On Agent Programming Language Support for Rational Communication

Yves Lespérance
lesperan@cse.yorku.ca

Steven Shapiro
steven@cs.toronto.edu

Dept. of Computer Science and Engineering, York University, Toronto, ON M3J 1P3, Canada

Résumé :
It is generally thought that agents in a multiagent system should be capable of communicating in a high-level, speech acts-based, agent communication language (ACL), for instance FIPA ACL. Many agent programming languages (APLs) such as Jack, JASON, JADE, etc., support communication in such ACLs, i.e., the production and reception/processing of ACL messages. They also support the specification of complex behaviors for agents, typically by allowing the definition of a library of hierarchical plans that are selected and executed at run time based on external or internal events. However, rational communication requires more than the ability to send and receive ACL messages ; the agent should understand the semantics and pragmatics of the associated communication acts. Without this, agents can only interact when they follow rigid protocols, which is difficult in open systems. Most APLs do not support any of this. In this paper, we argue that this is a serious deficiency that needs to be addressed and discuss the requirements for this. There is one tool that does support the processing of FIPA ACL message semantics and pragmatics, the JADE Semantic Add-On (JSA). But, JSA provides only limited support for the specification of complex agent behaviors. To gain a better understanding of the problem, we explore how one could combine JSA-like semantics and pragmatics processing capabilities with the complex behavior specification capabilities of a typical APL.

1 Introduction and Motivation

It is generally thought that agents in a multiagent system should be capable of communicating in a high-level, speech acts-based, agent communication language (ACL), for instance the ACL standardized by the Foundation for Intelligent Physical Agents (FIPA) [9] or KQML [11]. This allows agents not only to query/inform each other about what they know and request the performance of actions, but to make commitments to one another by making proposals and agreeing to a proposal or request. Such languages have a communication act layer separate from the content layer that can be used to make the illocutionary force (performative) of a message explicit. This allows some processing of messages, for example by a broker, without requiring detailed understanding of their content. FIPA ACL provides a semantic specification of the meaning of the available communication acts in the FIPA Semantic Language (SL), which is essentially a modal logic dealing with beliefs, intentions, and action.

Many frameworks for programming multiagent systems support the production, reception and processing of FIPA ACL messages, for instance Jack [5], JASON [2], JADE [1], IndiGolog [7] with the IG-JADE-PKSlib library [19], etc. However in nearly all cases, these tools process messages only in terms of their syntax and do not support the semantic interpretation of messages and reasoning about their illocutionary/rational effects and the associated agent beliefs and intentions. In fact, agents are generally assumed to interact only according to specified protocols (e.g., contract net, query-reply, etc.) where the types of performatives that may be exchanged in a conversation are rigidly specified (many such interaction protocols have also been standardized by FIPA). The agents only accept messages with the performatives (and content languages/ontologies) allowed by the protocol, and process them based on their performative label and content, without access to their semantics. As pointed out in [17, 18], this means that agents can only interact when they have been designed to interact and that it is very difficult to get agents in an open system to work together.

To solve this problem, one needs agent programming facilities that support rational communication, i.e., the semantic/pragmatic interpretation of FIPA ACL messages as well as reasoning over the resulting semantic representations. This would allow agents to understand the meaning/pragmatics of the messages they receive and to respond to them in a reasonable way, even when messages do not come according to a rigid protocol. Support for semantic/pragmatic interpretation of messages also has many other benefits. There are many ways to perform the same communication act, e.g., one can make an implicit request by informing someone that one has a particular intention. An agent that really understands the meaning of messages will be able to see beyond the difference in surface form. When applying cooperation principles, an agent with rational communication capabilities can exploit its understanding of the pragmatics of the exchanged messages.

For instance, it can detect lying/deception when receiving messages whose claims are inconsistent with what is known about the sender's beliefs/intentions. Furthermore, an agent with rational communication capabilities can in principle deal with various forms of communication failure. When it detects a false presupposition in a message, it can correct it. If a question or request is unclear, it can ask for a clarification. When a direct answer to a query is unlikely to achieve the ultimate goal of the requester, it can provide a helpful answer addressing the latter. One can easily imagine this sort of capabilities being very advantageous/profitable in an e-commerce agent for example.

There is one tool that attempts to provide the kind of rational communication facilities discussed above, the JADE Semantic Add-On (JSA) [17, 18]. JSA provides a library for manipulating FIPA ACL messages, and FIPA SL representations of the semantics of these messages. Semantic interpretation and pragmatic inference, including cooperative belief/intention adoption, is performed in a rule-based framework that can be customized. Facilities are also provided for storing SL semantic representations in a knowledge base and defining forward chaining and backward chaining inference rules over them.

Another requirement for agent programming tools is support for the specification of complex behaviors. The main approach to this is belief-desire-intention (BDI) agent programming languages/architectures, such as PRS [10] and its various successors, such as AgentSpeak [23], JASON [2], Jack [5], JAM [14], and CAN [30, 28], as well as the closely related 3APL [13]. These BDI agent programming languages were conceived as a simplified and operationalized version of the BDI (Belief, Desire, Intention) model of agency, which is rooted in philosophical work such as Bratman's [4] theory of practical reasoning and Dennett's theory of intentional systems [8].

An important feature of BDI-style programming languages and platforms is their interleaved account of sensing, deliberation, and execution [22]. In BDI systems, *abstract plans* written by programmers are combined and executed in real-time. By executing as they reason, BDI agents reduce the likelihood that decisions will be made on the basis of outdated beliefs and remain responsive to the environment by making adjustments in the steps chosen as they proceed. Unlike in classical planning-based architectures, *execution* happens at each step. The assumption is that the careful crafting of plans' preconditions to ensure the selection of appropriate plans at execution time, together with a built-in mechanism for retrying alternative options, will usually ensure that a successful execution is found, even in the context of a changing environment. There is also work on using learning to improve the way plans are selected [12, 15].

Another popular approach to agent programming is the Golog [16] family of high-level programming languages. In contrast to BDI APLs, programming languages in the Golog line aim for a middle ground between classical planning and normal programming. The idea is that the programmer may write a *sketchy* nondeterministic program involving domain specific actions and test conditions and that the interpreter will reason about these and search for a valid execution. The semantics of these languages is defined on top of the *situation calculus*, a popular predicate logic framework for reasoning about action [20, 25]. The interpreter for the language uses an action theory representing the agent's beliefs about the state of the environment and the preconditions and effects of the actions to find a provably correct execution of the program. By controlling the amount of nondeterminism in the program, the high-level program execution task can be made as hard as classical planning or as easy as deterministic program execution. In ConGolog [6], the language was extended to support concurrent programming. Finally in IndiGolog [7], the framework was generalized to allow the programmer to control planning/lookahead and support online execution and sensing the environment.

There are also other approaches to agent programming languages, for instance approaches based on process algebra or temporal logic. Bordini et al. [3] discusses many of these.

Clearly, it would be very useful to have an agent programming framework that provides both (JSA-like) communication and semantics/pragmatics processing capabilities as well as the complex behavior specification capabilities of a typical BDI or Golog-style agent programming language. However, developing such a tool presents very significant challenges. To be able to represent the illocutionary/perlocutionary effects of communication acts, one needs a very expressive logical framework with quantifiers and modal op-

erators for beliefs, intentions, time/action, etc. (e.g., FIPA SL). Entailment in such a logic is undecidable. Even if we restrict attention to the propositional fragment, deciding entailment has very high complexity. Thus using a complete reasoner for such a logic is incompatible with the reactivity requirements of agents operating in highly dynamic environments. One possible approach is to try to identify fragments of such logics where reasoning is decidable and practically feasible, along the lines of work on description logics. But as far as we know, no formalism of this type has ever been used to model the effects of a wide range of communication acts. An alternative approach is to use an incomplete set of inference rules and/or search control heuristics in performing inference. This means that the user must assume responsibility for the range of inferences handled and must often craft customized inference rules/proof strategies for his application. One example of this approach is the ARTIMIS [26] rational agent framework, where a modal logic theorem prover with customized proof strategies is used to implement cooperative dialogue systems (many elements of the ARTIMIS framework have been incorporated into JSA).

To get a better understanding of the issues involved in developing an agent programming framework that supports complex behavior specification facilities and rational communication, we examine in this preliminary project report how one could go about combining JSA with a typical BDI or Golog-style agent programming language. In the next section, we go over JSA and IndiGolog in a bit more detail. Then, we outline a generic architecture that combines JSA and an APL. We keep it generic so that it remains applicable to many different APLs. After that, we examine how the model could be realized with IndiGolog as the APL. We conclude by summarizing our results so far and discussing the work that remains.

2 Background

2.1 JSA

As mentioned earlier, agent communication languages (ACLs) are languages shared between agents that are used to formulate messages sent between them. In this paper, we focus on FIPA ACL. A FIPA ACL message contains a performative and may (and usually does) include other parameters, such as a sender, a receiver, and the content of the message. The performative indicates what the message is intended to achieve. Examples of performative are : Inform, Request, Inform Ref (which informs the recipient of the referent of a descriptor), Call for Proposal, Agree, Cancel, etc. Examples of FIPA ACL messages are given below.

FIPA ACL identifies a set of primitive and composite communication actions (such as Inform and Request) and gives them a semantics using a first-order, multimodal, multiagent logic called the FIPA Semantic Language (FIPA SL). FIPA SL has the following modalities :
- $(B\ i\ \phi)$: agent i believes ϕ
- $(U\ i\ \phi)$: agent i is uncertain whether ϕ holds
- $(I\ i\ \phi)$: agent i intends ϕ
- $(\textit{done}\ a\ \phi)$: action a has just occurred and ϕ held beforehand
- $(\textit{feasible}\ a\ \phi)$: action a can possibly occur and ϕ would hold if it did

The semantics of communicative acts in FIPA ACL are defined using *feasibility conditions* and *rational effects*. A feasibility condition specifies the condition that must hold for an act to be performed. A rational effect is the effect an agent intends to bring about by performing the act.

The main activity of a JSA agent is interpreting incoming FIPA ACL messages. The interpretation loop has access to a belief base which stores facts and maintains a history of previously executed actions (using FIPA SL as its representation language), and a planner. There are two main processes in the JSA interpretation loop, one is a *production function* and the other is a *consumption function*. The objects that these functions operate over (i.e., produce or consume) are FIPA SL sentences which are called *semantic representations* (SRs). When a message comes in, the production function reads the message and produces SRs according to the type and content of the message. The consumption function reads these SRs and acts on these SRs by, e.g., asserting beliefs, executing actions, and/or producing more SRs.

The production and consumption functions are implemented using *semantic interpretation principles* (SIPs). The SIPs are rules that operationalize the FIPA ACL semantics (via predefined SIPs) but can also be used to customize the behavior of agents in a particular domain, implement planning or complex activities, etc. (via user-defined SIPs).

As an example of the interpretation process

[17], suppose a *son* agent wants to know the current temperature from a *display* agent. There are different ways to formulate this request in FIPA ACL. One way is for the son to inform the display that it wants to know the current temperature. This is expressed by the following action which we denote *Act1*.

(*inform :sender son :receiver* (*:set display*)
:content ("(*I son* (*exists* ?*t*
(*B son* (*temperature* ?*t*)))))").

This is an inform action from the son to a set of recipients, which in this case is just the display agent. The content of the action is that the son intends that there be a t such that the son believes that t is the current temperature. When the display agent receives this message, the fact that the son agent sent the message is asserted in the belief base of the display agent :

(*B display* (*done* (*action son Act1*) *true*)).

This belief triggers the ActionFeature SIP which asserts (among other beliefs) the belief in the rational effect of the action. In this case, the rational effect is that the son intends that the display believes that the son intends to know the temperature.

(*B display* (*I son*
(*B display* (*I son*
(*exists* ?*t* (*B son* (*temperature* ?*t*)))))))).

A belief of this form, i.e., that the agent believes that another agent intends the agent to believe something, triggers the BeliefTransfer SIP. This is a user-customizable SIP which when fired causes the agent to adopt a belief of another agent. In this case, it causes the display to adopt the belief intended by the son :

(*B display* (*I son*
(*exists* ?*t* (*B son* (*temperature* ?*t*))))).

This type of belief, i.e., that the agent believes another agent has a particular intention triggers an IntentionTransfer SIP. This user-customizable SIP, when fired, causes the agent to adopt the intention of the other agent.

(*I display*
(*exists* ?*t* (*B son* (*temperature* ?*t*)))).

At this point, the RationalityPrinciple SIP kicks in. This SIP searches the agent's base of actions for an action whose rational effect matches the intention. If the content of the intention is to achieve a goal in the agent's environment, then the PlanningSIP could be triggered, which would try to create a plan to achieve the goal. In this case, the result of the SIP will be to produce an action which informs the son of the current temperature.

(*Inform-ref :sender display :receiver* (*:set son*)
:content "((*any* ?*t* (*temperature* ?*t*)))").

JSA can be customized in various ways. In addition to writing SIPs or modifying the default ones, users can write assert and query filters. These are called when a sentence is asserted or queried (resp.) in the belief base. A filter contains a trigger which matches certain sentences. If the trigger matches, then the filter is applied to the sentence. The conditions for belief and intention transfer are also customizable. These are the conditions under which an agent will adopt the beliefs or intentions of other agents. Intention transfer is the main mechanism to implement cooperation among agents in this framework.

2.2 The Situation Calculus and IndiGolog

The *situation calculus* [20, 25] is a predicate logic language for representing dynamically changing worlds in which all changes are the result of named *actions*. The constant S_0 is used to denote the *initial situation*, namely that situation in which no actions have yet occurred. There is a distinguished binary function symbol *do*, where *do*(a, s) denotes the successor situation to s resulting from performing the action a. Relations (resp. functions) whose values vary from situation to situation, are called *fluents*, and are denoted by predicate (resp. function) symbols taking a situation term as their last argument. There is a special predicate *Poss*(a, s) used to state that action a is executable in situation s.

Within this language, one can formulate action theories that describe how the world changes as the result of the available actions as in [25] :
- Axioms describing the initial situation, S_0.
- Action precondition axioms, one for each primitive action a, characterizing *Poss*(a, s).
- Successor state axioms, one for each relational fluent F (resp. functional fluent f), which characterize the conditions under which $F(\vec{x}, do(a, s))$ holds (resp. $f(\vec{x}, do(a, s)) = v$) in terms of what holds

in situation s ; these axioms may be compiled from effects axioms, but provide a solution to the frame problem [24].
– Unique names axioms for the primitive actions.
– A set of foundational, domain independent axioms for situations Σ.

On top of situation calculus action theories, logic-based programming languages can be defined, which, in addition to the primitive actions, allow the definition of complex actions. The ConGolog language [6], an extension of Golog [16], provides the following rich set of programming constructs :

α,	primitive action
$\phi?$,	wait for a condition
$\delta_1; \delta_2$,	sequence
$\delta_1 \mid \delta_2$,	nondeterministic branch
$\pi\, x.\, \delta$,	nondet. choice of argument
δ^*,	nondeterministic iteration
if ϕ **then** δ_1 **else** δ_2 **endIf**,	conditional
while ϕ **do** δ **endWhile**,	while loop
$\delta_1 \parallel \delta_2$,	concurrency with equal priority
$\delta_1 \rangle\!\rangle \delta_2$,	prioritized concurrency
$\delta^{\parallel}$,	concurrent iteration
$\langle\, \vec{x} : \phi \rightarrow \delta \,\rangle$,	interrupt
$p(\vec{\theta})$,	procedure call

Among these constructs, we notice the presence of nondeterministic constructs. These include $(\delta_1 \mid \delta_2)$, which nondeterministically chooses between programs δ_1 and δ_2, $\pi\, x.\, \delta$, which nondeterministically picks a binding for the variable x and performs the program δ for this binding of x, and δ^*, which performs δ zero or more times. Also notice that ConGolog includes constructs for dealing with concurrency. In particular $(\delta_1 \parallel \delta_2)$ expresses the concurrent execution (interpreted as interleaving) of the programs δ_1 and δ_2. Beside $(\delta_1 \parallel \delta_2)$, ConGolog includes other constructs for dealing with concurrency, such as prioritized concurrency $(\delta_1 \rangle\!\rangle \delta_2)$, where δ_1 runs at higher priority than δ_2, and interrupts $\langle\, \vec{x} : \phi \rightarrow \delta \,\rangle$. We refer the reader to [6] for a detailed account of ConGolog.

In [6], a single step transition semantics in the style of [21] is defined for ConGolog programs. Two special predicates *Trans* and *Final* are introduced. $\mathit{Trans}(\delta, s, \delta', s')$ means that by executing program δ starting in situation s, one can get to situation s' in one elementary step with the program δ' remaining to be executed. $\mathit{Final}(\delta, s)$ means that program δ may successfully terminate in situation s.

Offline executions of programs, which are the kind of executions originally proposed for Golog and ConGolog [16, 6], are characterized using the $\mathit{Do}(\delta, s, s')$ predicate, which means that there is an execution of program δ that starts in situation s and terminates in situation s' :

$$\mathit{Do}(\delta, s, s') \stackrel{\text{def}}{=} \exists \delta'.\mathit{Trans}^*(\delta, s, \delta', s') \wedge \mathit{Final}(\delta', s'),$$

where Trans^* is the reflexive, transitive closure of *Trans*. An offline execution of δ from s is a sequence of actions $a, \ldots, a_n$ such that : $\mathcal{D} \cup \mathcal{C} \models \mathit{Do}(\delta, s, do(a_n, \ldots, do(a_1, s) \ldots))$, where $\mathcal{D}$ is an action theory as mentioned above, and $\mathcal{C}$ is a set of axioms defining the predicates *Trans* and *Final* and the encoding of programs as first-order terms [6].

Golog and ConGolog programs were intended to be executed *offline*, that is, a complete execution was obtained before committing even to the first action. However, IndiGolog [7, 27], [1] the latest language within the Golog family, provides a formal logic-based account of interleaved planning, sensing, and action by executing programs *online* and using a specialized new construct $\Sigma(\delta)$, the *search operator*, to perform local offline planning when required.

Roughly speaking, an *online* execution of a program finds a next possible action, executes it in the real world, then obtains sensing information, and repeats the cycle until the program is completed. Formally, an online execution is a sequence of so-called online configurations of the form (δ, σ), where δ is a high-level program and σ is a history (see [7] for its formal definition). A history contains the sequence of actions executed so far as well as the sensing information obtained.

3 Generic Architecture

In this section, we describe the architecture we propose for combining JSA with APLs. There were two guiding principles we followed when designing our generic architecture for combining JSA with APLs. 1) We strove for modularity, i.e., we wanted to avoid the duplication of control structures and information storage in JSA and the APL. This facilitates the design and modification of a system by keeping information and control either in the JSA component or the APL component but not both. 2) We wanted to insulate the APL programmer as much as possible from having to deal with JSA programming. That is, we aimed to develop the JSA side

[1] http ://sourceforge.net/projects/indigolog/

of the system so that the APL programmer can avoid JSA programming altogether.

Generally, we kept the conversational aspects of the system on the JSA side and the information storage and planning aspects on the APL side. In particular, let the *objective belief base* of an agent be the set of beliefs the agent has about the world. These beliefs cannot contain belief or intention operators but may include (for ACLs that support them, such as IndiGolog) the temporal operators, i.e., *done* and *feasible*. The objective belief base is maintained on the APL side but is also used by JSA. However *subjective beliefs*, i.e., beliefs whose content contains beliefs and/or intentions are handled by JSA, since the APLs of which we are aware do not handle subjective beliefs. Planning and plan execution to bring about intentions are also handled on the APL side. The JSA component is used to handle communication with other agents. The default behavior of JSA is intended to be customized to suit the needs of a particular application using JSA and JADE. This customization includes what amounts to agent programming. Instead, we propose that the application-specific customization be done in the APL since APLs provide more sophisticated tools for programming agents.

To this end, we propose modifying JSA so that the objective beliefs of an agent that would be asserted by the JSA default behavior (e.g., as a result of an inform action) are passed on to the APL and stored in the belief base of the APL. If JSA has a query about an objective belief, the belief is passed on to the APL which answers the query if it can and the answer is passed back to JSA. This would be handled by adding an assert filter which checks if the asserted sentence is objective and if so it sends the sentence to be asserted in the APL belief base. The sentence is prevented from being asserted in the JSA belief base. Similarly, a query filter is put in place so that objective queries are sent to the APL and the answer is sent back to JSA. However, sentences that are not objective are still asserted and queried within JSA, such as beliefs about beliefs or beliefs about intentions.

The APL's intention base would be used to store the agent's intentions (using assert filters) and queries about the agent's intentions would be handled by the APL. The APL would be used for planning or using precompiled plans according to the capabilities of the APL.

The JSA process and the APL process would be running independently. However, the APL process must be responsive to the JSA process in order to handle assertions to the belief base and belief queries. The APL may also be called upon by the JSA process to plan or execute precompiled plans in order to generate actions to perform. The architecture is illustrated in Fig. 1.

The APL process might also want to send out messages on its own, i.e., independently of the JSA process interpreting received messages. This could be handled with a separate JADE or JSA process which takes as input messages to be sent by the APL, handles the translation into FIPA ACL, and sends the message to the appropriate agents. JSA has predefined methods that initiate communication (using FIPA ACL) with other agents that could be adapted for this purpose.

The architecture will have to be tailored to each APL since different APLs use different languages and have different capabilities. For example, IndiGolog has temporal operators and so can handle the *done* and *feasible* operators, but some BDI APLs do not have temporal operators. On the other hand, IndiGologdoes not have a built-in intention base nor does it have a built-in facility for updating beliefs about a particular fluent due to an exogenous action, so these would have to be added. Also, the form of the formulas that could be asserted in the belief base of both IndiGolog and BDI agents would have to be restricted to conjunctions of literals.

4 JSA+IndiGolog Architecture

In this section, we discuss how the generic architecture described above could be instantiated and implemented with the APL being IndiGolog. There are several issues that need to be addressed to obtain such an implemented architecture : how to handle 1) assertions of objective beliefs, 2) belief queries, 3) assertions of intentions, and 4) sending FIPA ACL messages from IndiGolog using JSA. Other issues, such as how to handle Call for Proposals performatives, intention and belief transfer, supporting queries from IndiGolog to the JSA belief base, etc., are left for future work.

As discussed above, objective beliefs will be stored in the IndiGolog belief base. A simple way to handle belief update is to introduce two special actions for each fluent F : $add_F(\vec{x})$, which makes the fluent $F(\vec{x})$ true, and $delete_F(\vec{x})$,

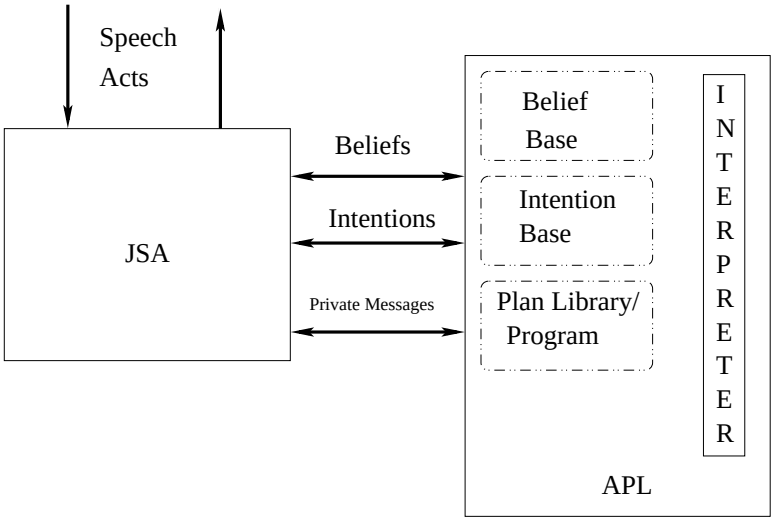

FIG. 1 – Diagram of JSA-APL architecture

which makes $F(\vec{x})$ false. We assume that the objective formulas to be asserted in the IndiGolog belief base are literals, however it is straightforward to generalize the account to handle conjunctions of literals. We also assume that the implementation of the generic architecture on the JSA side translates assertions of positive literals into add_F actions and assertions of negative literals into $delete_F$ actions. Then for each fluent F, we add the add_F and $delete_F$ actions to the successor state axiom for F. That is, if the successor state axiom for F is of the form :

$$F(\vec{x}, do(a, s)) \equiv \\ (\gamma^+(\vec{x}, s) \vee (F(\vec{x}, s) \wedge \neg\gamma^-(\vec{x}, s))),$$

we modify it as follows :

$$F(\vec{x}, do(a, s)) \equiv \\ [a = add_F(\vec{x}) \vee \gamma^+(\vec{x}, s) \vee \\ (F(\vec{x}, s) \wedge \neg(a = delete_F(\vec{x}) \vee \gamma^-(\vec{x}, s)))].$$

Note that this only performs a very simple form of belief update. In response to a belief being updated, one may want to revise related beliefs (e.g., to satisfy some state constraints), or hypothesize the occurrence of exogenous actions. This could be handled using programmer provided rules that can be triggered by belief updates. We leave the details for future work.

Handling belief base queries is a little more involved since it involves action on the part of the IndiGolog agent. When a query comes in, the agent checks its belief base to see if the query is true or false and replies accordingly. For simplicity, we assume here that the belief base is complete. But note there has been some work on allowing incomplete knowledge in IndiGolog [29], which can be used to relax this assumption.

Queries are implemented with the $query(\phi)$ exogenous action, which is executed by the JSA agent when it wants to know the value of the objective formula ϕ. The IndiGolog agent replies using the $reply(\phi)$ action, if ϕ currently holds, or $reply(\neg\phi)$, if $\neg\phi$ currently holds.

We introduce a new fluent $Queried(\phi)$ which holds, if ϕ has been queried but no reply has been issued yet :

$$Queried(\phi, do(a, s)) \equiv \\ (a = query(\phi) \vee \\ (Queried(\phi, s) \wedge a \neq reply(\phi) \wedge \\ a \neq reply(\neg\phi))).$$

Next we define a complex action which processes queries :

$$ProcessQueries \stackrel{\text{def}}{=} \\ \langle\, \phi : Queried(\phi) \rightarrow \textbf{if } \phi \textbf{ then } reply(\phi) \textbf{ else } \\ reply(\neg\phi) \textbf{ endIf} \,\rangle.$$

Now, given an IndiGolog program δ, we can enable query processing in δ by executing *ProcessQueries* concurrently with and at a higher priority than δ, i.e., $ProcessQueries \gg \delta$.

IndiGolog, unlike BDI languages, does not have a built-in intention base, and control in IndiGolog is driven by the agent's program as opposed to

its intentions. Thus, in general it would be up to the IndiGolog programmer to decide how intentions should be incorporated into the program.

We can implement an intention base in IndiGolog using a fluent, *Requested*(ϕ), where ϕ is a state formula representing an achievement goal (intentions to perform a complex action δ could be represented as *Done*(δ)). The exogenous action *request*(ϕ) adds ϕ to the intention base (i.e., causes *Requested*(ϕ) to hold), while the exogenous action *cancelReq*(ϕ) drops ϕ from the intention base.

The successor state axiom for *Requested* is as follows :

$$\textit{Requested}(\phi, \textit{do}(a, s)) \equiv (a = \textit{request}(\phi) \vee (\textit{Requested}(\phi, s) \wedge a \neq \textit{cancelReq}(\phi))).$$

The IndiGolog program can access the intention base (using the *Requested* fluent) in order to work towards achieving the intentions.

There will be cases where an IndiGolog agent wants to initiate FIPA ACL messages rather than just reacting to messages received from other agents. Since JSA already has a library for creating and sending FIPA ACL messages, we propose to use that rather than recreating it for IndiGolog (and for other ACLs using our generic architecture). To do this, we propose a special action *sendJSA*(*perf*, *list*), which takes as its first argument the performative of the message, and as its second argument, a list containing the remainder of the components of the message, which would include the recipient list for the message and whatever other components are appropriate for the given performative. The JSA agent would have a thread waiting for such a message, and upon receipt, it would create the appropriate FIPA ACL message for the performative and component list, and send the message to the recipients.

To implement the communication between JSA and IndiGolog, we would use TCP/IP sockets. The current IndiGolog implementation allows users to specify device managers, which listen on a specified socket for messages which can be custom translated into exogenous actions and then inserted into the history of actions. Similarly, device managers handle the implementation of physical actions, which would include messages sent from the IndiGolog agent to the JSA agent. It would not be difficult to implement a device manager to handle the communication between IndiGolog and JSA.

5 Discussion and Conclusion

In this paper, we outlined an approach to combining the sophisticated communication capabilities of JSA with the well-developed agent programming features of APLs. We believe the combination of these languages/tools can lead to a very flexible framework for implementing agent systems with complex behavior and communication capacities. First, we outlined a generic architecture that could be applicable to a variety of APLs, and then we discussed how the architecture could be implemented for IndiGolog.

As this work is still in its preliminary stages, there are many issues that remain to be resolved. One issue is where to handle the specification of when an agent will accept intention and belief transfers. In keeping with our strategy of isolating the APL programmer from having to customize JSA, we would suggest using special predicates on the APL side to specify under which conditions such transfers would be allowed. If the decisions about these transfers are static, i.e., the decisions could be made in advance, then this would be possible. However, if an agent is allowed to change its mind about whether to accept belief or intention transfers from another agent at run-time, then this would be difficult to implement on the APL side, since the beliefs about the beliefs and intentions of other agents (which would be needed to make these decisions) are stored on the JSA side and the APL does not have access to them. However, we note that APLs generally do not allow the representation of beliefs about the beliefs and intentions of other agents, so this would seem to be a reasonable restriction.

Another issue still to be resolved is how to accommodate Call for Proposal messages. A Call for Proposal is a FIPA ACL performative that is used to initiate negotiations between agents and is therefore somewhat more difficult to implement in an APL than simpler performatives such as Inform and Request (the recipient must decide what proposal to make, if any). We reserve the handling of this performative for future work. In future work, we also plan to implement the architecture for IndiGolog and a BDI APL, and evaluate the resulting platforms.

Références

[1] F.L. Bellifemine, G. Claire, and D. Greenwood. *Developing Multi-Agent Systems*

with JADE. Wiley, West Sussex, England, 2007.

[2] Rafael H. Bordini, Jomi Fred Hübner, and Michael Wooldridge. *Programming Multi-agent Systems in AgentSpeak Using Jason*. Wiley Series in Agent Technology. Wiley, West Sussex, England, 2007.

[3] R.H. Bordini, M. Dastani, J. Dix, and A. El Fallah-Seghrouchni, editors. *Multi-Agent Programming : Languages, Platforms and Applications*. Springer, New York, NY, 2005.

[4] Michael E. Bratman. *Intentions, Plans, and Practical Reason*. Harvard University Press, 1987.

[5] Paolo Busetta, Ralph Rönnquist, Andew Hodgson, and Andrew Lucas. JACK intelligent agents : Components for intelligent agents in Java. *AgentLink News*, 2 :2–5, January 1999.

[6] Giuseppe De Giacomo, Yves Lespérance, and Hector J. Levesque. ConGolog, a concurrent programming language based on the situation calculus. *Artificial Intelligence Journal*, 121(1–2) :109–169, 2000.

[7] Giuseppe De Giacomo and Hector J. Levesque. An incremental interpreter for high-level programs with sensing. In Hector J. Levesque and Fiora Pirri, editors, *Logical Foundations for Cognitive Agents : Contributions in Honor of Ray Reiter*, pages 86–102. Springer, Berlin, 1999.

[8] Daniel Dennett. *The Intentional Stance*. The MIT Press, 1987.

[9] The Foundation for Intelligent Physical Agents, an IEEE Computer Society Standards Committee. `www.fipa.org`.

[10] Michael P. Georgeff and Amy L. Lansky. Reactive reasoning and planning. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pages 677–682, Seattle, USA, 1987.

[11] The DARPA Knowledge Sharing Initiative External Interfaces Working Group. Specification of the KQML agent-communication language – plus example agent policies and architectures. Available at `http ://www.cs.umbc.edu/kqml/`.

[12] Alejandro Guerra-Hernández, Amal El Fallah-Seghrouchni, and Henry Soldano. Learning in BDI multi-agent systems. In Jürgen Dix and João Alexandre Leite, editors, *Computational Logic in Multi-Agent Systems, 4th International Workshop, CLIMA IV*, volume 3259 of *Lecture Notes in Artificial Intelligence (LNAI)*, pages 218–233. Springer, 2004.

[13] Koen V. Hindriks, Frank S. de Boer, Wiebe van der Hoek, and John-Jules Ch. Meyer. Agent programming in 3APL. *Autonomous Agents and Multi-Agent Systems*, 2 :357–401, 1999.

[14] Marcus J. Huber. JAM : A BDI-theoretic mobile agent architecture. In *Proceedings of the Annual Conference on Autonomous Agents (AGENTS)*, pages 236–243, New York, NY, USA, 1999. ACM Press.

[15] Samin Karim, Budhitama Subagdja, and Liz Sonenberg. Plans as products of learning. In *IEEE/ACM International Conference on Intelligent Agent Technology (IAT 2006)*, pages 139–145. IEEE Computer Society, 2006.

[16] Hector J. Levesque, Ray Reiter, Yves Lespérance, Fangzhen Lin, and Richard B. Scherl. GOLOG : A logic programming language for dynamic domains. *Journal of Logic Programming*, 31 :59–84, 1997.

[17] Vincent Louis and Thierry Martinez. The JADE semantic agent : towards agent communication oriented midleware. *AgentLink News*, 18 :16–18, August 2005.

[18] Vincent Louis and Thierry Martinez. Operational model for the FIPA-ACL semantics. In Frank Dignum, Rogier M. van Eijk, and Roberto Flores, editors, *Agent Communication II : International Workshops on Agent Communication, AC 2005 and AC 2006, Utrecht, Netherlands, July 25, 2005, and Hakodate, Japan, May 9, 2006, Selected and Revised Papers*, volume 3859 of *Lecture Notes in Artificial Intelligence (LNAI)*, pages 1–14. Springer-Verlag, 2006.

[19] Erick Martinez and Yves Lespérance. IG-JADE-PKSlib : an agent-based framework for advanced web service composition and provisioning. In *Proc. of the AAMAS 2004 Workshop on Web-services and Agent-based Engineering*, pages 2–10, New York, NY, USA, July 2004.

[20] John McCarthy and Patrick J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. *Machine Intelligence*, 4 :463–502, 1969.

[21] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI-FN-19, Computer Science Department, Aarhus University, Denmark, 1981.

[22] Martha E. Pollack. The uses of plans. *Artificial Intelligence Journal*, 57(1) :43–68, 1992.

[23] Anand S. Rao. AgentSpeak(L) : BDI agents speak out in a logical computable language. In W. Vander Velde and J. W. Perram, editors, *Proceedings of the Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World. (Agents Breaking Away)*, volume 1038 of *Lecture Notes in Computer Science (LNCS)*, pages 42–55. Springer-Verlag, 1996.

[24] Ray Reiter. The frame problem in the situation calculus : A simple solution (sometimes) and a completeness result for goal regression. In Vladimir Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computation : Papers in Honor of John McCarthy*, pages 359–380. Academic Press, San Diego, CA, 1991.

[25] Ray Reiter. *Knowledge in Action. Logical Foundations for Specifying and Implementing Dynamical Systems*. The MIT Press, 2001.

[26] D. Sadek and P. Bretier. ARTIMIS : Natural dialogue meets rational agency. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1030–1035, 1997.

[27] Sebastian Sardina, Giuseppe De Giacomo, Yves Lespérance, and Hector J. Levesque. On the semantics of deliberation in IndiGolog – from theory to implementation. *Annals of Mathematics and Artificial Intelligence*, 41(2–4) :259–299, August 2004.

[28] Sebastian Sardina and Lin Padgham. Goals in the context of BDI plan failure and planning. In Edmund H. Durfee, Makoto Yokoo, Michael N. Huhns, and Onn Shehory, editors, *Proceedings of Autonomous Agents and Multi-Agent Systems (AAMAS)*, pages 16–23, Hawaii, USA, May 2007. ACM Press.

[29] Stavros Vassos and Hector Levesque. Progression of situation calculus action theories with incomplete information. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 2024–2029, 2007.

[30] Michael Winikoff, Lin Padgham, James Harland, and John Thangarajah. Declarative & procedural goals in intelligent agent systems. In *Proceedings of Principles of Knowledge Representation and Reasoning (KR)*, pages 470–481, Toulouse, France, April 2002.