IG-JADE-PKSlib: An Agent-Based Framework for Advanced Web Service Composition and Provisioning

Erick Martínez
Department of Computer Science
York University
Toronto, Ontario, Canada M3J 1P3
erickm@cs.yorku.ca

Yves Lespérance
Department of Computer Science
York University
Toronto, Ontario, Canada M3J 1P3
lesperan@cs.yorku.ca

ABSTRACT

In this paper we describe an agent-based infrastructure and toolkit to develop inter-operable, intelligent, multi-agent systems for Web service composition (WSC) and provisioning. Our toolkit is realized through an interface library (IG-JADE-PKSlib) that combines state of the art agent-based and planning technologies (i.e., the IndiGolog model-based agent programming language, the JADE agent platform, and the PKS planning system). We show that each of these tools has its strengths and weaknesses, but combined together, they provide a very powerful toolkit. We argue that this infrastructure is particularly well suited for developing next generation Web services (WS) applications.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements / Specifications—*tools*; D.2.2 [**Software Engineering**]: Design Tools and Techniques—*software libraries*; I.2.11 [**Artificial Intelligence**]: Distributed Artificial Intelligence—*intelligent agents, multiagent systems*

Keywords

Multiagent systems, semantic web, web services

1. INTRODUCTION

The Internet is rapidly evolving from being just a repository of information into a vehicle for emerging Web services technologies. Web services (henceforth WS) are self-contained, self-described, active, and modular software applications that can be advertised, discovered, and invoked over the Web [11], e.g., an airline travel service or a book-buying service. Because of their autonomous and heterogeneous nature, WSs are naturally associated with agents. Agents are persistent computational entities (e.g., a software program or a robot) capable of perceiving and acting, in an autonomous manner, in their environment in order to meet their design objectives. They can also interact and communicate with other agents. Agents can incorporate reasoning techniques (e.g., planning, decision making, and learning) to achieve flexible and effective behaviour [22].

An interesting and demanding challenge is to generate composed services to meet new needs by assembling existing services, probably offered by different organizations. Thus, there is a demand for modelling techniques and tools for Web service composition (henceforth WSC). In this paper, we describe our efforts to develop next generation tools for WSC and provisioning. Our main motivation is to provide an agent-based infrastructure and toolkit to develop software agents capable of implementing reliable, large-scale inter-operation of WSs and support advanced features such as WSC, WS customization, execution monitoring and failure handling. Such agents should not only be general, easy to use and customizable, but reusable by different users under changing conditions.

The use of high-level model-based programming languages for the specification and execution of complex actions in dynamic domains has been addressed in [8, 6, 19]. IndiGolog [8] is part of the Golog-family[1] of high-level languages developed by the Cognitive Robotics Group at the University of Toronto and extended at York University. IndiGolog provides a practical framework for implementing autonomous agents as it supports plan execution, sensing, exogenous actions handling and planning in incompletely known dynamic environments. It also has some mechanisms to provide execution monitoring and re-planning. The latter capabilities are important for Web service enactment, as agents can keep track of how responsive particular services are and control how much is delegated to them. However, IndiGolog is mainly intended for designing individual autonomous agents. In [16], an extension of the Golog programming language is proposed to address the WSC problem. The designer provides high-level generic procedures for composed services and these are customized based on user constraints. This approach operates under the assumptions of reasonable persistence of sensed information and complete independence between sensing and world-altering actions. But there are scenarios where these assumptions do not hold.

[1]This software is freely available for non-commercial research purposes; for more details, visit http://www.cs.utoronto.ca/cogrobo/ and http://www.cs.yorku.ca/~lesperan/IndiGolog/.

Many complex applications are best delivered as multi-agent systems (MAS), that is, systems that involve several interacting, intelligent agents pursuing some set of goals [21]. Various software frameworks/platforms have been developed for constructing and delivering MAS. The Java Agent Development Framework (JADE) [2] is an open source, Java-based, FIPA[2]-compliant software framework for developing multi-agent applications. It was jointly developed by TILAB and AOT Labs. JADE facilitates interoperability with other agent applications/platforms that are compliant with the FIPA specifications. It also provides support for some types of ontologies. However, JADE does not provide reasoning mechanisms for agents. It does provide an interface to the JESS [10] expert system shell, but JESS has limitations with respect to reasoning about dynamic domains.

An approach to WSC using planning is presented in [4]. This preliminary work is based on combining a semantic data-type matching algorithm with another algorithm based on interleaved search and execution. The former facilitates the mapping of data between heterogeneous type structures, and the latter is used to overcome the problem of incomplete knowledge regarding the actions in the domain. WS procedures usually involve a lot of information gathering/sensing actions. If we are going to perform WSC as plan synthesis, we will need a planner that supports incomplete information and sensing actions. Some work has addressed the problem of planning under conditions of incomplete knowledge and sensing, for instance, the PKS planner [17] that uses a generalization of the STRIPS [9] approach, and the planner discussed in [3], that uses a model checking approach. One of the features that makes PKS interesting for WSC, is its ability to generate parameterized conditional plans that include sensing actions. In addition, PKS generates plans at the knowledge level without considering all the different ways the physical world can be configured. In [14], some preliminary empirical evidence to validate the effectiveness of using PKS to represent and solve WSC problems is presented.

In this paper we describe our work on an agent-based infrastructure and toolkit, for inter-operable, intelligent, multi-agent systems for Web service composition (WSC) and provisioning. Our toolkit is realized through an interface library *IG-JADE-PKSlib*, that combines state of the art agent-based and planning technologies, i.e., the IndiGolog agent programming language, the JADE agent platform, and the PKS planning system. Each of these tools has its strengths and weaknesses, but combined together, they provide a very powerful toolkit.

A related toolkit is *IG-OAAlib* [12] an interface library that supports the inclusion of IndiGolog agents in an Open Agent Architecture (OAA) system. OAA [13] is a framework developed at SRI International for integrating heterogeneous applications in a distributed infrastructure. However, as OAA provides no built-in support for inter-operation with other agent platforms, nor for standard communication languages and ontologies; it has limitations as an infrastructure for developing WS applications.

The rest of the paper is organized as follows. In Sections 2, 3, and 4 we review the basic components of our toolkit, i.e., IndiGolog, JADE and PKS respectively. Details of the IndiGolog-JADE interface (*IG-JADElib*) are presented in Section 5, Next, in Section 6, we discuss a high-level approach to supporting FIPA-compliant agent interaction protocols in IndiGolog agents. We illustrate this by showing the library implementation of the participant role (CN-participant) of the Contract Net Protocol (CNP). Then, in Section 7 we present an approach to adding PKS support to *IG-JADElib* and describe how it is implemented. A WS travel planning example is examined in section 8. In particular, we sketch how we build and deploy this WS application as a multi-agent system using the *IG-JADE-PKSlib*. Finally, in Section 9 we conclude and discuss future work.

[2]FIPA stands for Foundation for Intelligent Physical Agents.

2. INDIGOLOG

IndiGolog [8] is a high-level model-based programming language based on the situation calculus [15] a predicate logic language for representing dynamic domains. IndiGolog supports plan execution, sensing, exogenous events handling, planning in incompletely known dynamic environments, execution monitoring, and re-planning. In the situation calculus, the constant S_0 denotes the initial situation and the binary function symbol $do(a, s)$ denotes the situation resulting from action a being performed in situation s. Relations and functions whose values vary from situation to situation are called *relational fluents* and *functional fluents* respectively. These *fluents* are denoted by predicate and functional symbols that take a situation term s as last argument.

An IndiGolog agent specification has two parts: a declarative specification of the domain and its dynamics in the situation calculus, and a procedural description of the agent behaviour in the IndiGolog process language. The domain dynamics specification is a situation calculus theory that includes the following types of axioms:

- Axioms describing the initial situation, S_0
- Action precondition axioms, one for each primitive action a, characterizing $Poss(a, s)$, i.e., when primitive action a is possible in situation s.
- Successor state axioms, one for each fluent F, stating the conditions under which $F(x, do(a, s))$ holds in the situation $do(a, s)$ in terms of what holds in situation s; these axioms provide a solution to the frame problem [19].
- Sensed fluent axioms, one for each primitive action a, relating the value returned by a sensing action to the fluent condition it senses in the environment.
- Unique names axioms for the primitive actions.
- Some foundational, domain independent axioms.

The behaviour of an IndiGolog agent is specified procedurally using the following set of high-level programming constructs:

α,	primitive action
$\phi?$,	wait for a condition
$\delta_1;\delta_2$,	sequence
$\delta_1 \mid \delta_2$,	non-deterministic branch
$\pi\, x.\delta$,	non-deterministic choice of arguments
δ^*,	non-deterministic iteration
if ϕ **then** δ_1 **else** δ_2 **endIf**,	conditional
while ϕ **do** δ **endWhile**,	while loop
$\delta_1 \mid\mid \delta_2$,	concurrent execution
$\delta_1 \rangle\rangle\ \delta_2$,	prioritized concurrency
$\delta^{\mid\mid}$,	concurrent iteration
$< x : \phi \rightarrow \delta >$,	interrupt
proc $p(\vec{\theta})\ \delta$ **endProc**,	procedure definition
$p(\vec{\theta})$,	procedure call
$\sum(\delta)$,	search operator
noOp	do nothing

Note the presence of several non-deterministic constructs: $\delta_1 \mid \delta_2$, which non-deterministically chooses between programs δ_1 and δ_2; $\pi\, x.\delta$, which non-deterministically picks a binding for the variable x and performs the program δ for this binding of x; and δ^*, which performs program δ zero or more times. IndiGolog also incorporates a rich account of (interleaved) concurrency. In particular, construct $\delta_1 \mid\mid \delta_2$ captures the concurrent execution of the programs δ_1 and δ_2. Additionally, construct $\delta_1 \rangle\rangle\ \delta_2$ is used to express prioritized concurrency, i.e., δ_2 executes only when δ_1 is completed or blocked. An interrupt $< x : \phi \rightarrow \delta >$ has a trigger condition ϕ and a body δ. If the interrupt gets control from a higher priority process and the condition ϕ holds, then the interrupt triggers and its body δ is executed. Once δ completes execution, the interrupt may trigger again. Finally, construct $\sum(\delta)$ does a lookahead search over the non-deterministic program δ, to find a way to successfully execute it – this can be used to do planning. IndiGolog has a formal semantics based on single steps of computation or transitions [7].

3. JADE

JADE [2] is a software framework and middle-ware enabling technology for developing multi-agent systems. As a FIPA-compliant agent platform, JADE provides an implementation for the following components:

- **Agent Management System (AMS).** This agent is responsible for controlling access to the platform, authentication and registration of participating agents.
- **Directory Facilitator (DF).** This agent provides a yellow page service to the agents in the platform.
- **Agent Communication Channel (ACC).** This agent provides a white page service. It also supports inter-agent communication and inter-operability within and across different platforms.

JADE also provides an implementation of the full FIPA communication model, i.e., agent interaction protocols (AIPs), FIPA Agent Communication Language (ACL), SL content language, ontology support and, transport protocols. In JADE, agent communication involves the exchange of ACL messages between agents. ACL has a formally defined semantics and pragmatics which avoid any ambiguity derived from the usage of the language. ACL messages within the same platform are exchanged encoded as Java objects. However, when the communicating agents belong to different platforms the ACL messages are converted from/into string format. AIPs describe communication patterns as legal sequences of messages exchanged between agents. Many common AIPs are supported in JADE.

Agent communication languages such as FIPA ACL distinguish between content languages, and ontologies. The former, describe how the content of a message is encoded to be transmitted between two agents. Thus, content languages are typically domain independent. The latter, describe the vocabulary used and some of its semantics. By default, JADE provides support for three content languages: (i) LEAP, i.e., a lightweight binary representation, well-suited for embedded applications; (ii) FIPA SL, i.e, a rich (human-readable) text encoding for messages; and (iii) Java codec, i.e., an efficient Java encoding for exchanging messages within the same platform. Additionally, JADE supports user-defined content languages based on external ontologies. In particular, it provides generic XML/RDF codecs for ACL message content based on XML/XML Schema and RDF/RDF Schema. JADE also provides some built-in support for some FIPA-based ontologies. At the time of writing this paper, other wide-spread ontology languages such as DAML-S and OWL were not yet supported.

4. PKS

The PKS planning system[3] [1, 17, 18] supports the generation of conditional plans with sensing actions for problems involving incomplete knowledge. It is based on a generalization of STRIPS [9]. In STRIPS, the state of the world is represented by a database and actions are represented as updates to that database.

Databases. In PKS, there are four databases, each one storing a different type of knowledge. The contents of these databases have a fixed formal interpretation in a first-order modal logic of knowledge that characterizes the agent's knowledge state (KB).

K_f : This database stores knowledge of positive and negative atomic facts and knowledge of the value of functions on fixed arguments, e.g., $p(a, b),\ \neg q(c),\ f(a) = b,\ f(b) \neq c$. The closed world assumption does not apply.

K_w : This database is used for plan time modelling of the effects of sensing actions. Intuitively, $\phi \in K_w$ means that at planning time either the agent knows ϕ or it knows $\neg\phi$, where ϕ can be a conjunction of atomic formulas. The agent will only resolve this disjunction at execution time.

K_v : This database K_v is used for plan time modelling of the effect of sensing actions that return numeric values. K_v stores unnested function terms whose values will be known to the agent at execution time.

K_x : This database contains information about disjunctive (exclusive *or*) knowledge of ground literals of the form $(l_1|l_2|...|l_n)$. Intuitively, this formula represents the fact that the agent knows that exactly one of the l_i is true.

Goals. Simple goals can be represented as primitive queries. A primitive query can take one of the following forms: (i) $K(\alpha)$, is α known to be true? (ii) $K(\neg\alpha)$, is α known to

[3]This software is also freely available for non-commercial research purposes; for more details, visit http://www.cs.utoronto.ca/˜rpetrick/research/pks/.

Action	Precondition	Effects
$sFSpace(n, d)$	$K(existsF(n, d))$	$add(K_w, availF(n, d))$
$bookF(n, d)$	$K(availF(n, d))$	$add(K_f, bookedF(n, d))$ $del(K_f, availF(n, d))$

Table 1: *sFSpace* and *bookF* actions.

be false? (iii) $K_w(\alpha)$, does the agent know whether α? (iv) $K_v(t)$, does the agent know the value of t? (v) the negation of any of the previous queries. In the above, α represents any ground atomic formula, and t represents any variable free term. Complex goals can be expressed as queries which include primitive queries, conjunctions of queries, disjunctions of queries, and quantified queries where the quantification ranges over the set of known objects.

Actions. Actions are specified in terms of three components: parameters, preconditions, and effects. In Table 1, the specifications for actions $sFSpace(n, d)$ and $bookF(n, d)$ are given. The former is a knowledge-producing action that senses for seat availability on flight number n on date d. The effect of $sFSpace(n, d)$ is modelled by adding a new literal $availF(n, d)$ to the K_w database, i.e., the agent will know whether there are seats available on the flight. The latter is a physical action to book a flight. The agent can only book flights known to be available. Actions' effects are specified as a set of database updates, some of which can be conditional.

Domain specific update rules (DSUR). These rules are used to specify additional action effects and correspond to state invariants at the knowledge level, e.g.,

$$K(\neg existsF(n, d)) \Rightarrow add(K_f, \neg availF(n, d))$$

captures some additional effects of sensing for flight n on date d, if the agent finds there is no such flight, then it follows the flight is not available. DSURs may be triggered in any KB provided their conditions are satisfied.

Planning problems. A planning problem in PKS is defined as a tuple $\langle I, A, U, G\rangle$, where I is the initial state, A is a nonempty set of action specifications, U is a set of DSURs, and G is a goal condition.

Note that the only forms of incomplete knowledge that can be expressed are complete lack of knowledge about an atom, by leaving it out of K_f, and knowledge that only one of a finite set of literals is true using K_x (as well as information about what will be known at run time as supported by K_w and K_v). There is no reasoning by cases other than by going through a set of cases that have been explicitly enumerated. The PKS system relies on an efficient, but incomplete, inference algorithm that uses a forward chaining approach to find plans [1]. This algorithm examines the KB's contents to evaluate preconditions and goals.

5. INTERFACING INDIGOLOG AND JADE

One motivation in our work is to enhance IndiGolog agents' capabilities to participate in multi-agent systems (MAS). As well, we want to provide a tool for building reasoning agents that is JADE compatible. Thus, our IndiGolog-JADE (IG-JADE) interface adds value to both tools by combining their strengths. This naturally raises the issue of how to design, implement and deploy MAS involving heterogeneous platforms, i.e., rule-based and object-oriented.

One important technical issue that has to be addressed is the communication scheme of the participating agents / platforms. At a lower level of abstraction this involves routing the communication flows between platforms, and establishing what kind of communication protocols are supported by each platform. Our current low level implementation uses TCP/IP sockets. At a higher level, our interface should also address other issues like the structure of the messages, the use of standard communications languages (e.g., FIPA ACL) vs. application specific languages, and similarly with respect to the use of content languages (e.g., FIPA SL).

To interface an IndiGolog agent to JADE, one must relate the messages it receives and sends to its internal representation/domain theory. Incoming messages can be treated as exogenous actions and outgoing messages can be viewed as primitive actions (or ways of implementing primitive actions). In principle, our approach is based on incorporating a JADE peer agent for each IndiGolog agent. We support two approaches for interfacing IndiGolog agents to JADE in our toolkit. The first approach is perhaps the most portable and flexible, and it involves using a generic JADE agent that can act as a messenger for its IndiGolog peer and handles FIPA ACL (and eventually FIPA SL) messages. The IndiGolog agents can process ACL messages using Prolog library functions that we will describe in Section 5.1. So in the first approach we exploit Prolog's strengths for building language processing software. In the second approach, the implementer of the application creates an instance of the generic JADE messenger agent, that performs specific actions as response to the reception of different types of messages. This agent translates ACL messages into domain specific exogenous actions for the corresponding IndiGolog agent, and similarly it translates primitive actions into ACL messages. An instance of the messenger agent is created for a specific IndiGolog agent. Basically, they communicate using a private, application specific content language that only handles the relevant interactions in terms of defined primitive and exogenous actions. So in the second approach, we use JADE's full library support for dealing with FIPA ACL and SL.

5.1 IndiGolog Interface Details

One way to make IndiGolog agents portable as part of cross-platforms MAS is to enhance them with mechanisms to support agent communication languages. To this end, our current implementation provides flexible functionalities for monitoring, processing, storing and retrieving FIPA ACL messages as part of complex communication flows. Figure 1 illustrates the main components of the IndiGolog interface (all our contributions are depicted using dotted lines). Note that incoming messages correspond to exogenous events and outgoing messages correspond to actions.

- **Message Processor.** This module handles all incoming and outgoing communication, in particular, syntactic and semantic processing of messages. The Message Processor module has been implemented using a Prolog definite clause grammar (DCG) that covers a large portion of FIPA ACL and it is realized through the following three predicates:
  (i) $tokenize\_acl(+Stream, -TokList)$, i.e., produces a

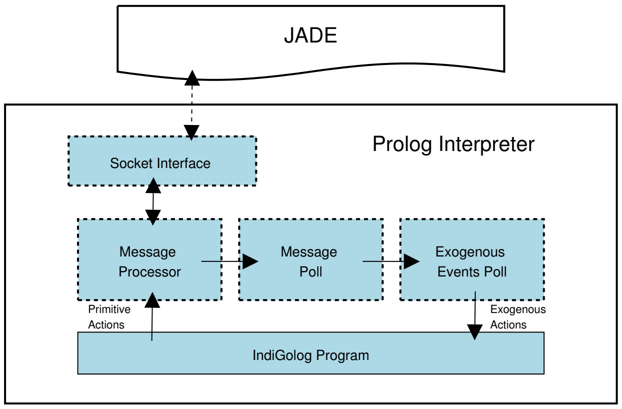

Figure 1: IndiGolog single agent interface high-level view

tokenized list $TokList$ from the incoming $Stream$;
(ii) $analyze\_acl(-SemContent, +TokList, [\,])$, i.e., first do syntactic analysis to check whether a given list of tokens $TokList$ is legal, and then if desired, do semantic analysis (which can be done in the usual DCG way by introducing additional predicates in the grammar); $SemContent$ is instantiated to the resulting Prolog term or to *nil* if the message does not conform to the specification;
(iii) $generate\_acl(+Envelope, +ContExpr, -Msg)$, i.e., generates an ACL-compliant message Msg from the specified Prolog term $Envelope$ and the content expression $ContExpr$. Msg is instantiated to a list which is a string representation of an ACL message.

- **Message Poll.** This module stores all messages received by the IndiGolog agent and provides functionality for adding and retracting messages.

- **Exogenous Actions Poll.** This module provides a mechanism for monitoring, storing and retrieving exogenous actions that represent incoming messages. It is monitored by the IndiGolog interpreter which inserts new exogenous actions in the agent's situation, updating its KB. This can then influence the behaviour of the agent.

We think that this generic interface provides some basic communication functionalities on the IndiGolog side for communicating with other agent platforms. The programmer should decide what kind of communication language is best suited for each particular application. In principle, we think that a standard communication language (e.g., FIPA ACL) should be used for application domains where complex agent conversation flows are expected. Otherwise, a simpler application specific language can be used. The current implementation provides support for a large portion of FIPA ACL as well as for a limited subset of the FIPA SL content language standard. In particular, all the performatives of the FIPA interaction protocols are supported as part of the content language. We also intend to explore how to handle the mapping between ontological representations encoded in content languages and action theories.

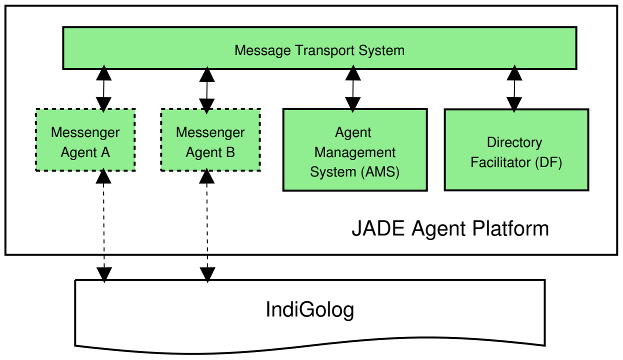

Figure 2: JADE interface high-level view

5.2 JADE Interface Details

Figure 2 illustrates the main components of the JADE interface (our contributions are depicted using dotted lines). The messenger agent is socket-based, and provides client and server functionalities to handle incoming and outgoing communication from/to other JADE agents. Basically, this agent maintains two queues of messages. As can be expected, the incoming queue corresponds to exogenous actions on the IndiGolog side, and similarly, the outgoing queue corresponds to primitive actions.

The *IG-JADElib* is still a work-in-progress, and at present a proof of concept. It would be interesting to explore a more robust approach to low level communication, perhaps using the Java Messaging System (JMS). JMS [20] defines a generic Java language interface to message services and supports most common messaging models (e.g., point-to-point and publish/subscribe). JMS seems well-suited for heterogeneous application integration because it is an industry-supported interface specification – not an implementation. However, it remains to be seen whether this approach can be effectively applied to Prolog-based applications.

6. FIPA PROTOCOLS IN INDIGOLOG

6.1 Overview

Another way to enhance the IndiGolog agent capabilities to participate in MAS is to incorporate support for complex agent interaction protocols (AIP). In this section we use an example, the Contract Net Protocol (CNP), to illustrate a high-level approach to adding AIP roles to IndiGolog agents. In particular, we demonstrate how some built-in IndiGolog mechanisms, i.e., concurrency with prioritized interrupts; can be gracefully adapted for this task. Our approach provides a way to integrate (as part of the agent's behaviour) the typical reactive nature of some interaction protocol roles, without compromising the proactive capabilities of the agent. That is, we want our IndiGolog agent to be capable of participating in complex interactions, while still working to satisfy their objectives. We plan to support all FIPA protocols. Note that all FIPA protocols have a formally defined semantics.

6.2 An Example: FIPA Contract Net Participant Role

This role is encoded in (1) as a generic reactive behaviour to three kinds of exogenous events:

(i) *requestForBids*$(Cno, Task)$, i.e., a new request for bids; (ii) *acceptProposal*$(Aid, Cno, Task)$, i.e., the agent's proposal was accepted by the CN-initiator; and (iii) *rejectProposal*$(Aid, Cno, Task)$, i.e., the proposal was rejected by the CN-initiator.

```
proc $mainCNParticipant(Aid)$
% initialize bidding process
  $< cn, t :$
    $bid\_requested(cn, t) = true$
      $\rightarrow initCNParticipant(cn, t) >$
% inform task results
  $\rangle\rangle < cn, t :$
    $(task\_status(cn, t) = completed \vee$
     $task\_status(cn, t) = aborted)$
      $\rightarrow informTaskResults(cn) >$                    (1)
% if proposal rejected reset fluents
  $\rangle\rangle < aid, cn, t :$
    $(initiator\_responded(aid, cn) = true \wedge$
    $(proposal\_accepted(aid, cn, t) = false)$
      $\rightarrow handleRejection(cn) >$
% if nothing to do, wait
  $\rangle\rangle < true \rightarrow$ noOp $>$
endProc
```

The top priority interrupt in (1) takes care of initiating the CN-participant behaviour whenever a new bid requests arrives. This is handled by executing the following procedure:

```
proc $initCNParticipant(Cno, Task)$
% non-deterministic pick of agent ID
  $\pi$ $aid$ [
    $(my\_bidding\_ID = aid)?$;
% acknowledge not understood proposal
    if $\neg understoodCall(Cno, Task)$ then
      $sendNotUnderstoodMsg(aid, Cno, Task)$
% handle bidding process                                  (2)
    else
      $ack(aid, Cno, Task)$;
      $decideBid(Cno, Task)$;
      $(bidDecisionMade(Cno, Task))?$;
      $handleBidding(Cno, Task)$;
    endIf ]
endProc
```

This acknowledges the receipt of the request for proposals, whether it was understood or not and if so, tries to make a bid decision and then handle the bidding. Note that *decideBid*$(Cno, Task)$ and *handleBidding*$(Cno, Task)$ are domain specific procedures that should be provided for each application. The second highest priority interrupt in (1) takes care of detecting task completion or task failure and informing the contractor of the results. This makes the agent highly reactive not only to new incoming requests, but also guarantees that the CN-initiator always gets rapid feedback. The next interrupt handles the case where the proposal is rejected by the CN-initiator. The *handleRejection*(cn) procedure is provided to handle situations where the agent has to reallocate some resources. This procedure should be defined by the implementer, but often it will do nothing.

Accepted proposals must be handled by the agent, as part of its overall objective-achieving behaviour, at a time of the agent's choosing. This not part of the CN-participant role procedure, but is handled in a domain specific way. Usually, the task solver behaviour will run concurrently with the CN-participant role and perhaps other behaviours. For example, we might have:

```
proc $mainControl(Aid)$
% concurrent exec. task solver and AIP role
  $taskSolver \parallel mainCNParticipant(Aid)$;          (3)
endProc
```

A simple example of a task solver procedure goes as follows [5]:

```
proc $taskSolver(Cno, Task)$
% if contract granted call task solver
  $< new\_contract\_granted = true$
      $\rightarrow resetNewContractFluent$;
        [...implementation...]                            (4)
% if nothing to do, wait
  $\rangle\rangle < true \rightarrow$ noOp $>$
endProc
```

Note that the *new_contract_granted* fluent is associated to the *acceptProposal*$(Aid, Cno, Task)$ exogenous event. Therefore, whenever a new contract is granted the interrupt is triggered and the task solver takes over the task.

IndiGolog supports planning for finding ways of achieving tasks or scheduling their achievement. A new contract being granted could trigger re-planning. In this way, the agent can deal with incoming contracts "on the fly" and optimize their scheduling. Also note that we can handle multiple auctions and contracts concurrently as we use the contract number *cn* to distinguish between them.

7. ADDING PKS-BASED CAPABILITIES

Another motivation in our work is to develop software agents capable of implementing reliable, large-scale inter-operation of Web services, Web service composition, customization, etc. Such agents should be general, easy to use and reusable by different users under changing conditions. In [14], some preliminary empirical evidence to validate the effectiveness of using PKS to represent/solve WSC problems is presented. Even if in terms of the overall scalability more experiments are still required, some early results are very promising. In particular, PKS provides an interesting alternative to the computational drawbacks of possible worlds reasoning and also supports a high-level specification for automated WSC. A sample goal, book a flight with a price equal or less than the user's maximum price, is illustrated in (5):

$$\begin{array}{l} \%\ \textit{book company within budget} \\ \exists_k(x)[K(airCo(x)) \wedge K(bookedFlight(x)) \wedge \\ \qquad K(\neg priceGtMax(x))] \quad | \\ \%\ \textit{no flight booked} \\ KnowNoBudgetFlight \quad | \\ KnowNoAvailFlight \quad | \\ KnowNoFlightExists \end{array} \tag{5}$$

Note that quantifiers are restricted to range over the set of known objects / constants in the domain. For convenience, we introduced three abbreviations: $KnowNoBudgetFlight$, which represents the case where there is at least one available flight but none within the budget; $KnowNoAvailFlight$, which encodes the case where there is no seat available; and $KnowNoFlightExists$, which represents the case where there is no flight. PKS will generate a plan that satisfies user customization constraints. The domain specification should be represented as described in Section 4. In [14] we show how a range of WSC problems, like the above example, can be represented and solved in PKS.

We considered several alternatives for incorporating knowledge-based planning capabilities into our infrastructure. We could either directly interface JADE and PKS, or IndiGolog and PKS, or both. In the end, we chose the former approach (i.e., JADE-PKS) as PKS is implemented in C++ and it seemed simpler and more natural to interface two object-oriented systems by just assembling their corresponding APIs. Our implementation is realized through a full-fledged JADE agent which acts as a wrapper for the PKS planning system. As a result, the wrapper agent implicitly benefits from all the functionality available within the JADE platform, and can also interact with IndiGolog agents connected to JADE using the IG-JADE interface. In particular, the wrapper agent can participate in complex conversation flows as part of standard interaction protocols. In addition, our architecture can also eventually accommodate the implementation of different schemes to map a given ontology into a PKS domain specification. The PKS-wrapper agent has two default behaviours, i.e., a problem solver behaviour which encapsulates all the knowledge-based planning functionality, and a request-responder behaviour which corresponds to the responder role in the FIPA-Request interaction protocol.

To incorporate PKS-based capabilities into any multi-agent application deployed using the *IG-JADE-PKSlib* toolkit, the implementer should create an instance of the PKS-wrapper agent. This agent will automatically register its capabilities with the JADE Directory Facilitator (DF). In particular, it will register the content language (i.e, FIPA SL) and ontology (i.e., `pks-ontology`) that it can understand, as well as the interaction protocols (i.e., FIPA-Request) that it is able to participate in. The agent finishes up the initialization stage by starting up its request-responder behaviour and then it is ready to process any valid incoming request.

To communicate and interact with the PKS agent, any requester agent must use a matching content language, ontology, and protocol. We provide the `pks-ontology` which defines the basic vocabulary and structure (schema) of concepts, actions and predicates relevant to the PKS planning system domain. Some relevant terms of the `pks-ontology` are presented in Table 2. For example, the `Search` concept describes the details of the type of search used by the PKS agent to find a solution. This concept is defined by two slots: (i) search-type, i.e., the search type can be one of depth-first search, breadth-first search or iterating deepening search; and (ii) `seach-cutoff-depth`, i.e., the cut-off depth value for terminating the search. On the other hand, action `Solve` describes the PKS agent action of solving a given problem under certain conditions. This action is defined by four slots: (i) `solve-problem`, i.e., the problem to be solved; (ii) `solve-search`, i.e., the search type details; (iii) `solve-output`, i.e., the output details; and (iv) `solve-wait-at-most`, i.e., the maximum amount of time (in milliseconds) the agent should spend trying to find a solution. Note that some terms have optional slots. All the mandatory slots are marked with an asterisk. Also note that, the input problem must be specified in a file using PKS-compliant XML syntax.

Term	Type	Slots
`Search`	concept	`search-type`*: {"dfs","bds","ids"} `seach-cutoff-depth`: ≪ number ≫
`Output`	concept	`output-format`*: {"tree","prolog"}
`File`	concept	`file-path`*: ≪ string ≫
`Problem`	concept	`prob-input-file`*: ≪ `File` ≫ `prob-name`: ≪ string ≫
`Solve`	action	`solve-problem`*: ≪ `Problem` ≫ `solve-search`: ≪ `Search` ≫ `solve-output`: ≪ `Output` ≫ `solve-wait-at-most`*: ≪millisecs≫

Table 2: PKS Ontology.

In the future, it would also be interesting to interface the PKS planning system directly to the IndiGolog interpreter, to provide extended capabilities for planning with incomplete knowledge and sensing actions. There are two ways this can be achieved. The first way is to have the interpreter pass planning calls to PKS, and wait for it to find solutions. This approach is efficient but not very flexible. The underlying Prolog implementation is platform-dependent, and there is no standard handling of low level communications. The second way is to extend the IndiGolog interpreter by incorporating a Prolog-based re-implementation of the PKS planning system. Although this approach is more flexible, it is also less efficient. In both cases, IndiGolog agents would benefit from having built-in knowledge-based planning capabilities. Also, the representation would be more compact as we only deal with standard action theories to specify our agents.

8. A WS TRAVEL DOMAIN EXAMPLE

We have seen in the previous section how one may want to perform WSC to solve problems like booking a flight between two cities that satisfy the constraints of a particular user. In this section we describe a system that is being implemented using the *IG-JADE-PKSlib* toolkit that performs WSC. A high-level view of an implementation of the system

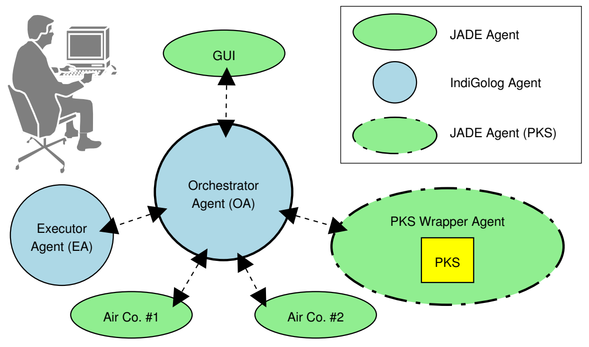

Figure 3: Air travel demo high-level view

is depicted in Figure 3. In this deployment scenario, there is an IndiGolog orchestrator agent (OA) capable of making air travel arrangements. For simplicity, we assume that this agent knows the set of air companies one can choose from. Each air company is represented by a JADE agent, which exposes all the different WSs available for that company (e.g., find a flight, check seat availability on the flight, check flight cost, and book the flight). When the user requests travel arrangements, the OA requests from the air company agents a functional description of the available atomic WSs, in terms of pre-conditions and effects. We assume they are represented using some given service ontology. We discuss this in the conclusion. Then, it generates an appropriate PKS domain specification in the form of an XML file and sends it as a request to a JADE PKS-wrapper agent. If the PKS-wrapper agent responds to the request and is able to generate a plan to make travel arrangements, then the OA passes it over as a request to an IndiGolog executor agent. Finally, if the executor honours the request and is able to complete the task successfully, the OA marks the task as completed and informs the user's GUI agent of the results. Note that the OA can at any time abort the task if cannot be completed, and will also inform the GUI agent about this.

Also note that the kind of protocol described in Section 6 (e.g., Contract Net) would also come handy if one wanted to script an agent that performs other tasks such as booking a rent-car or a hotel room. These are available in IndiGolog as well as JADE.

9. CONCLUSION AND FUTURE WORK

In this paper we described ongoing work on an agent-based infrastructure and toolkit, for developing inter-operable, intelligent MASs. We presented a high-level approach to adding AIP roles to IndiGolog agents, as well as supporting the FIPA ACL communication language. We also showed that adding PKS capabilities to our infrastructure provides a powerful planning mechanism for applications involving information gathering actions such as WSs. Our toolkit has been designed to accommodate arbitrary MASs, but we claim that it is particularly well suited for developing advanced WS applications. Our approach so far has been pragmatic, focusing on what functionalities are most convenient and useful for an application designer.

One important issue that arises from this work is the question of how to provide mechanisms to translate standard ontology description languages used in WS applications (e.g., OWL, DAML-S, etc.) into the type of action theories and process specifications that IndiGolog and PKS use. Clearly, more work is required to understand all the requirements which should be supported by the library. We also intend to perform some experiments and case studies and validate the performance and scalability of the integrated framework. The first release of the *IG-JADE-PKSlib* library will be soon available at http://www.cs.yorku.ca/~erickm/ig-jade-pks/.

ACKNOWLEDGEMENTS

Ksenia Choubina and Stan Zhiyan Li have made significant contributions to the IndiGolog-JADE interface. We thank the anonymous reviewers for their useful comments.

10. REFERENCES

[1] F. Bacchus and R. Petrick. Modeling an agent's incomplete knowledge during planning and execution. In *Proceedings of the Sixth International Conference on Principles of Knowledge Representation and Reasoning (KR-1998)*, pages 432–443, San Francisco, CA, June 1998. Morgan Kaufmann Publishers.

[2] F. Bellifemine, A. Poggi, and G. Rimassa. Jade: A FIPA-compliant agent framework. In *Proceedings of PAAM-1999*, pages 97–108, April 1999.

[3] P. Bertoli, A. Cimatti, M. Roveri, and P. Traverso. Planning in non-deterministic domains under partial observability via symbolic model checking. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-2001)*, pages 473–478, 2001.

[4] M. Carman, L. Serafini, and P. Traverso. Web service composition as planning. In *Proceedings of the ICAPS-2003 Workshop on Planning for Web Services*, Trento, Italy, June 2003. Università di Trento.

[5] K. Choubina. Interfacing IndiGolog and Jade - contract net protocol implementation. Unpublished manuscript, Department of Computer Science, York University, Apr. 2003.

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

[7] G. De Giacomo, Y. Lespérance, H. J. Levesque, and S. S. na. On the semantics of deliberation in IndiGolog: From theory to implementation. In *Proceedings of the Eighth International Conference on Knowledge Representation and Reasoning (KR-2002)*, pages 603–614, April 2002.

[8] G. De Giacomo and H. J. Levesque. An incremental interpreter for high-level programs with sensing. In H. J. Levesque and F. Pirri, editors, *Logical Foundations for Cognitive Agents*, pages 86–102. Springer-Verlag, 1999.

[9] R. E. Fikes and N. J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.

[10] E. Friedman-Hill. Jess, the expert system shell for the java platform. Documentation manual, Sandia National Laboratories, http://herzberg.ca.sandia.gov/jess/docs/manual.pdf, Mar. 2003.

[11] K. Gottschalk and IBM Team. Web services architecture overview: The next stage of evolution for e-business. Article, IBM, http://www-106.ibm.com/developerworks/web/library/w-ovr/, Sept. 2000.

[12] A. Lapouchnian and Y. Lespérance. Interfacing IndiGolog and OAA: A toolkit for advanced multiagent applications. *Applied Artificial Intelligence*, 16:813–829, 2002.

[13] D. L. Martin, A. J. Cheyer, and D. B. Moran. The open agent architecture: A framework for building distributed software systems. *Applied Artificial Intelligence*, 13:91–128, 1999.

[14] E. Martínez and Y. Lespérance. Web service composition as a planning task: Experiments using knowledge-based planning. In *Proceedings of the ICAPS-2004 Workshop on Planning and Scheduling for Web and Grid Services*, Whistler, BC, Canada, June 2004. To appear.

[15] J. McCarthy and P. C. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence*, volume 4, pages 463–502. Edinburgh University Press, 1979.

[16] S. McIlraith and T. C. Son. Adapting Golog for composition of semantic web services. In *Proceedings of the Eighth International Conference on Knowledge Representation and Reasoning (KR-2002)*, pages 482–493, Apr. 2002.

[17] R. P. A. Petrick and F. Bacchus. A knowledge-based approach to planning with incomplete information and sensing. In *Proceedings of the Sixth International Conference on Artificial Intelligence Planning and Scheduling (AIPS-2002)*, pages 212–221, Menlo Park, CA, Apr. 2002. AAAI Press.

[18] R. P. A. Petrick and F. Bacchus. Reasoning with conditional plans in the presence of incomplete knowledge. In *Proceedings of the ICAPS-03 Workshop on Planning under Uncertainty and Incomplete Information*, pages 96–102, Trento, Italy, June 2003. Università di Trento.

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

[20] Sun Microsystems. Java message service specification - version 1.1. Documentation, Sun Microsystems, http://java.sun.com/products/jms/docs.html, Mar. 2002.

[21] G. Weiss. Prologue. In G. Weiss, editor, *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*, page 1. The MIT Press, 2001.

[22] M. Wooldridge. Intelligent agents. In G. Weiss, editor, *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*, chapter 1, pages 27–77. The MIT Press, 2001.