Interfacing IndiGolog and OAA – A Toolkit for Advanced Multagent Applications

Alexei Lapouchnian and Yves Lespérance
Department of Computer Science,
York University,
Toronto, ON M3J 1P3
Canada
email: {alexei, lesperan}@cs.yorku.ca

Abstract

In this paper we describe an interface library IG-OAAlib that supports the development of Open Agent Architecture (OAA) agents using the IndiGolog agent programming language. OAA is a multiagent infrastructure that supports facilitated communication. IndiGolog is a high-level agent programming language based on logic that supports planning and allows complex agent behaviours to be specified. Full-fledged IndiGolog agents written using our interface library can be both reactive and proactive, thus overcoming one of the limitations of the OAA framework. The interface hides all of the low-level procedures that are used to communicate with the OAA system as well as OAA initialization, thereby leaving the IndiGolog programmer free to concentrate on the functionality of the agent.

1 Introduction

IndiGolog [De Giacomo and Levesque, 1999] is a very-high-level programming language for intelligent agents and robots that supports on-line planning and plan execution in dynamic and incompletely known environments. It allows the programmer to specify a logical model of the domain in the situation calculus and uses it to perform projection in planning/search and update when actions occur. Complex behaviours combining planning and reactivity can be specified in a rich concurrent programming language. It has been implemented on top of Prolog and is a very effective tool for programming individual agents for tasks that require planning and reasoning.

However, many applications are best delivered as multiagent systems that involve multiple interacting agents with specialized skills. Agents programmed in IndiGolog can be included in such systems, but until recently, they had always been interfaced using low-level protocols such as TCP/IP. In this paper, we describe a new interface mechanism IG-OAAlib that allows the easy integration of IndiGolog agents in multi-agent systems that use SRI's Open Agent Architecture (OAA) [Martin *et al.*, 1999] infrastructure. OAA provides high-level brokered communication facilities that can automatically route requests to agents that have the capabilities to serve them. It uses a Prolog-like Interagent Communication Language that makes it a good match for IndiGolog. The combination of OAA and IndiGolog provides a very powerful tool for developing multiagent systems for advanced applications (such as [McIlraith and Son, 2001]). As an example, we describe a multirobot mail delivery system that has been implemented using the framework.

Our IndiGolog-OAA interface mechanism allows IndiGolog agents to be *both proactive and reactive*. This overcomes a major limitation of Prolog-based agents in OAA since both IndiGolog and OAA require them to run their separate event loops. Here, we propose a solution that integrates these event loops, therefore allowing an IndiGolog agent to monitor both OAA and IndiGolog events concurrently.

2 IndiGolog

2.1 IndiGolog Agent Structure

An IndiGolog agent includes the following:

- A specification of the application domain dynamics. This is done declaratively in situation calculus [McCarthy and Hayes, 1979; Reiter, 2001].
- Behaviour specification. This is specified procedurally in a rich programming language with loops, non-determinism, concurrency, interrupts, etc. IndiGolog agents may perform sensing actions to acquire information at runtime as well as react to exogenous events.

2.2 Specifying Domain Dynamics in Situation Calculus

In IndiGolog domain theories are specified in the situation calculus [McCarthy and Hayes, 1979; Reiter, 2001], a language of predicate logic for representing dynamically changing worlds. In this language, a possible world history, which is simply a sequence of actions, is represented by a first order term called a *situation*. The constant *So* is used to denote the initial

situation and the term *do(a,s)* denotes the situation resulting from action *a* being performed in situation *s*.

Relations and functions that vary from situation to situation, called *predicate fluents* and *functional fluents* respectively, are represented by predicate and function symbols that take a situation term as last argument. A domain of application will be specified by theory that includes the following types of axioms [De Giacomo and Levesque, 1999; Reiter, 2001]:

- Axioms describing the initial situation, *So*.
- Action precondition axioms, one for each primitive action *a*, characterizing *Poss(a,s)*, which means that primitive action *a* is possible in situation *s*.
- Successor state axioms, one for each fluent *F*, which characterize the conditions under which *F(x,do(a,s))* holds in terms of what holds in situation *s*; they provide a solution to the frame problem [Reiter, 1991].
- Sensed fluent axioms, which relate 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.

In the current IndiGolog implementation, the initial situation is specified as a set of Prolog clauses, which means that only completely specified initial situations can be handled. We hope to accommodate limited forms of incompleteness in the future implementations

2.3 Behaviour specification

The behaviour of an IndiGolog agent is specified procedurally using a rich set of high-level programming constructs which include recursive procedures, if-then-else, while loops, non-deterministic execution of two programs, non-deterministic choice of arguments, non-deterministic iteration of a program, concurrent execution of two programs with or without prioritization, interrupts, etc.

A powerful *search block* facility is available in IndiGolog. By default IndiGolog programs are executed in an *on-line* fashion: all the non-deterministic choices are treated as random ones and, any action selected is executed immediately. On the other hand, for a program in a search block the interpreter does an *offline* search. It looks for a sequence of actions constituting a legal execution of the program resolving non-deterministic choices appropriately, *before* actually executing them. After a sequence of actions is found for the search block, it needs to be rechecked if an exogenous action occurs to see if it still leads to the final situation for the search block. If the previously found sequence of actions is no longer valid, replanning (a new search) is done.

3 The Open Agent Architecture

The Open Agent Architecture is a framework for constructing multiagent systems developed at SRI International [Martin *et al.*, 1999]. The primary goal of OAA is to provide a means for integrating heterogeneous applications in a distributed infrastructure. OAA incorporates some of the dynamism and extensibility of blackboard approaches, the efficiency associated with distributed objects (e.g. CORBA, DCOM), and the rich and complex interactions of communicating agents.

OAA provides a communication infrastructure for the agents as well as the Interagent Communication Language (ICL) that is used to exchange information between agents. The system has at least one special agent called *facilitator*. This agent acts as a broker/matchmaker and all interagent communication goes through it. The facilitator keeps track of all the agents in its system, their addresses, and their *capabilities*. Requests are automatically routed to agents that have the capabilities to handle them. It is possible to create a hierarchy of facilitators, each with its own subsystem of agents. The current version of OAA supports such agents written in Java, Quintus and SICStus Prolog, C/C++, and Compaq's Web Language.

When a client agent enters the system, it connects to the facilitator agent and provides it with a list of *solvables* – the agent's capabilities. These provide the high level interface to the agent. A callback method associated with a capability is invoked when a request involving that capability is received. Agents can dynamically add and remove solvables. The solvables can be of two types: *procedure* and *data*. Procedure solvables describe some service that can be performed by the agent, while data solvables are most commonly used to create a data storage that is shared among the agents in the system.

When an agent wants some services performed by other agents, it issues an *oaa_Solve(goal, parameters)* request that is forwarded to an appropriate agent by the facilitator. The *goal* part of such request is an ICL description of the service to be performed. A number of parameters can be used in the *oaa_Solve* request to specify, for example, whether this call should be blocking, or to say whether multiple agents are allowed to attempt to solve the problem simultaneously. The result of the query is returned by binding variables as in Prolog.

4 Our IndiGolog-OAA Interfacing Scheme

Our interfacing scheme is designed to integrate full-fledged IndiGolog agents in an OAA-based system without giving up any of the usual functionalities (e.g. data solvables, interrupts, etc.) of either tools. It supports the integration into an OAA system of IndiGolog agents that are both reactive and proactive, thus overcoming one of the major limitations of Prolog-based agents in OAA (see Figure 1). To be able to execute IndiGolog program while keeping track of incoming OAA events, we need to integrate the event loops of IndiGolog and OAA.

To allow this, we need to use an asynchronous communication scheme. Other agents should be using non-blocking calls when requesting services from IndiGolog OAA agents built using this interface. It is up

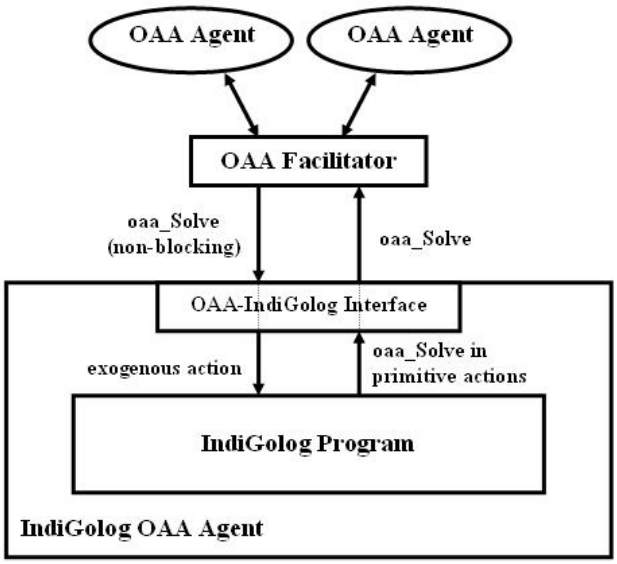

Figure 1. IndiGolog-OAA Interface

to the IndiGolog program to decide when and how to respond to these requests. We advise that the calls to OAA made from an IndiGolog agent using this interface also be non-blocking, to allow the agent to react promptly to its incoming events.

An IndiGolog OAA agent using this interface will be able to execute its program (e.g. reasoning/planning) while still keeping track of incoming OAA messages (most notably requests for service coming from other agents in the system). Support for exogenous events in IndiGolog allows us to automatically check the OAA library for incoming events after every action executed by the IndiGolog interpreter. The process of receiving OAA events is completely transparent to the programmer: they appear in the program as IndiGolog exogenous actions.

The OAA primitives can be used in the implementations of IndiGolog primitive actions. The interface lets the OAA library handle all the incoming messages that are not calls to the solvables the agent has defined. Such events may be related to the management of data solvables defined at this agent and auxiliary activities such as message tracing.

In order to be able to react to OAA events appropriately, an IndiGolog agent needs to have exogenous actions defined, one for every procedure solvable that the agent declares. Incoming OAA events that are intercepted by this interface appear in IndiGolog program as these exogenous actions. They are inserted into the action history in the order that they are received. Successor state axioms involving these exogenous actions should be defined, changing the values of certain fluents in accordance with the event received.

The interface also hides all of the code that is needed to connect to the OAA facilitator, declare solvables, etc. However, here we concentrate on the other benefits of this interface.

We next present an example application before returning to the details of the interface implementation.

5 Example Application: Multirobot Mail Delivery

5.1 Overview

Let us now describe an application that we have implemented with our toolkit. It involves a multirobot mail delivery system. The setting is a virtual office environment, which models the graduate labs area in our department. This environment is populated by a varying number of robots capable of delivering packages (the robots are currently simulated). The assignment of packages to robots is the responsibility of a dispatcher agent. The dispatcher and the robots implement a variant of the contract net protocol to select the best robot to deliver a package. The system is open in the sense that the robots can come online and go offline (presumably after completing the orders they were assigned) at any time. If no robot can deliver a package, the order is queued until there is a robot available. The GUI agent is used to get users' orders and visualize the system by displaying the status and locations of all the robots in the system.

This example system includes six different agents. Five of them are implemented in Java. The mail delivery robots are actually implemented using an architecture that involves two agents: a high-level control agent written in IndiGolog and a low-level control agent written in Java. The IndiGolog-based High-Level Control agent (HLC) is responsible for bidding for available mail delivery orders and for constructing optimal plans for carrying out the orders awarded to the robot. It takes full advantage of the IndiGolog-OAA interface through which it can execute its package delivery plan while responding to requests for bids coming from the dispatcher and modifying the plan to incorporate newly awarded orders. The Java-based Low-Level Control agent (LLC) simulates the movement of the robot through the environment. Each robot has a unique ID. This ID is given to both the LLC and the HLC and is used by them to find each other and form a single logical robot controller while still remaining two separate agents.

5.2 Individual Agent Details

5.2.1 The GUI, PathPlanner, and DB Agents

The GUI agent displays the virtual environment with the current position and status of every mail delivery robot and package as well as the status of all delivery orders. It is used by the user of the system to place orders for package delivery. This agent is multithreaded and all the synchronous calls to OAA are executed in their own threads, thus allowing it to accommodate a large number of robots and orders. The robots use the GUI's solvables `action_update` (we omit parameters here) and `position_update` to send information about their current activity and location respectively.

The PathPlanner agent knows the distances and paths between any pair of locations. It is used mainly by the robots to prepare bids for new orders and for

traveling from location to location. It has two solvables: `distance` returns the distance between a pair of locations while `path` returns a list of locations that constitutes a path from one location to another.

The DB agent accepts bids from robots and sends them to the Dispatcher while also keeping track of queued orders. Since the DB agent acts like a blackboard, its functionality could have been easily implemented by the OAA facilities such as triggers and data solvables. Unfortunately, there were difficulties with that approach and we decided to have a dedicated DB agent instead of relying on the OAA functionality.

5.2.2 The Dispatcher

The Dispatcher is responsible for taking orders from the GUI agent and distributing them among available robots. After receiving an order, the Dispatcher checks it for validity and then issues a call for bids that is sent to all the mail delivery robots currently online. The Dispatcher does not have to know the addresses of the agents it is sending this call for bids to, or how many such agents are currently in the system. The OAA Facilitator automatically forwards this query to all the agents that are capable of handling it, thus illustrating the openness and scalability of OAA.

The robots will reply to the call for bids by sending their bids to the Dispatcher. It will then compare the bids and select the robot that is the closest to the origin of the mail package being processed by the Dispatcher and award the order to that robot. If there are no replies to the call for bids, the order is queued. The request for bids will then be sent to any robot that posts "available" status and automatically awarded to the first robot that replies with a bid.

The GUI uses the Dispatcher's solvable `request_delivery` to inform it of a new order.

5.2.3 The Low-Level Robot Control Agent

The LLC is the low-level motion control subsystem of a mail delivery robot. It acts on orders from the corresponding High-Level Control agent. From the point of view of the HLC moving from one location to another is a primitive action `go(Loc1,Loc2)`. On the other hand, the LLC is interested in the exact path it needs to follow. The LLC uses the PathPlanner's `path` solvable to get that path. While following the path the LLC sends updates on the position of the robot to the GUI agent. To simulate the lengthy task of moving from one location to another, the time that the LLC "travels" between two locations is proportional to the distance between them. When LLC reaches its destination, it sends a `movement_complete` event to its HLC.

5.2.4 The High-Level Control Agent

The HLC is the high-level reasoning part of the mail delivery robot. It is implemented in IndiGolog and is responsible for bidding for new delivery orders and constructing and executing plans for delivering the awarded packages. This agent uses the IndiGolog-OAA interfacing mechanism described earlier and is able to effectively execute its package delivery plans while monitoring for incoming OAA events and reacting appropriately to calls for bids and new contract assignments. The agent has three solvables defined: `request_for_bids` is used by the Dispatcher to ask the robot to bid on a newly placed order; the `deliver` event is sent by the Dispatcher to award an order to the robot; and the `movement_complete` event is used by the LLC to notify the HLC of its arrival at the destination.

The following fluents are used by the HLC to model the world state:

- `current_location` – stores the current location of the robot
- `next_location` – stores the next location of the robot, where it is currently moving
- `canmove` – true when the robot is stationary, false otherwise
- `delivery(From,To,OrderNo)` – stores order status (ordered / onboard / completed)
- `bid_requested(From,To,OrderNo)` – true when the robot has to bid on the order
- `llc_address` – stores the OAA address of the corresponding LLC agent
- `dist(From,To)` – stores the distance between From and To locations

The following causal laws are used to update these fluents' values when OAA events arrive.

Fluent `bid_requested` becomes true for a particular order when `request_for_bids` is received:

```
causes_val(request_for_bids(F,T,ON),
      bid_requested(F,T,ON),true,true).
```

Fluent `delivery` becomes 'ordered' when the agent is awarded the delivery:

```
causes_val(deliver(F,T,ON),
      delivery(F,T,ON),ordered,true).
```

The `movement_complete` message from the associated Low-Level Control agent signals that the robot has reached the destination:

```
causes_val(movement_complete,
      canmove,true,true).
causes_val(movement_complete,
      current_loc,N,N=next_location).
```

Most of the primitive actions used by HLC have self-explanatory names and we will only mention that the `delivery_completed` action sends a message to the GUI agent saying that the robot has successfully completed the delivery; the primitive action `go(LLC_addr,From,To)` sends the `go(From,To)` event to LLC agent. The extra parameter `LLC_addr` is used in the call to `oaa_Solve` to tell the Facilitator that this event has to be sent only to the one particular LLC agent associated with the given robot, not all the agents capable of handling `go`. Similarly the LLC uses the address of the corresponding HLC to send `movement_complete` events. Presented below is the main procedure of the HLC agent (see `http://www.cs.yorku.ca/~lesperan/IG-OAAlib/` for the complete source code).

```
proc(control, [
  prioritized_interrupts([
    %high priority: handles bid requests
    interrupt([f,t,o],
        bid_requested(f,t,o)=true,
        pi([l,d], [?(l=next_location),
        ?(d=dist(l,f)), bid(o,d)])),
    %medium priority: handles newly assigned orders
    interrupt([f,t,o], and(canmove,
        delivery(f,t,o)=ordered),
        search(pconc(minimize_distance(0),
                     envSimulator))),
   %low priority interrupt: when nothing to do, wait
    interrupt(true,no_op) ]) ]).
%Environment simulator – simulates exogenous actions
proc(envSimulator,while(canmove=false,
            sim(movement_complete))).
```

The high priority interrupt fires when the agent receives a `request_for_bids` event from the Dispatcher. It produces a bid that is sent back to the Dispatcher. Presently, the bid is simply based on the distance from the location where it is currently heading (for simplicity we do not allow the robots to change directions midway) to the new package sender's location; more interesting bidding strategies could be used. The medium priority interrupt fires when the Dispatcher awards a new delivery to this robot. Then, the HLC plans an optimal delivery route that serves all orders assigned to the robot. The lowest priority interrupt is there simply to prevent the HLC from terminating when it has nothing to do.

To plan a delivery route, the second interrupt runs an iterative deepening search procedure (`minimize_distance`) to come up with an offline plan that minimizes the distance the agent has to travel. In our domain theory the precondition axiom for the `go` action requires the robot to be stationary – the `canmove` fluent has to be true. The only way for `canmove` to become true is through one of the causal laws above. This in turn is triggered by the arrival of a `movement_complete` event from the LLC. Since HLC is doing *offline* planning, and the plan has to be ready *before* it is executed, we run the offline planning routine concurrently with an environment simulator [Lespérance and Ng, 2000] that simulates `movement_complete` events. When the plan is executing, HLC will actually wait for the arrival of `movement_complete` before asking LLC to move to a new location.

The HLC code for route planning appears below. The iterative deepening search routine tries to come up with a plan to deliver all of the assigned packages with a given distance bound. If unable to do so, it increments the bound.

```
%Iterative deepening search
proc(minimize_distance(Max),
  ndet( serve_customers(Max),
      pi(nd,[?(nd is Max+1),
          minimize_distance(nd)]))).

proc(serve_customers(Max),
  ndet([       %Have all the orders been delivered?
      ?(neg(some([from,to,orderNo],
        or(delivery(from,to,orderNo)=ordered,
        delivery(from,to,orderNo)=onboard)))),
      no_op    %Ground case - done
    ],[ % Nondet. pick up or drop off an order
      ndet([  % Pick values s.t. the tests (?) succeed
            pi([f,t,on,llc,l,m,d],[
              ?(delivery(f,t,on)=ordered),
              ?(l = current_location),
              ?(llc = llc_address),
              %Execute actions with the picked params
              go(llc,l,f), pickUp(on),
              ?(d=dist(l,f)), ?(m is Max - d),
              %If the distance allowed is not used up,
              %recurse, else fail
              ?(m>=0), serve_customers(m)
            ])
           ],[
            pi([f,t,on,l,llc,m,d],[
              ?(delivery(f,t,on)=onboard),
              ?(l = current_location),
              ?(llc = llc_address),
              go(llc,l,t), dropOff(on),
              delivery_completed(on,f,t),
              ?(d=dist(l,t)), ?(m is Max - d),
              ?(m>=0), serve_customers(m)
]) ]) ]) ).
```

6 Interface Implementation Details

To allow IndiGolog agents to be both proactive and reactive, we want our IndiGolog-OAA interface to process incoming OAA events without giving complete control to the OAA library. A special exogenous action `get_event` is defined as part of the interface (see below). It runs after every primitive action in an IndiGolog program (unless it is specifically disabled). `get_event` first executes the procedures found in the main OAA event loop: it gets the top priority OAA event from the communication library and lets OAA process it. It repeats this until there are no more events waiting (i.e., until it receives 'timeout' event).

```
exog_occurs(get_event,E,H) :-
%Get OAA events. For events that we are interested in,
%get_event will automatically add them to oaa_event_queue
    oaa_loop,
    oaa_event_queue(Q), %Get the queue
    \+ Q = [], %Succeeds if the queue is not empty
    %Add OAA events to IndiGolog history
    extract_events(E,H,Q).
```

The reason why we need to let OAA process the events rather than extract them manually is simple: in addition to events that result from some agents requesting the services of our IndiGolog agent there are other OAA events that we don't want to deal with. These could be events that update a data solvable declared at this particular agent, or these could be events that turn on the system's tracing facility, and so on. We

let the OAA library handle these events since we want the combined IndiGolog OAA agent to be compliant with OAA specifications to the maximum degree possible.

To achieve the task of separating OAA events that are calls to user-defined solvables from other OAA events, we define and register with OAA a default callback that is called every time some agent requests the services of our IndiGolog OAA agent. This callback is given the goal (the problem that we have to solve) and adds this goal to a queue that holds the OAA events to be processed by the IndiGolog agent. The OAA library sends the success message to the caller immediately. This is why other agents should only send non-blocking requests for the services provided by IndiGolog OAA agents. If some other agent waits for the answer to its query, its `oaa_Solve` call will return successfully, but the variables through which that agent expects to get the answer will remain unbound.

`get_event` then calls the `extract_event` procedure that processes the queue and extracts the events from it. It adds the events to IndiGolog program history, thus turning the newly received goals into exogenous actions that appear to have been executed and changing the value of fluents appropriately. The programmer has to provide the appropriate causal laws/ successor-state axioms. Suppose that we register the solvable: `movement_complete(Location).` This could be a notification from a certain mobile robot that our IndiGolog agent controls. The following axiom specifies one of the possible changes in the system caused by the arrival of the goal `movement_complete(Location)`:

```
causes_val(movement_complete(Loc),
           current_location,Loc,true).
```

This effectively says that the value of the fluent `current_location` changes to become the location that the robot has just arrived at.

7 Conclusion and Future Work

In this paper, we have presented an IndiGolog-OAA interfacing mechanism that we think adds value to both tools. It provides easy access to a multiagent platform for IndiGolog, allowing us to use this language in a wide range of new applications. Moreover, since OAA's ICL is Prolog-based, it makes it a great match to the current implementation of IndiGolog. On the other hand, the built-in concurrency of IndiGolog allows IndiGolog-based OAA agents to be both reactive and proactive and thus much more powerful than the previously supported Prolog-based agents. This interface adds a new powerful high-level programming language to the set of languages supported by OAA. The system is available for download at `http://www.cs.yorku.ca/~lesperan/IG-OAAlib/`.

We are interested in applying this work in a variety of domains. One area of interest is personal service robotics with robots having multiple skills such as finding people, giving tours, etc. With Erich Leung, we have started integrating software agents into the system that locates people based on where they logged-in and their typical schedule. We would also like to use IndiGolog to program a smarter matchmaker for some domain, one that supports compound queries. Other potentially interesting applications include semantic web services [McIlraith *et al.*, 2001].

The choice of OAA as a multiagent platform to interface IndiGolog to arose from their common Prolog heritage, which suits them to developing agents that perform reasoning and planning. We are also examining the use of IndiGolog in combination with FIPA-compliant platforms and would like to develop tools for this.

References

[De Giacomo and Levesque, 1999] G. De Giacomo and H. Levesque. An incremental interpreter for high-level programs with sensing. In Logical Foundations for Cognitive Agents, Contributions in Honor of Ray Reiter, pages 86–102, 1999. Agent-Based Systems, LNCS. Springer-Verlag, 2000.

[De Giacomo *et al.*, 2000] G. De Giacomo, Y. Lespérance, and H.J. Levesque. ConGolog, a concurrent programming language based on the situation calculus. *Artificial Intelligence*, 121, 109-169, 2000.

[Lespérance and Ng, 2000] Y. Lespérance and H.-K. Ng. Integrating Planning into Reactive High-Level Robot Programs. In *Proceedings of the Second International Cognitive Robotics Workshop*, 49-54, Berlin, Germany, August, 2000.

[Martin *et al.*, 1999] 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, January-March 1999.

[McCarthy and Hayes, 1979] John McCarthy and Patrick Hayes, 'Some philosophical problems from the standpoint of artificial intelligence', in *Machine Intelligence*, eds., B. Meltzer and D. Michie, volume 4, 463–502, Edinburgh University Press, Edinburgh, UK, (1979).

[McIlraith and Son, 2001] S. McIlraith and T. C. Son. Adapting Golog for Programming the Semantic Web. *Proceedings of the Fifth Symposium on Logical Formalizations of Commonsense Reasoning (Common Sense 2001),* May 2001.

[McIlraith *et al.,* 2001] S. McIlraith., T.C. Son, and H. Zeng 'Semantic Web Services' , *IEEE Intelligent Systems. Special Issue on the Semantic Web.* 16(2):46-53, March/April, 2001. Copyright IEEE, 2001.

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

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