the subgoal of the plan instance with *id* i, i.e., $\mathsf{plan\text{-}remainder}(i, !\,g; \pi)$ and $atom(j, g)$ such that j is the *id* of the instance of g created by executing $!\,g$ in i
- $\mathsf{state}(i, \sigma)$: the plan instance with *id* i has execution state flags σ
- $\mathsf{set\text{-}remainder}(i, \pi)$ set the (unexecuted) suffix of the plan instance with *id* i to π
- $\mathsf{set\text{-}substitution}(i, \theta)$: set the substitution of the plan instance with *id* i to θ, where θ may be an implicit substitution resulting from the unification of two terms $t(x) = t(a)$
- $\mathsf{set\text{-}state}(i, \sigma)$ set the execution state flags of the plan instance with *id* i to σ
- $\mathsf{delete\text{-}plan}(i)$: delete the plan instance with *id* i, together with its suffix, substitution and subgoal (if any)
- $\mathsf{cycle}(n)$: the current deliberation cycle is n

Clauses & Macros Additional mental state and plan state queries can be defined using Prolog-style Horn *clauses* of the form:

$$q \leftarrow q_1, \ldots, q_n$$

where $q_1, \ldots, q_n$ are mental or plan state queries or their negation. Negation is interpreted as negation as failure, and we assume that the set of clauses is always *stratified*, i.e., there are no cycles in predicate definitions involving negations. Clauses are evaluated as a sequence of queries, with backtracking on failure.

Additional mental state and plan state actions can be defined using *macros*. A macro is a sequence of mental state and/or plan state queries/tests and actions. Macros are evaluated left to right, and evaluation aborts if an action or query/test fails. For example, the mental state action add-atom(a) which does not return an instance *id* can be defined by the macro: add-atom(b) $=$ $\mathsf{add\text{-}atom}(\_, b)$. Macros can also be used to define *type specific* mental state actions, e.g., to add an instance of the atom b as a belief and signal a belief addition event as in *Jason* [3], we can use the macro

$$\text{add-belief}(b) = \text{add-atom}(\text{belief}(b)), \text{add-atom}(+\text{belief}(b))$$

In what follows, we assume the following clause-definable plan state queries and macro-definable plan state actions: intention(i): the plan instance with *id* i is intended; executable-intention(i): the intention with *id* i has no subgoal (hence no subintention); scheduled(i): a step of the plan instance with *id* i will be executed at the current deliberation cycle; failed(i): the plan instance with *id* i has `failed`; add-intention(i): add the `intended` flag to the plan instance with *id* i.

Object Rules To select appropriate plans given its mental state, an agent uses *object rules*. Object rules correspond to plan selection

constructs in conventional BDI agent programming languages, e.g., plans in *Jason* [3], or PG rules in 3APL [8]. The syntax of an object rule is given by:

$$\textit{reasons}\ [\ :\ \textit{context}\ ]\ \rightarrow \pi$$

where *reasons* is a conjunction of non-negated primitive mental state queries, *context* is boolean expression built of mental state queries and π is a plan. The context may be null (in which case the ":" may be omitted), but each plan instance must be justified by at least one reason. The reason and the context are evaluated against the agent's mental state and both must return true for π to be selected. Firing an object rule gives rise to a new instance of the plan π that forms the right hand side of the rule which is justified by the atom instances matching the *reasons*.

Meta-rules To update the agent's state, specify which plan instances to adopt as intentions and select which intentions to execute in a given cycle an agent uses *meta-rules*. The syntax of a meta-rule is given by:

$$\textit{meta-context} \rightarrow m_1;\ \ldots\ ; m_n$$

where *meta-context* is a boolean expressions built of mental state and plan state queries and $m_1, \ldots, m_n$ is a sequence of mental state and/or plan state actions. When a meta-rule is fired, the actions that form its right hand side are immediately executed.

Meta-APL Programs A meta-APL program $(D, \mathcal{R}_1, \ldots, \mathcal{R}_k, A)$ consists of a set of clause and macro definitions D, a sequence of rule sets $\mathcal{R}_1, \ldots, \mathcal{R}_k$, and a set of initial atom instances A. Each *rule set* $\mathcal{R}_i$ is a set of object rules or a set of meta-rules that forms a component of the agent's deliberation cycle. For example, rule sets can be used to update the agent's mental and plan state, propose plans or create and execute intentions.

2.2 Meta-APL Core Deliberation Cycle

The meta-APL core deliberation cycle consists of three main phases. In the first phase, a user-defined $sense()$ function updates the agent's mental state with atom instances resulting from perception of the agent's environment, messages from other agents etc. In the second phase, the rule sets comprising the agent's program are processed in sequence. The rules in each rule set are run to quiescence to update the agent's mental and plan state. Each rule is fired once for each matching set of atom and/or plan instances. Changes in the mental and plan state resulting from rule firing directly update the internal (implementation-level) representations maintained by the deliberation cycle, which may allow additional rules to match in the same or subsequent rule sets. Processing a set of object rules creates new plan instances. Processing a set of meta-rules may involve updating the agent's beliefs and goals, deleting intentions for achieved goals, deleting unintended plan instances from the previous deliberation cycle, updating the agent's intentions or selecting which intention(s) to execute at this cycle, etc. Finally, in the third phase, the next step of all `scheduled` object-level plans is executed. The deliberation cycle then repeats. Cycles are numbered starting from 0 (initial cycle), and the cycle number is incremented at each new cycle.

3. OPERATIONAL SEMANTICS

In this section, we give the operational semantics of meta-APL in terms of a transition system. We first present the configuration of meta-APL agent programs (henceforth agent configuration) before presenting the transition rules. Each transition transforms one configuration into another and corresponds to a single computation/execution step.

Meta-APL Configuration An agent configuration is a tuple $\langle D, \mathcal{R}_1 \ldots \mathcal{R}_k, A, \Pi, J, S, p, n\rangle$ where D is a set of clause and macro definitions, each $\mathcal{R}_i, 1 \leq i \leq k$ is a either a set of object rules or a set of meta-rules, A is a set of atom instances, Π is a set of plan instances, $J \subseteq (ID \times ID)$ is a justification relation between plan instance ids and justification ids, $S \subseteq (ID \times ID)$ is a sub-goal relation between plan instance ids and subgoal ids, $0 \leq p \leq k+2$ is a phase indicator (where 0 is the *sense* phase, $1 \leq p \leq k$ correspond to the rule sets, and $k{+}1$, $k{+}2$ correspond to the *exec* phase), and $n \in \mathbb{N}$ is the current deliberation cycle. An atom instance $\alpha \in A$ is a tuple (i, a, c) where: i is the *id* of the instance, a is the atom of which i is an instance, and c is the cycle at which the instance was created. A plan instance $\rho \in \Pi$ is a tuple (i, π, π', f, c) where: i is the *id* of the instance, π is the plan of which i is an instance, π' is the remainder of i, f is the set of execution state flags of i (a subset of {`failed`, `stepped`, `scheduled`, `intended`} and any user-defined flags), and c is the cycle at which the instance was created. Since the agent's clause and macro definitions, and object and meta-rules do not change during execution, we use $\langle A, \Pi, J, S, p, n\rangle$ to denote the configuration when no ambiguity can arise.

The initial configuration of an agent is defined by its initial atom instances A_0, and execution starts in the *sense* phase: $\langle A_0, \emptyset, \emptyset, \emptyset, 0, 0\rangle$.

Mental & Plan State Queries The evaluation of a query with respect to a configuration results in a substitution θ which is the most general unifier (mgu) of the query and some element of the configuration. Given t_1 and t_2, we write $t_1 = t_2 \mid \theta$ iff t_1 and t_2 unify with mgu θ.

Below we state how each primitive query type is evaluated against a configuration $C = \langle A, \Pi, J, S, p, n\rangle$

- $C \vdash \mathsf{atom}(i, a) \mid \theta$ iff $\exists (i_1, a_1, c_1) \in A$: $(i_1, a_1) = (i, a) \mid \theta$
- $C \vdash \mathsf{cycle}(c)$ iff $c = n$
- $C \vdash \mathsf{plan}(i, \pi) \mid \theta$ iff $\exists (i_1, \pi_1, \pi_1', f_1, j_1) \in \Pi$: $(i_1, \pi_1) = (i, \pi) \mid \theta$
- $C \vdash \mathsf{plan\text{-}remainder}(i, \pi) \mid \theta$ iff $\exists (i_1, \pi_1, \pi_1', f_1, j_1) \in \Pi$ such that $(i_1, \pi_1') = (i, \pi) \mid \theta$
- $C \vdash \mathsf{justification}(i, j) \mid \theta$ iff $\exists (i_1, j_1) \in J$: $(i_1, j_1) = (i, j) \mid \theta$
- $C \vdash \mathsf{substitution}(i, \mu) \mid \theta$ iff $\exists (i_1, \pi_1, \pi_1'\theta_1', f_1, j_1) \in \Pi$ such that $(i_1, \theta_1') = (i, \mu) \mid \theta$
- $C \vdash \mathsf{subgoal}(i, j) \mid \theta$ iff $\exists (i_1, j_1) \in S$: $(i_1, j_1) = (i, j) \mid \theta$
- $C \vdash \mathsf{state}(i, f) \mid \theta$ iff $\exists (i_1, \pi_1, \pi_1', f_1, j_1) \in \Pi$: $(i_1, f_1) = (i, f) \mid \theta$
- $C \vdash \mathrm{not}(q) \mid \theta$ iff there is no θ' for the variables in q left unsubstituted by θ such that $C \vdash q \mid \theta\theta'$
- $C \vdash q_1, q_2, \ldots, q_n \mid \theta$ iff $\exists \theta_1, \ldots, \theta_n$ such that $\theta = \theta_1 \ldots \theta_n$ and $C \vdash q_1 \mid \theta_1, \ldots, C \vdash q_n \mid \theta_1 \ldots \theta_n$
- $C \vdash q \mid \theta$ where $q \leftarrow q_1, \ldots, q_n$ iff $C \vdash q_1, q_2, \ldots, q_n \mid \theta$

Mental & Plan State Actions For each mental and plan state action act, we define a binary relation $\xrightarrow{act}$ on configurations that describes the resulting configuration when act is performed.

- $\langle A, \Pi, J, S, p, n\rangle \xrightarrow{\mathsf{add\text{-}atom}(i,a)} \langle A \cup \{(i, a, n)\}, \Pi, J, S, p, n\rangle$ where i is a new *id*
- $\langle A, \Pi, J, S, p, n\rangle \xrightarrow{\mathsf{delete\text{-}atom}(i)} \langle A', \Pi', J', S', p, n\rangle$ where
  $A' = A \setminus \{(i', \_, \_) \in A \mid (i, i') \in (S \cup J^{-1})^*\}$
  $\Pi' = \Pi \setminus \{(i', \_, \_, \_, \_) \in \Pi \mid (i, i') \in (S \cup J^{-1})^*\}$

$S' = \{(i', j') \in S \mid (i', \_, \_, \_, \_) \in \Pi', (j', \_, \_) \in A'\}$
$J' = \{(i', j') \in J \mid (i', \_, \_, \_, \_) \in \Pi', (j', \_, \_) \in A'\}$
(above, R^* denotes the reflexive transitive closure of a binary relation R).

- $\langle A, \Pi, J, S, p, n\rangle \xrightarrow{\text{set-substitution}(i,\theta)} \langle A, \Pi', J, S, p, n\rangle$ where $\Pi' = \Pi \setminus \{(i, \pi, \pi'\theta', f, c) \in \Pi\} \cup \{(i, \pi, \pi'\theta, f, c)\}$

- $\langle A, \Pi, J, S, p, n\rangle \xrightarrow{\text{set-state}(i,\sigma)} \langle A, \Pi', J, S, p, n\rangle$ where $\Pi' = \Pi \setminus \{(i, \pi, \pi', f, c)\} \cup \{(i, \pi, \pi', \sigma, c)\}$

- $\langle A, \Pi, J, S, p, n\rangle \xrightarrow{\text{delete-plan}(i)} \langle A', \Pi', J', S', p, n\rangle$ where $\Pi' = \Pi \setminus \{(i', \_, \_, \_, \_) \in \Pi \mid (i, i') \in (S \cup J^{-1})^*\}$
$A' = A \setminus \{(i', \_, \_, \_) \in A \mid (i, i') \in (S \cup J^{-1})^*\}$
$S' = \{(i', j') \in S \mid (i', \_, \_, \_, \_) \in \Pi', (j', \_, \_) \in A'\}$
$J' = \{(i', j') \in J \mid (i', \_, \_, \_, \_) \in \Pi', (j', \_, \_) \in A'\}$

Action sequences Given the definitions of actions above, we can now specify the effects of sequences of actions and macros. Both are defined by non-empty sequences $m_1; \ldots; m_k$ where each element m_i is a primitive or macro-defined mental or plan state query or action.

$$\langle A_0, \Pi_0, J_0, S_0, p, n\rangle \xrightarrow{m_1;\ldots;m_k} \langle A_j, \Pi_j, J_j, S_j, p, n\rangle$$

iff $\langle A_{i-1}, \Pi_{i-1}, J_{i-1}, S_{i-1}, p, n\rangle \xrightarrow{m_i} \langle A_i, \Pi_i, J_i, S_i, p, n\rangle$ for every $1 \leq i \leq j$, and either $j = k$ or $j < k$ and there is no m_{j+1} transition out of $\langle A_j, \Pi_j, J_j, S_j, p, n\rangle$.

3.1 Transition Rules

The execution of a meta-APL agent program modifies its initial configuration by means of transitions that are derivable from the transition rules given below.

Sense In the *sense* phase, the agent's mental state is updated by the user-defined $sense()$ function

$$\frac{A' = sense(env, A)}{\langle A, \Pi, J, S, 0, n\rangle \longrightarrow \langle A', \Pi', J, S, 1, n\rangle}$$

The definition of $sense()$ depends on the nature of the agent's interaction with its environment and its deliberation cycle, but typically results in the addition and/or removal of atom instances.

Apply In the *apply* phase the rules in each rule set $\mathcal{R}_a, 1 \leq a \leq k$ are run to quiescence to update the agent's mental and plan state.

If $\mathcal{R}_a$ contains object rules, a plan instance is created for each applicable object rule. An object rule $(r : c \rightarrow \pi) \in \mathcal{R}_a$ is applicable if its condition evaluates to true in the current configuration under substitution θ, and if there is no plan instance in the plan base with exactly the same plan body and justifications

$$\frac{\begin{array}{l}\exists(r : c \rightarrow \pi) \in R_a : \langle A, \Pi, J, S, a, n\rangle \vdash (r : c) \mid \theta \wedge \\ \nexists(i, \pi\theta, \_, \_, \_) \in \Pi : \{(i, j) \mid j \in ids(r\theta)\} \subseteq J, \\ \Pi' = \Pi \cup \{(i_{\text{new}}, \pi\theta, \pi\theta, \{\,\}, n)\} \text{ where } i_{\text{new}} \text{ is a new id}, \\ J' = J \cup \{(i_{\text{new}}, j) \mid j \in ids(r\theta)\}\end{array}}{\langle A, \Pi, J, S, a, n\rangle \longrightarrow \langle A, \Pi', J', S, a, n\rangle}$$

The new plan instance $\pi\theta$ is added to the plan base, and its justifications recorded. The function $ids(q)$ collects the *ids* of the atom instances used to answer the query $r = q_1, q_2, \ldots, q_n$. When no more object rules can be applied, the phase is advanced to $a + 1$

$$\frac{\begin{array}{l}\forall(r : c \rightarrow \pi) \in R_a : \langle A, \Pi, J, S, a, n\rangle \vdash (r : c) \mid \theta \\ \quad \Longrightarrow \exists(i, \pi\theta, \_, \_) \in \Pi : \{(i, j) \mid j \in ids(r\theta)\} \subseteq J\end{array}}{\langle A, \Pi, J, S, a, n\rangle \longrightarrow \langle A, \Pi, J, S, a + 1, n\rangle}$$

If $\mathcal{R}_a$ contains meta-rules, the actions in the body of each applicable meta-rule in $\mathcal{R}_a$ are executed. A meta-rule $(c \rightarrow \pi) \in \mathcal{R}_a$ is applicable if its condition evaluates to true in the current configuration.

$$\frac{\begin{array}{l}\exists(c \rightarrow \pi) \in R_a : \langle A, \Pi, J, S, a, n\rangle \vdash c \mid \theta \\ \langle A, \Pi, J, S, a, n\rangle \xrightarrow{\pi\theta} \langle A', \Pi', J', S', a, n\rangle\end{array}}{\langle A, \Pi, J, S, a, n\rangle \longrightarrow \langle A', \Pi', J', S', a, n\rangle}$$

When no more meta-rules can be applied, the phase is advanced to $a + 1$

$$\frac{\forall(c \rightarrow \pi) \in R_a : \langle A, \Pi, J, S, a, n\rangle \nvdash c}{\langle A, \Pi, J, S, a, n\rangle \longrightarrow \langle A, \Pi, J, S, a + 1, n\rangle}$$

Exec The *exec* phase consists of two sub-phases. In the first $p = k + 1$ sub-phase, the `stepped` flags of plan instances executed at the previous cycle are deleted, and the phase is advanced to $k + 2$

$$\frac{\begin{array}{l}\Pi' = \Pi \setminus \{(i, \pi, \pi', f \cup \{\texttt{stepped}\}, m) \in \Pi\} \cup \\ \quad \{(i, \pi, \pi', f, m) \in \Pi \mid (i, \pi, \pi', f \dot\cup \{\texttt{stepped}\}, m) \in \Pi\}\end{array}}{\langle A, \Pi, J, S, k + 1, n\rangle \longrightarrow \langle A, \Pi', J, S, k + 2, n\rangle}$$

where $\dot\cup$ denotes the disjoint union operator over sets.

In the second $p = k + 2$ sub-phase, one step of each `scheduled` plan instance is executed. If the step completes successfully, the `scheduled` flag is replaced by `stepped`; if the action fails, `scheduled` is replaced with `failed`.

External actions are performed in the agent's environment. We assume that each external action can signal whether the action succeeded or failed. The first transition handles the case in which the action succeeds

$$\frac{\begin{array}{l}\exists(i, \_, ea; \pi, f, m) \in \Pi : \texttt{scheduled} \in f \ \wedge ea \text{ succeeds} \\ \Pi' = \Pi \setminus \{(i,\_,ea;\pi,f,m)\} \cup \{(i,\_,\pi, f \setminus \{\texttt{scheduled}\} \cup \{\texttt{stepped}\}, m)\}\end{array}}{\langle A, \Pi, J, S, k + 2, n\rangle \longrightarrow \langle A, \Pi', J, S, k + 2, n\rangle}$$

The second transition handles the case where the action fails

$$\frac{\begin{array}{l}\exists(i, \_, ea; \pi, f, m) \in \Pi : \texttt{scheduled} \in f \ \wedge ea \text{ fails} \\ \Pi' = \Pi \setminus \{(i,\_,ea;\pi,f,m)\} \cup \{(i,\_,\pi, f \setminus \{\texttt{scheduled}\} \cup \{\texttt{failed}\}, m)\}\end{array}}{\langle A, \Pi, J, S, k + 2, n\rangle \longrightarrow \langle A, \Pi', J, S, k + 2, n\rangle}$$

Mental state tests are evaluated against the configuration. If the test is successful, the resulting substitution is applied to the plan instance

$$\frac{\begin{array}{l}\exists(i, \_, ?\, b; \pi, f, m) \in \Pi : \texttt{scheduled} \in f \wedge \langle A, \Pi, J, S, k + 2, n\rangle \vdash b \mid \theta \\ \Pi' = \Pi \setminus \{(i,\_,?b;\pi,f,m)\} \cup \{(i,\_,\pi\theta, f \setminus \{\texttt{scheduled}\} \cup \{\texttt{stepped}\}, m)\}\end{array}}{\langle A, \Pi, J, S, k + 2, n\rangle \longrightarrow \langle A, \Pi', J, S, k + 2, n\rangle}$$

If the test fails, the `scheduled` flag is replaced by `failed`

$$\frac{\begin{array}{l}\exists(i, \_, ?\, b; \pi, f, m) \in \Pi : \texttt{scheduled} \in f \wedge \langle A, \Pi, J, S, k + 2, n\rangle \nvdash b \mid \theta \\ \Pi' = \Pi \setminus \{(i,\_,?b;\pi,f,m)\} \cup \{(i,\_,\pi, f \setminus \{\texttt{scheduled}\} \cup \{\texttt{failed}\}, m)\}\end{array}}{\langle A, \Pi, J, S, k + 2, n\rangle \longrightarrow \langle A, \Pi', J, S, k + 2, n\rangle}$$

A mental state action ma, where ma is add-atom or delete-atom, updates the agent's configuration

$$\frac{\begin{array}{l}\exists(i, \_, ma; \pi, f, m) \in \Pi : \texttt{scheduled} \in f \\ \langle A, \Pi, J, S\rangle \xrightarrow{ma} \langle A', \Pi', J', S'\rangle \\ \Pi'' = \Pi' \setminus \{(i,\_,ma;\pi,f,m)\} \cup \{(i,\_,\pi, f \setminus \{\texttt{scheduled}\} \cup \{\texttt{stepped}\}, m)\}\end{array}}{\langle A, \Pi, J, S, k + 2, n\rangle \longrightarrow \langle A', \Pi'', J', S', k + 2, n\rangle}$$

The evaluation of a subgoal results in the creation of a new instance of the goal atom (with the substitution of the plan instance applied to any variables in the goal), together with a subgoal relation associating the atom and plan instances

$$\frac{\begin{array}{l}\exists(i, \_, !\, g; \pi\theta, f, m) \in \Pi : \texttt{scheduled} \in f \\ A' = A \cup \{(i_{\text{new}}, g\theta, n)\} \text{ where } i_{\text{new}} \text{ is a new } id \\ \Pi' = \Pi \setminus \{(i,\_,!g;\pi\theta,f,m)\} \cup \{(i,\_,\pi\theta, f \setminus \{\texttt{scheduled}\} \cup \{\texttt{stepped}\}, m)\} \\ S' = S \cup \{(i, i_{\text{new}})\}\end{array}}{\langle A, \Pi, J, S, k + 2, n\rangle \longrightarrow \langle A', \Pi', J, S', k + 2, n\rangle}$$

When all `scheduled` plan instances have been processed, the *sense* phase of the next deliberation cycle begins

$$\frac{\forall(i,\_,a;\pi,f,j)\in\Pi : \texttt{scheduled}\notin f}{\langle A,\Pi,J,S,k+2,n\rangle \longrightarrow \langle A,\Pi,J,S,0,n+1\rangle}$$

4. CORRECT TRANSLATION

In this section we show how *Jason* and 3APL programs (and their associated deliberation strategies) can be translated into meta-APL to give equivalent behavior under weak bisimulation equivalence.

First we introduce the notion of weak bisimulation and justify using it to compare agent programs. The notion was introduced in [19] and applied to agent programs in [14]. The formal definition of weak bisimulation is given below. A transition system consists of a set of states/configurations S and a set of transitions $\xrightarrow{a}$ between states as specified by the operational semantics of the language, where the label a of the transition is either an external action or any other transition/internal action τ. The set of possible transition labels for a program with a set of external actions Act is denoted by Act^τ (it corresponds to $Act \cup \{\tau\}$). We use an abbreviation $s \stackrel{l}{\Rightarrow} s'$ to say that there is a path from s to s' labelled with a sequence of labels l, and skip all τs from the label (so that if the path from s to s' contains only τ transitions, we say that $s \stackrel{\epsilon}{\Rightarrow} s'$ (the label of the path is the empty string ϵ). The function $observe$ returns observable or meaningful properties of the agent's state, for example beliefs.

DEFINITION 1 (WEAK BISIMULATION). *Let $(S, \{\xrightarrow{a}|\ a \in Act^\tau\})$ and $(T, \{\xrightarrow{a}|\ a \in Act^\tau\})$ be two transition systems. A relation $\cong \subseteq S \times T$ is a weak bisimulation if for any $s \cong t$, it is the case that:*

1. *$observe(s) = observe(t)$,*
2. *if $s \xrightarrow{\tau} s'$, then there exists $t' \in T$ such that $t \stackrel{\epsilon}{\Rightarrow} t'$ and $s' \cong t'$; if $s \xrightarrow{a} s'$ where $a \in Act$, then there exists $t' \in T$ such that $t \stackrel{a}{\Rightarrow} t'$ and $s' \cong t'$, and*
3. *if $t \xrightarrow{\tau} t'$, then there exists $s' \in S$ such that $s \stackrel{\epsilon}{\Rightarrow} s'$ and $s' \cong t'$; if $t \xrightarrow{a} t'$ where $a \in Act$, then there exists $s' \in S$ such that $s \stackrel{a}{\Rightarrow} s'$ and $s' \cong t'$.*

The reason for using weak rather than strong bisimulation (which matches all properties of states and all transitions) is that the latter notion is not informative when comparing programs written in different languages. Clearly, the same behaviour in two different languages may have to be implemented using a different number and type of internal operations, and the sets of state variables used by the two programs will probably be different. We want to match only the external actions produced by the two programs, and 'meaningful' properties of states, such as beliefs. The translations given below generate agent programs that are equivalent in this sense.

4.1 Jason

We assume the syntax and operational semantics defined in [3, Ch.10]. Given a *Jason* program (bs, ps), where bs are the agent's initial beliefs and ps are the agent's plans, we translate it into a meta-APL program $(A, D, \mathcal{R}_1, \mathcal{R}_2, \mathcal{R}_3)$. D defines macros for adding beliefs and goals and a query for plan triggering events:

add-belief(b) = add-atom(belief(b)); add-atom(+belief(b))
delete-belief(b) = delete-atom(belief(b)); add-atom(-belief(b))
trigger-event$(i) \leftarrow$ atom(i,e), plan-trigger(e), not justification$(_,i)$

$\mathcal{R}_1$ contains meta-rules to remove non-intended plan instances from the previous cycle, to remove completed intentions, and to select an event to process at this cycle:

plan$(i,_)$, not intention$(i) \rightarrow$ delete-plan(i)

intention(i), plan-remainder(i,ϵ), justification(i,j), not subgoal$(_,j) \rightarrow$ delete-atom(j)

intention(i), plan-remainder(i,ϵ), justification(i,j), subgoal(k,j), substitution(i,s_i), substitution(k,s_k) $\rightarrow$ set-substitution$(k, s_i \cup s_k)$, delete-atom(j)

cycle(c), not selected-event$(_,c)$, trigger-event(i) $\rightarrow$ add-atom(selected-event(i,c))

$\mathcal{R}_3$ contains meta-rules to nondeterministically select a plan instance for the selected event, to select an intention to execute at this cycle, and to generate a test goal addition event if the intention selected at this cycle starts with a test goal that evaluates to false:

cycle(c), selected-event(i,c), not (justification(j',i), intention(j')), justification$(j,i) \rightarrow$ add-intention(j)

not scheduled$(_)$, executable-intention$(i) \rightarrow$ schedule(i)

scheduled(i), plan-remainder$(i, ?q\,;\pi)$, not q $\rightarrow$ set-remainder$(i, !(+test(q))\,;\pi)$

Together D, $\mathcal{R}_1$ and $\mathcal{R}_3$ define the *Jason* deliberation cycle, and are common to all *Jason* programs.

A and $\mathcal{R}_2$ are specified by a translation function tr. tr transforms each belief $b \in bs$ into two atom instances $tr(b) = (i_b,$ belief$(b), 0)$, $(i'_b,$ +belief$(b), 0)$ representing the belief b and the corresponding belief addition event. tr transforms each *Jason* plan $te_i : c_i \leftarrow h_i \in ps$ into an atom instance plan-trigger$(tr(te_i))$ (where $tr(te_i)$ is given below) in A and a corresponding meta-APL rule in $\mathcal{R}_2$, with the translations of te_i, c_i and h_i forming the reason, context and plan respectively. The translation of te_i depends on its type and is given by: tr(+b) = +belief(b), tr(-b) = -belief(b), tr(+!g) = +goal(g) and tr(+?g) = +test(g). Each element of the plan context $c_i = c_i^1 \& \ldots \& c_i^k$ is transformed into a corresponding mental state query: $tr([not]c_i^j) = [not]$belief(c_i^j) (with "&" replaced by ","). The definition of the plan body translation $tr(h_i)$ is similarly straightforward. External actions and subgoals are unchanged. Test goals are translated into corresponding mental state tests, and the addition and deletion of beliefs are translated into corresponding type specific mental state actions defined using macros: $tr(ea) = ea$, $tr(!g)$ = !+goal(g), $tr(?b)$ = ?belief(b), tr(+b) = add-belief(b), tr(-b) = delete-belief(b) and $tr(h_i^1; \ldots; h_i^n) = tr(h_i^1); \ldots; tr(h_i^n)$.

Before stating the weak bisimulation result for the *Jason* translation into meta-APL, we need to define which properties of the states of *Jason* and meta-APL programs are observable (the function $observe$). For a *Jason* configuration s, we stipulate that if the phase of s is ProcMsg, $observe(s)$ returns (Bs, Es, Is), where Bs are (suitable representations of) the agent's beliefs, Es the relevant events in the event base, and Is are the uncompleted intentions in the intention base. Otherwise $observe(s) = \top$ where $\top$ is an empty observation. For a meta-APL configuration t we stipulate that observation of t is possible if the value of the stage counter is 0 (the agent is in 0 phase). Then $observe(t) = (Bs, Es, Is)$ where Bs are instances of beliefs, Es of relevant events in the atom base, and Is of uncompleted intentions in the intention base. Otherwise $observe(t)$ returns $\top$.

THEOREM 1. *Every* Jason *program (bs, ps) is weakly bisimilar to its meta-APL translation $tr(bs, ps) = (A, D, \mathcal{R}_1, \mathcal{R}_2, \mathcal{R}_3)$.*

The proof (and the proof of Theorem 3 below for 3APL) uses a

general result which states that if there is a strong bisimulation between cycles in two transition systems (where cycles intuitively correspond to agent deliberation cycles, starting and ending with the sense phase), then there is a weak bisimulation between the two transition systems. We state the main definitions needed for this result, and the theorem itself, below.

Let $(S, \{\xrightarrow{a} | \ a \in Act^{\tau}\}$ be a transition system and s_0 the initial state in this transition system. Consider a tree unravelling of the system starting in s_0. Let us denote the states in the tree unravelling by $RC(s_0)$ and let $SC(s_0) \subseteq RC(s_0)$ be the set of configurations which correspond to the beginning of a deliberation cycle (sensing, processing messages etc.).

DEFINITION 2 (DELIBERATION CYCLE). *A deliberation cycle of $RC(s_0)$ is a finite sequence of transitions $s_1 \xrightarrow{a_1} \ldots \xrightarrow{a_{n-1}} s_n$ $(n > 1)$ where:*

- $s_i \in RC(s_0)$ *for all* $1 \leq i \leq n$,
- $a_i \in Act^{\tau}$ *for all* $1 \leq i < n$,
- $s_1, s_n \in SC(s_0)$, *and*
- $s_i \notin SC(s_0)$ *for all* $1 < i < n$.

For a cycle $c = s_1 \xrightarrow{a_1} \ldots \xrightarrow{a_{n-1}} s_n$, we define $first(c) = s_1$ and $last(c) = s_n$. If $s = s_i$ for some $i \in \{1, \ldots, n\}$, we write $label(c|s)$ to denote the label of the prefix of c from $first(c)$ until s, i.e., $label(c|s) = label(s_1 \xrightarrow{a_1} \ldots \xrightarrow{a_{i-1}} s_i)$.

Let $DC(s_0)$ denote the set of all deliberation cycles of $RC(s_0)$. We define *transitions* between consecutive deliberation cycles in $RC(s_0)$ as follows. Given $c, c' \in DC(s_0)$, $c \xrightarrow{l} c'$ iff $last(c) = first(c')$ and $l = label(c)$. For any $c \in RC(s_0)$, $observe(c) = observe(first(c))$. Slightly abusing notation, we write $s \in c$ if s occurs in c. We also lift the notation $c \sim d$ from pairs of cycles to pairs of sets of cycles by setting $C \sim D$ iff for $\forall\, c \in C$, $\exists\, d \in D$ such that $c \sim d$, and $\forall\, d \in D$, $\exists\, c \in C$ such that $c \sim d$.

THEOREM 2. *Let s_0 and t_0 be two initial configurations. If there exists a strong bisimulation $\sim \subseteq DC(s_0) \times DC(t_0)$ where, for any $c \sim d$, the following conditions hold:*

1. *$\forall\, s \in c$ where $s \neq last(c)$, $\exists\, t \in d$ such that $t \neq last(d)$, $label(c|s) = label(d|t)$, $observe(s) = observe(t)$ and $\{c' \in DC(s_0) \mid first(c) = first(c'), s \in c'\} \sim \{d' \in DC(t_0) \mid first(d) = first(d'), t \in d'\}$,*
2. *$\forall\, t \in d$ where $t \neq last(d)$, $\exists\, s \in c$ such that $s \neq last(c)$, $label(d|t) = label(c|s)$, $observe(t) = observe(s)$ and $\{c' \in DC(s_0) \mid first(c) = first(c'), s \in c'\} \sim \{d' \in DC(t_0) \mid first(d) = first(d'), t \in d'\}$,*

then, $RC(s_0)$ and $RC(t_0)$ are weakly bisimilar.

This technical result is useful since it allows us to prove weak bisimulation equivalence between two transition systems if we can prove that there is a strong bisimulation between the cycles. The main idea of constructing such a strong bisimulation between deliberation cycles of a *Jason* program and its meta-APL translation, is to determine the selections that have been made within the deliberation cycles of both agents. In particular, each *Jason* deliberation cycle has at most three selections: a selected event for which there is at least one relevant plan; an applicable plan among these relevant plans to add to the set of intentions; and an intention from the set of intentions to execute. Similarly, each deliberation cycle of its meta-APL translation also has at most three selections: a selected event by the fourth meta rule in R_1; a new plan instance (corresponding to an applicable plan in *Jason*) to become an intention by the first meta rule in R_3; and an intention to be executed by the second meta rule in R_3. The strong bisimulation is constructed by matching deliberation cycles of a *Jason* agent and its meta-APL translation based on the selections made in each deliberation cycle. In the proof, we show that such a construction yields a strong bisimulation which also satisfies the two conditions stated in Theorem 2. This gives us the proof that there is a weak bisimulation between the two transition systems.

4.2 3APL

A similar translation can be defined for 3APL. We assume the syntax and operational semantics given in [8]. We translate a 3APL program (cs, bs, gs, pg, pr), where cs are the capabilities (i.e., belief update actions), bs are the initial beliefs, gs are the initial goals, pg are the planning goal rules and pr are the plan revision rules, into a meta-APL program $(A, D, \mathcal{R}_1, \mathcal{R}_2, \mathcal{R}_3, \mathcal{R}_4)$ where $\mathcal{R}_1$ and $\mathcal{R}_4$ implement the 3APL deliberation strategy, and A, D, $\mathcal{R}_2$ and $\mathcal{R}_3$ are specified by a translation function tr. The translation of A and D are similar to *Jason* except that capabilities are translated into macros in D defined in terms of mental state actions. In the interests of brevity we focus on the translation of the planning goal and plan revision rules that constitute a 3APL agent's program, and the meta-APL rules that implement the 3APL deliberation strategy.

$\mathcal{R}_1$ and $\mathcal{R}_4$ are common to all 3APL programs. $\mathcal{R}_1$ contains meta-rules to remove goals which are believed, non-intended plan instances from the previous cycle and completed intentions.

$\text{goal}(g), \text{belief}(g) \rightarrow \text{delete-atom}(goal(g))$
$\text{revise-plan}(i, p) \rightarrow \text{delete-atom}(\text{revise-plan}(i, p))$
$\text{plan}(i, \_), not(\text{intention}(i)) \rightarrow \text{delete-plan}(i)$
$\text{intention}(i), \text{plan-remainder}(i, \epsilon) \rightarrow \text{delete-plan}(i)$

$\mathcal{R}_4$ contains meta-rules to select a plan instance to revise, a plan instance for each goal or belief, and an intention to execute at this cycle.

$\text{cycle}(c), \text{not selected-PR}(\_, c), \text{revise-plan}(i, p_b)$
$\quad \rightarrow \text{add-atom}(\text{selected-PR}(i, c)), \text{set-remainder}(i, p_b)$

$\text{cycle}(c), \text{not selected-PG}(\_, c), \text{justification}(i, r),$
$\quad \text{not}\ (\text{justification}(j, r), \text{intention}(j))$
$\quad \rightarrow \text{add-atom}(\text{selected-PG}(i, c)), \text{add-intention}(i)$

$\text{not scheduled}(\_), \text{intention}(i), \text{not failed}(i) \rightarrow \text{schedule}(i)$

$\mathcal{R}_2$ and $\mathcal{R}_3$ contain the translation of the 3APL program. The translation function tr transforms each 3APL planning goal rule $\kappa \leftarrow \beta \mid \pi \in pg$ into a corresponding meta-APL rule in $\mathcal{R}_2$, with the translations of κ, β and π forming the reason, context and plan respectively. The translation of κ is given by $tr(\kappa) = \text{goal}(\kappa)$. Each element of the belief condition β is transformed into a corresponding mental state query. When translating the plan body $tr(\pi)$, external actions are unchanged $tr(ea) = ea$, test goals are translated into corresponding mental state tests, $tr(b?) = ?b$ and mental actions are translated into corresponding mental state macros in D. Abstract plans are translated as corresponding 'external' actions which always fail (causing the plan to block). tr translates each plan revision rule $\pi_h \leftarrow \beta \mid \pi_b \in pr$ as a meta-rule in $\mathcal{R}_3$

$\text{plan}(i, \pi_h), \text{intention}(i), tr(\beta) \rightarrow \text{add-atom}(\text{revise-plan}(i, \pi_b))$

where $tr(\beta)$ is a set of mental state queries corresponding to β, and the atom $\text{revise-plan}(i, \pi_b)$ indicates that plan instance i can be revised with π_b.

Before stating the equivalence result, we need to define the $observe$ function used in stating weak bisimulation equivalence. For a 3APL configuration s, we stipulate that s is observable if it is in Message phase. Then $observe(s) = (\sigma, \gamma, I)$ where σ are beliefs in the belief base, γ are goals in the goal base and I are intentions in the

intention base. Otherwise $observe(s) = \top$. For a meta-APL configuration t which is in 0 phase, $observe(t) = (Bs, Gs, Is)$ where Bs are beliefs in the atom base, Gs are goals in the atom base, and Is are uncompleted intentions in the intention base. Otherwise $observe(t) = \top$.

THEOREM 3. *Every 3APL program (bs, gs, pg, pr) is weakly bisimilar to its meta-APL translation $tr(bs, gs, pg, pr) = (A, D, \mathcal{R}_1, \mathcal{R}_2, \mathcal{R}_3, \mathcal{R}_4)$.*

The proof uses Theorem 2 and a construction of strong bisimulation between cycles in the transition system of a 3APL program and cycles in the transition system of its meta-APL translation. Similar to the translation of *Jason*, such a strong bisimulation is constructed by matching deliberation cycles of the 3APL agent and its meta-APL translation. Here, each deliberation cycle of 3APL has at most three selections: an applicable PG rule to apply; an applicable PR rule to apply; and an intention to execute. Correspondingly, each deliberation cycle of the meta-APL translation makes at most three selections: a new plan instance (generated by the translation of a PG rule in R_2) to become an intention by the second meta-rule of R_4; an intention to be revised by the translation of a PR rule in R_3 and the first meta-rule of R_4; and an intention to be executed by the third meta rule of R_4. We show that this construction gives rise to a strong bisimulation which satisfies two conditions of Theorem 2.

5. VERIFICATION

Translation of the individual agent programs comprising a heterogeneous multi-agent program results in a set of meta-APL programs that interact through a common environment. As shown in the previous section, the meta-APL translations have equivalent behavior under weak bisimulation equivalence to the original heterogeneous multi-agent program.

In this section, we briefly outline how we verify properties of this set of meta-APL programs using Maude [6] and its associated LTL model checker. Maude has previously been used both to prototype agent languages and for verification, e.g., [24, 11, 23]. It can model check systems whose states involve arbitrary algebraic data types—the only assumption is that the set of states reachable from a given initial state is finite. Compared to propositional model checkers, this greatly simplifies modeling of the agents' (first-order) rules and deliberation strategies.

The Maude encoding of meta-APL consists of two parts: a set of Maude modules defining types, equations and rules that encode the meta-APL operational semantics, and a set of additional modules generated by the tr function that encode the concrete meta-APL program of each agent. The encoding of the meta-APL operational semantics requires approximately 60 equations and 15 rules (about 300 lines of Maude code).

5.1 Example

As an example of our approach, we present verification results for a variant of the 'Mars scenario' [5]. In the Mars scenario, two robots cooperate to remove garbage from the surface of Mars. Robot `r1` searches for two pieces of garbage randomly positioned in a grid environment. When it finds a piece of garbage, it brings the piece to robot `r2` which incinerates the garbage. *Jason* programs for `r1` and `r2` are given in [5]. We replaced the program for `r2` with an equivalent 3APL program. Both programs and their deliberation cycles were translated as described in Section 4, and we used Maude to verify the six properties given in [5]. As expected, all properties hold of the combined heterogeneous multi-agent program.

Bordini et al. [5, 2] also give the time required to model check the property

$$((\textbf{Int}\ \texttt{r1}\ continue(check) \wedge (\textbf{Bel}\ \texttt{r1}\ checking(slots)))$$

using both the SPIN and Java Pathfinder (JPF) model checkers. The SPIN approach relies on an encoding of *Jason* programs and deliberation strategy in the SPIN modeling language PROMELA. With JPF, the model checker is used to model check the Java code implementing the *Jason* interpreter as it executes the agent programs. For SPIN, verification required 65.8 seconds, and with JPF verification required over 18 hours. When the garbage is placed at fixed positions on the grid (i.e., the system has a single initial state), verification requires 5.25 seconds for SPIN and 76.3 seconds for JPF.

The approaches presented by Bordini and colleagues are specific to the *Jason* agent programming language. Dennis et al [10] verified a variant of the Mars scenario using their generic approach to MAS verification which allows the verification of heterogeneous multi-agent programs. They report a time of 9 hours to model check the property above for a system in which the *Jason* programs for `r1` and `r2` were translated into the agent programming language GWENDOLEN.

Using our framework, verification with Maude requires 362 seconds (for multiple initial states) and 1.8 seconds when the garbage is at fixed positions in the grid. Although our approach is slower than the SPIN encoding of the (homogeneous) *Jason* implementation of the Mars scenario for multiple initial states, it is significantly faster than the AIL approach of Dennis et al to verifying heterogeneous multi-agent programs.

6. RELATED WORK

There is a considerable amount of work on verifying BDI agent programs and multi-agent systems, e.g., [21, 1, 5, 2, 18]. However, with the exception of the work on the Agent Infrastructure Layer (AIL) [4, 9, 10], there has been relatively little work on verification of heterogeneous multi-agent programs. AIL is a collection of Java classes abstracting capabilities of BDI agent programming languages. The interpreters of each language in a heterogeneous multi-agent program are reimplemented using AIL, and programs verified using the AJPF model checker. (In [11] a prototype implementation in Maude of the AIL and a translation of AgentSpeak(L) into the Maude AIL is described. However, no proof of correctness of the translation is given, and subsequent work on AIL focussed on AJPF.) To verify heterogeneous BDI programs using AIL, an encoding of the target APL's operational semantics must therefore be defined in Java. In [10] the authors state that there are currently no formal proofs that the AIL translations of the BDI languages they consider (GOAL, SAAPL, etc.) are correct. Indeed Dennis et al. note such correctness results would be a "significant task". A key advantage of our approach is that we can prove a correspondence between the operational semantics of the target BDI language and the operational semantics of its translation into meta-APL, and hence guarantee that the meta-APL translation has identical behavior to the heterogeneous MAS being verified. In contrast, with AIL, both the target of translation and the model-checker operate on a lower level of abstraction, which makes it more difficult to prove correctness of the translation and may also explain differences in model-checking performance.

Due to limited space, we can only acknowledge some of the research which influenced the design of meta-APL, including work on BDI agent programming languages which provide support for programming their own deliberation cycle, e.g., [12, 15, 7, 8], architectures and frameworks for programming BDI deliberation cycles, e.g., [20, 10], and work on translating between BDI agent programming languages, e.g., [16, 14].

7. CONCLUSIONS

We defined meta-APL, a BDI-based agent programming language that allows both an agent's plans and its deliberation strategy to be encoded as part of the agent program. We gave the operational semantics of meta-APL and showed that it is possible to give provably correct translations of *Jason* and 3APL programs into meta-APL. We briefly outlined a verification framework for meta-APL multi-agent programs based on Maude. The translations of *Jason* and 3APL to meta-APL and from meta-APL to Maude are relatively simple (much simpler than an encoding of a BDI language for a model checker). Moreover, preliminary experimental results indicate that our approach requires significantly less time to verify properties compared to [10].

In future work, we plan to extend our approach to other BDI agent programming languages. Based on our experience with *Jason* and 3APL, we are confident that languages such as GOAL [17] can translated in a straightforward way. While each additional language will require a proof of the correctness of the translation into meta-APL, our existing Maude verification framework can be used without modification to model check the resulting translation.

8. REFERENCES

[1] R. Bordini, M. Fisher, C. Pardavila, and M. Wooldridge. Model checking AgentSpeak. In *Proceedings of the 2nd International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2003)*, pages 409–416. ACM, 2003.

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

[3] R. Bordini, J. Hübner, and M. Wooldridge. *Programming multi-agent systems in AgentSpeak using Jason*. Wiley, 2008.

[4] R. H. Bordini, L. A. Dennis, B. Farwer, and M. Fisher. Automated verification of multi-agent programs. In *23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008)*, pages 69–78. IEEE, 2008.

[5] R. H. Bordini, M. Fisher, W. Visser, and M. Wooldridge. Verifiable multi-agent programs. In *Programming Multi-Agent Systems*, volume 3067 of *LNCS*, pages 72–89. Springer, 2004.

[6] M. Clavel, S. Eker, P. Lincoln, and J. Meseguer. Principles of Maude. In J. Meseguer, editor, *Electronic Notes in Theoretical Computer Science*, volume 4. Elsevier Science Publishers, 2000.

[7] M. Dastani, F. de Boer, F. Dignum, and J. Meyer. Programming agent deliberation: an approach illustrated using the 3APL language. In *Proceedings of the 2nd International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2003)*, pages 97–104. ACM, 2003.

[8] M. Dastani, M. B. van Riemsdijk, and J.-J. C. Meyer. Programming multi-agent systems in 3APL. In *Multi-Agent Programming: Languages, Platforms and Applications*, pages 39–67. Springer, 2005.

[9] L. A. Dennis, B. Farwer, R. H. Bordini, and M. Fisher. A flexible framework for verifying agent programs. In *Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2008)*, pages 1303–1306. IFAAMAS, 2008.

[10] L. A. Dennis, M. Fisher, M. P. Webster, and R. H. Bordini. Model checking agent programming languages. *Automated Software Engineering*, 19(1):5–63, 2012.

[11] B. Farwer and L. Dennis. Translating into an intermediate agent layer: A prototype in Maude. In *Proceedings of Concurrency, Specification, and Programming CS&P2007*, pages 168–179, 2007.

[12] M. P. Georgeff and A. L. Lansky. Reactive reasoning and planning. In *Proceedings of the Sixth National Conference on Artificial Intelligence, AAAI-87*, pages 677–682, 1987.

[13] M. P. Georgeff, B. Pell, M. E. Pollack, M. Tambe, and M. Wooldridge. The Belief-Desire-Intention model of agency. In *Intelligent Agents V, Agent Theories, Architectures, and Languages, 5th International Workshop, (ATAL'98)*, volume 1555 of *LNCS*, pages 1–10. Springer, 1999.

[14] K. Hindriks. *Agent programming languages: programming with mental models*. PhD thesis, University of Utrecht, 2001.

[15] K. Hindriks, F. De Boer, W. Van der Hoek, and J. Meyer. Agent Programming in 3APL. *Autonomous Agents and Multi-Agent Systems*, 2(4):357–401, 1999.

[16] K. Hindriks, Y. Lespérance, and H. Levesque. An embedding of ConGolog in 3APL. In *Proceedings of the 14th European Conference on Artificial Intelligence (ECAI'2000)*, pages 558–562, 2000. ECAI, IOS Press.

[17] K. V. Hindriks. Programming rational agents in GOAL. In A. El Fallah Seghrouchni, J. Dix, M. Dastani, and R. H. Bordini, editors, *Multi-Agent Programming: Languages, Tools and Applications*, pages 119–157. Springer US, 2009.

[18] S.-S. T. Q. Jongmans, K. V. Hindriks, and M. B. van Riemsdijk. Model checking agent programs by using the program interpreter. In *Proceedings of the 11th International Workshop Computational Logic in Multi-Agent Systems (CLIMA XI)*, volume 6245 of *LNCS*, pages 219–237. Springer, 2010.

[19] R. Milner. *Communication and concurrency*. Prentice-Hall, Inc., 1989.

[20] P. Novák and J. Dix. Modular BDI architecture. In *Proceedings of the 5th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2006)*, pages 1009–1015. ACM, 2006.

[21] S. Shapiro, Y. Lespérance, and H. Levesque. The cognitive agents specification language and verification environment for multiagent systems. In *Proceedings of the 1st International Joint conference on Autonomous Agents and Multiagent Systems*, pages 19–26. ACM, 2002.

[22] T. T. Doan, N. Alechina, and B. Logan. The agent programming language meta-APL. In *Proceedings of the Ninth International Workshop on Programming Multi-Agent Systems (ProMAS 2011)*, pages 72–87, 2011.

[23] M. B. van Riemsdijk, L. Astefanoaei, and F. S. de Boer. Using the Maude term rewriting language for agent development with formal foundations. In M. Dastani, K. V. Hindriks, and J.-J. Ch. Meyer, editors, *Specification and Verification of Multi-agent Systems*, pages 255–287. Springer, 2010.

[24] M. B. van Riemsdijk, F. S. de Boer, M. Dastani, and J.-J. C. Meyer. Prototyping 3APL in the maude term rewriting language. In *Proceedings of the 5th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2006)*, pages 1279–1281. ACM, 2006.