Journal of Intelligent Information Systems, ?, 275–294 (????)

IMPACT: A system for building agent applications

T.J. ROGERS, ROBERT ROSS, AND V.S. SUBRAHMANIAN

{*rogers*, *robross*, *vs*}@cs.umd.edu

Department of Computer Science, Institute for Advanced Computer Studies and Institute for Systems Research, University of Maryland, College Park, MD 20742, USA.

Received November 25, 1995; Revised March 30, 1996

Editor: Zbigniew Ras

Abstract. IMPACT (Interactive Maryland Platform for Agents Collaborating Together) provides a platform and environment for agent and software interoperability being developed as a joint, multinational effort with participants from the University of Maryland, the Technische Universität Wien, Bar-Ilan University, the University of Koblenz, and the Universita di Torino. Here, we describe the overall architecture of the IMPACT system, and outline how this architecture (i) allows agents to be developed either from scratch, or by extending legacy code-bases, (ii) allows agents to interact with one another, (iii) allows agents to have a variety of capabilities (reactive, autonomous, intelligent, mobile, replicating) and behaviors, and (iv) how IMPACT provides a variety of infrastructural services that may be used by agents to interact with one another.

Keywords: software agents, heterogeneous software integration, active rules

1. Introduction

Over the last few years, there has been increasing interest in the area of software agents that provide a wide variety of services for human users. Such services are spread over a wide area, and include identification of interesting newspaper articles, software robots that perform tasks (and plan) on a user's behalf, content based routers, agent based telecommunication applications, and solutions to logistics applications. Despite all of this wide interest, the theory of agent systems is remarkably undeveloped. Most existing "intelligent agent" systems are either well-engineered systems that lack a firm theoretical basis, or involve agent theories that are as yet unimplemented.

IMPACT ("Interactive Maryland Platform for Agents Collaborating Together") is a multinational project whose aim is to define a formal theory of software agents, implement it (or carefully specified fragments of it) efficiently, and develop an appropriate suite of applications on top of this implementation. The primary aim of this paper is to describe the IMPACT system. The reader will find full technical details of the mathematical theory underlying IMPACT in [2, 5, 6, 4, 7].

The organization of this paper is as follows. In Section 2 we briefly describe what an agent is, and illustrate how such agents may be used to support a variety of properties traditionally attributed to agents. In Section 3, we will describe the architecture of the IMPACT system and show that it consists of five basic components. In Section 4, we will describe the most important part of the IMPACT

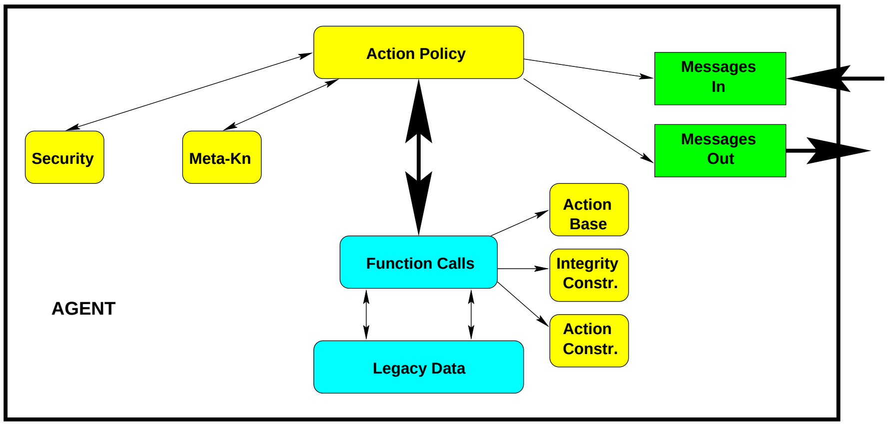

Figure 1. Basic Architecture of IMPACT Agents

system, viz. the Agent Development Environment — this is used to "agentize" a specialized or a legacy piece of software. Section 5 describes the IMPACT Server which agents may use to find other agents providing certain given services and/or to resolve type and ontological questions. Section 6 describes the "data connections" used by IMPACT to access software that has not been "agentized." Section 7 describes an Agent Roost — a program that manages a set of sleeping processes and wakes them when a state change that affects them has occurred. Section 8 describes the IMPACT Agent Log that is a facility provided by IMPACT to log actions and/or messages taken by agents.

2. Agent Definition

As different application programs reason with different types of data and as even programs that deal with the same types of data often manipulate that data in a variety of different ways, it is critical that any notion of agent-hood be applicable to arbitrary software programs. Agent developers should be able to select data structures that best suit the application functions desired by users of the application they are building. Figure 1 shows the architecture of a full-fledged software agent. It is important to note that all agents have the same architecture and hence the same components, but the *content* of these components can be different, leading to different behaviors and capabilities offered by different agents.

2.1. Agent Data Structures

As all agents are built "on top" of some existing body of code, we first need an abstract definition of what that body of code looks like.

- First, we need a specification of the data types or data structures that the agent manipulates. As usual, each data type has an associated *domain* which is the space of objects of that type. For example, the data type `countries` may be an enumerated type containing names of all countries. At any given point, the *instantiation* or *content* of a data type is some subset of the space of the data-objects associated with that type.

- The above set of data structures is manipulated by a set of functions that are callable by external programs (such functions constitute the *application programmer interface* or API of the package on top of which the agent is being built. An agent includes a specification of all these API function calls' signatures (i.e. types of the inputs to such function calls and types of the output of such function calls).

Let us consider a simple example. Suppose we wish to "agentize" a body of software code used to reason about geographic information. The geographic information may be represented as a "quadtree" data structure. In this case, the data types managed by the agent might be:

1. The usual primitive data types (`integer, real, string, bool`), and

2. Specialized data types called `point` (a record containing `x,y` fields) and `node` (a record containing `info`, `point`, `nw`, `ne`, `sw`, `se` fields) where the last four fields are of type "pointer to `node`" and the `info` field is of type "a pointer to some (for now unspecified) record field".

3. If τ is one of these data types, then $\{\tau\}$ (the set containing only "τ") is also a data type.

The functions supported by the API associated with this application may be:

1. `range(p : point, n : ^node, r : int)` which takes as input, a point, a pointer to the root of a quadtree, and an integer, and returns as output, the set of all points in the tree which are within `r` units of the given point `p`.

2. `getinfo(p : point, n : ^node)` which takes as input, a point and a pointer to the root of a quadtree, and returns as output, the `info` field of the given point.

Of course, many other functions may also be defined by the API for the geographic reasoning application, but for purposes of illustration, the above two are enough.

Based on knowledge of the data structures and functions supported by a body of code, we may use a unified query language to query them. If p is the name of a package, and f is an n-ary function defined in that package, then $p : f(a_1, \ldots, a_n)$ is a code call. This code call says "Execute function f as defined in package p on the stated list of arguments." We assume this code call returns as output, a *set* of objects — if an atomic object is returned, it can be coerced into a set anyway. For instance, in the previous quadtree example, a code call could be `quadtree:range((20,30),t,40)` which says "find all objects in the quadtree whose root is pointed to by $\mathtt{t}$ which are within 40 units of location $(20, 30)$" — this query returns a set of points.

An *atomic code call condition* is an expression of the form $\mathtt{in}(\mathtt{X}, \mathtt{p} : \mathtt{f}(\mathtt{a}_1, \ldots, \mathtt{a}_n))$ which succeeds if $\mathtt{X}$ is in the set of answers returned by the code call in question. For example,

$$\mathtt{in}((25, 25), \mathtt{quadtree} : \mathtt{range}((20, 30), \mathtt{t}, 40))$$

is an atomic code call condition that succeeds if the point $(25, 25)$ is in the quadtree whose root is pointed to by $\mathtt{t}$ and is within 40 units of location $(20, 30)$.

A *code call condition* is a conjunction of atomic code call conditions and deconstruction and constraint operations. An example of a deconstruction operation is $\mathtt{V.x}$ which accesses the $\mathtt{x}$ field of a variable $\mathtt{V}$ ranging over records that have an $\mathtt{x}$ field. An example of a constraint is $\mathtt{V.x} > 25$ which checks to see if this condition is true.

Code call conditions provide a simple, but powerful syntax to access heterogeneous data structures. For example, the code call condition

$$\mathtt{in}(\mathtt{X}, \mathtt{oracle} : \mathtt{select}(\mathtt{emp}, \mathtt{sal}, >, 100000))\ \&$$

$$\mathtt{in}(\mathtt{Y}, \mathtt{image} : \mathtt{select}(\mathtt{imdb}, \mathtt{X.name}))\ \&$$

$$\mathtt{in}(\text{“}\mathtt{Mary}'', \mathtt{imagedb} : \mathtt{findpeople}(\mathtt{Y}))$$

is a complex condition that accesses and merges data across Oracle and an image database. It first selects all people who make over 100K from an Oracle database and for each such person, finds a picture containing that person with another person called Mary. It generalizes the notion of join in relational databases to a join across a relational and image database.

2.2. Message Box

In addition to the data types of the code that an agent is built on top of, IMPACT provides a special "messaging" package which may be "added on" to agents so that they are able to handle messaging. The message data type is basically a table having the schema

$$(\mathtt{IO}, \mathtt{Src}, \mathtt{Dest}, \mathtt{Message}, \mathtt{Time})$$

where the `IO` field specifies if an incoming or outgoing message is being referenced, `Src,Dest` respectively specify the source and destination of the message, and `Time` specifies the time at which the message was received/sent. The syntax of the field `Message` can be defined in many different ways (e.g. a text file, a remote procedure call, a KQML message, etc.) The message box package includes functions to access such messages using operations from the standard relational algebra [12].

At any given point in time, the actual set of objects in the data structures (and message box) managed by the agent constitutes the *state* of the agent. For example, in the case of a quadtree data structure, the state of the agent consists of (i) all the messages in the agent's message box, and (ii) all the nodes and information actually in the quadtree. Likewise, in the case of the image database above, the contents of the image database agent contains all the image objects in the database.

2.3. Actions

The agent has a set of *actions* that can change its state. Such actions may include reading a message from the message box, responding to a message, executing a request "as is", executing a modified request, cloning a copy of the agent and moving it to a remote host, updating the agent data structures, etc. Even doing nothing may be an action.

Every action has a precondition (which is a code call condition), a set of effects that describe how the agent state changes when the action is executed, and an *execution script or method* consisting of a body of physical code that implements the action. For instance, when a robotic agent executes the action `move((X,Y),(X',Y'))`, not only must its state be updated to reflect the move, but an execution script to actually move the robot to its new location must be executed. Such an action may be stated as follows:

1. **Precondition:** $\mathtt{in}((\mathtt{X},\mathtt{Y}), \mathtt{robot\_data}:\mathtt{location}())$.
2. **Add list:** $\mathtt{in}((\mathtt{X}',\mathtt{Y}'), \mathtt{robot\_data}:\mathtt{location}())$.
3. **Delete list:** $\mathtt{in}((\mathtt{X},\mathtt{Y}), \mathtt{robot\_data}:\mathtt{location}())$.

This specification assumes that the robot's data structures include a package called `robot_data` which supports an API function call named `location()` which returns the current location of the robot.

2.4. Notion of Concurrency

The agent has an associated body of code implementing a *notion of concurrency*. Intuitively, a notion of concurrency takes a set of actions as input, and returns a single action (which "combines" the input actions together) as output. There are numerous possible notions of concurrency.

1. Ordered Execution: Suppose $\{\alpha_1, \ldots, \alpha_n\}$ is a set of actions (with instantiated parameters). If there exists a total ordering on the α_i's (e.g. a lexicographic ordering), then one way of executing these actions concurrently is via the composite action defined as follows:

   ORDERED_EXEC$(\{\alpha_1, \ldots, \alpha_n\}, S)$ {
   $L' = \emptyset$; $S' = S$;

L = sorted list enumerating $\alpha_1, \ldots, \alpha_n$ in ascending order.
while $L \neq \emptyset$ **do** {
  $\alpha = head(L)$;
  $L = tail(L)$;
  if $pre(\alpha)$ is true in S' then {
    $L' = append(L', \alpha)$;
    S' = new state obtained by executing α in S';
  } ($\star$ end **if** $\star$)
} ($\star$ end **while** $\star$)
Execute L' in order in state S;
} ($\star$ end **ORDERED_EXEC** $\star$)

Clearly, this algorithm can be executed in time O($n$log$n$) because the most expensive step in the above algorithm is sorting L (assuming that executing actions in a state and evaluating preconditions can be done in constant time).

2. Three or four other examples notions of concurrency have been devised in IMPACT and proposed in [5, 7]. (The preceding example notion of concurrency is new).

2.5. *Action Constraints*

Each agent has a set of *action constraints* which are rules of the form "If the state satisfies some code call condition, then actions $\{a_1, \ldots, a_n\}$ cannot be concurrently executed." In the case of a robot, an example of an action constraint is:

$$\{move(left), move(right)\} \hookleftarrow$$

This action constraint says that an agent cannot move both left and right at the same time.

2.6. *Integrity Constraints*

Each agent has a set of *integrity constraints* that states of the agent are expected to satisfy. Such integrity constraints are of the form "If a code call condition is true, then a code call atom must be true." For instance, a robotic agent may have an integrity constraint which indicates that it can only be at one point at a given time. This is expressed as:

$$P = P' \leftarrow in(P, robot\_data : location()) \wedge in(P', robot\_data : location()).$$

Alternatively, if robots are viewed as being entities on a battlefield, then an example integrity constraint may be:

$$\begin{aligned} 1 = 2 \leftarrow\ & in(P, robot\_data : location()) \wedge \\ & in(Q, quadtree : range(P, t, 25)) \wedge \\ & in(true, Q.\texttt{info.hostile}). \end{aligned}$$

This constraint says that if there is a hostile entity within 25 units of the robot, then $1 = 2$. Of course, this means that for integrity to be preserved, no hostile unit must be within 25 units of the robot. (Whenever an enemy entity is within this distance, the robot must presumably move away!).

2.7. Agent Program

Each agent has a set of rules called the *Agent Program* specifying the operating principles under which the agent is functioning. These rules describe the do's and dont's for the agent. They specify what the agent may do, what it must do, what it may not do, etc. The Agent Program uses deontic modalities to implement what the agent can and cannot do. If $\alpha(\vec{t})$ is an action with parameters $\vec{t}$, then $\mathbf{O}\alpha(\vec{t}), \mathbf{P}\alpha(\vec{t}), \mathbf{F}\alpha(\vec{t}), \mathbf{Do}\alpha(\vec{t}), \mathbf{W}\alpha(\vec{t})$ are called *action status atoms.* These action status atoms are read (respectively) as $\alpha(\vec{t})$ is *obligatory, permitted, forbidden, done,* and the obligation to do $\alpha(\vec{t})$ is *waived.* If A is an action status atom, then A and $\neg A$ are called *action status literals.* An *agent program* is a finite set of rules of the form:

$$A \leftarrow \chi \wedge L_1 \wedge \cdots \wedge L_n$$

where A is an action status atom, χ is a code call condition, and $L_1, \ldots, L_n$ are action status literals. A simple agent program is given below. This agent program may be implemented by an entity-tracking program that tracks locations of friendly and enemy units on a digital battlefield.

$$\begin{aligned}
\mathbf{Do}\mathit{send_warning}(E, M) \leftarrow\ & \mathit{in}(L, \mathit{entity} : \mathit{curloc}(E)) \wedge \\
& \mathit{in}(E', \mathit{entity} : \mathit{hostiles}()) \wedge \\
& \mathit{in}(L', \mathit{entity} : \mathit{curloc}(E')) \wedge \\
& \mathit{in}(D, \mathit{quadtree} : \mathit{distance}(L, L')) \wedge \\
& < (D, 25) \wedge \\
& = (M, \mathit{file} : \mathit{create}(E', L', D)). \\
\mathbf{Do}\mathit{log}(E, M, T) \leftarrow\ & \mathbf{Do}\mathit{send_warning}(E, M) \wedge \\
& \mathit{in}(T, \mathit{clock} : \mathit{now}()) \wedge \\
& \neg \mathbf{F}\mathit{logging}.
\end{aligned}$$

The first rule says that the entity tracking program must send a warning message M to entity E if a hostile enemy entity is within 25 units of E and M is a file created by the entity tracking program using various pieces of information it has about the enemy unit. The second rule says that whenever such a warning message is sent, the fact that the message was sent must be logged unless logging is forbidden (which may be the case when the enemy has lots of electronic eavesdropping equipment in the area).

2.8. *Agent Behavior*

We assume that when the agent is initially constructed and deployed, all integrity constraints are satisfied by the agent's state. This is analogous to requiring that the agent, when built and deployed, is not "messed up" right at the beginning. Given this, we can ensure that whatever the agent does, it maintains consistency of the integrity constraints by never executing actions that force it to transition to an inconsistent state.

In this paper, we will assume that an agent $\mathcal{B}$ can *directly* change an agent $\mathcal{A}$'s state only by sending it a message (and thus causing an update to the agent's mailbox). All other changes to agent $\mathcal{A}$'s state must be made by agent $\mathcal{A}$, perhaps as a response to such a message from agent $\mathcal{B}$. There is no loss of generality in making this assumption — for instance, if agents $\mathcal{A}$ and $\mathcal{B}$ share a common workspace, this workspace can be part of both their respective message boxes. Similarly, for clock driven agents, one can think of a "clock" agent sending a message to agent $\mathcal{A}$. Likewise, a physical sensor (e.g. a temperature sensor) may be viewed as an entity that sends a message to an agent that tracks temperature data and reacts to it.

Thus, everytime an agent $\mathcal{A}$ receives a message, its integrity constraints *may* get violated. The agent's job is to compute a set of actions to take which, if executed concurrently, satisfy the following conditions:

1. satisfies the action constraints
2. leads to a new state (of the agent) that satisfies the integrity constraints
3. satisfies all rules of the agent's program.

In fact, in our framework, a *status set* is a set of ground action status atoms that preserve such conditions, and also satisfy some consistency conditions. We will not go into the formal theory and semantics of agent programs here — that is described in detail in [5]. The important point to remember is that agents continuously respond to messages (state changes) by computing such a status set, and concurrently executing all the actions of the form $\mathbf{Do}\alpha$ in that status set. The entire implementation of the IMPACT system is built to efficiently and scalably support this need.

3. IMPACT Architecture

The IMPACT system contains *five* major software components that support the creation and deployment of software agents. It is important to note that IMPACT provides support for (i) developing an agent *prior* to deploying it, and (ii) support for a deployed agent to carry out its mission in accordance with the semantics of the agent.

1. **Agent Development Environment.** When agent developers build agents, they use the Agent Development Environment (**AgentDE**for short). The **AgentDE**

provides a graphical interface, and a set of libraries that agent developers may pick and choose from to configure their agents. The **AgentDE** provides a variety of algorithms that agent developers may use to "test" out their agents' behaviors.

2. **IMPACT Server.** Once agent developers have "finished" developing an agent, they can register the services provided by the agent with the IMPACT Server. The IMPACT Server allows agents (once deployed) to query the IMPACT Server with requests for agents providing a service that is needed. The IMPACT Server also supports a variety of ontology and thesaurus related services, and maintains a hierarchy of types that agent developers may wish to access at the time when they are developing their agents.

3. **Agent Roost.** Once an agent's development is completed, and the agent is "deployed" by the developer, an executable body of code is shipped to an Agent Roost. Numerous Agent Roosts can be available on the network, and there is no need for synchronization across Agent Roosts. The Agent Roost manages the incoming messages of each agent. Whenever an agent receives a message, the Agent Roost "wakes" the agent, so that it can compute a status set, and act in accordance with its stated action policy.

4. **IMPACT Connections.** There are many cases when IMPACT agents need to access agents built in other frameworks (e.g. IBM's Aglet framework[10]), and/or access servers (e.g. Oracle server, Hermes[1] server), etc. The IMPACT Connections module supports this kind of connectivity.

5. **AgentLog.** In many agent applications, logging is an important activity for a variety of reasons, ranging from security needs to debugging and performance analysis requirements. The IMPACT AgentLog is a program that supports the creation, update, and maintenance of logs. In fact, the AgentLog is also useful for debugging during the agent development cycle.

In the remaining sections, we will describe these diverse components of the IMPACT system.

4. Agent Development Environment

As described earlier on in the paper, an agent consists of:

1. A specification of the data types manipulated by the agent,

2. A list of API function calls (with input/output types) supported by the agent,

3. A set of actions that the agent might take,

4. A notion of concurrency,

5. A set of integrity constraints,

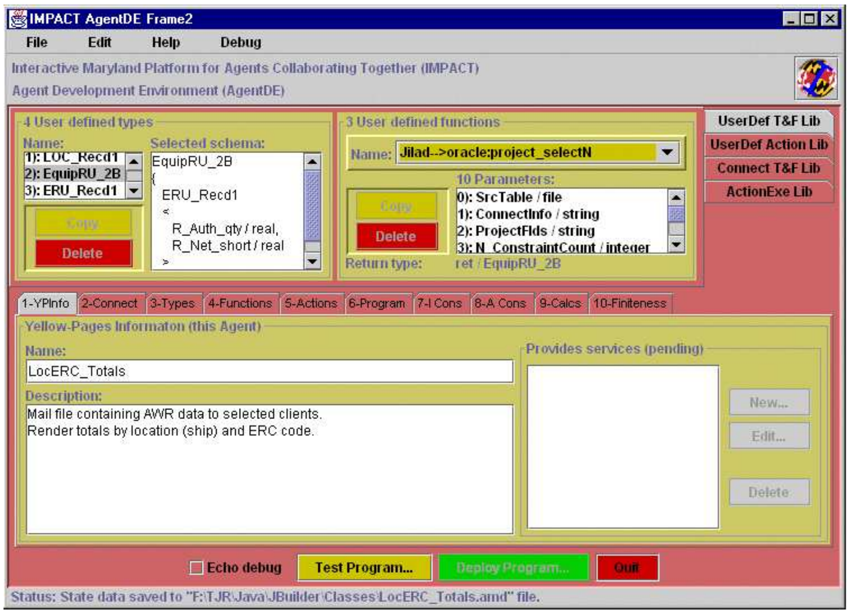

Figure 2. Types and Functions in the **AgentDE**

6. A set of action constraints, and

7. An agent program.

The IMPACT **AgentDE** provides an environment within which an agent developer can express all the above components of an agent, test that they work properly, and "deploy" the agent. We explain each of these three steps below.

The **AgentDE** maintains a library of data types, API functions, actions, and notions of concurrency that have been encountered in the past. Whenever an agent developer builds a new agent and defines new types, functions, actions, etc., these new items get added to the appropriate library so that when the same (or another authorized developer) accesses the **AgentDE**, these definitions are directly importable from the library for use in the agent development process. Figure 2 shows the basic interface of the **AgentDE** when an agent developer has finished specifying the data types and functions manipulated by her agent. Figure 3 shows the interface options that allow selections of actions by the agent developer, while Figure 4 shows computation parameters (e.g., notions of concurrency) that an agent developer may pick. Figure 5 shows a sample agent program.

Once the agent developer has specified these parameters, she may start testing the agent. The **AgentDE** contains code that performs the following **compile-time** tasks:

1. Parsing the 7 parts of the agent described above.

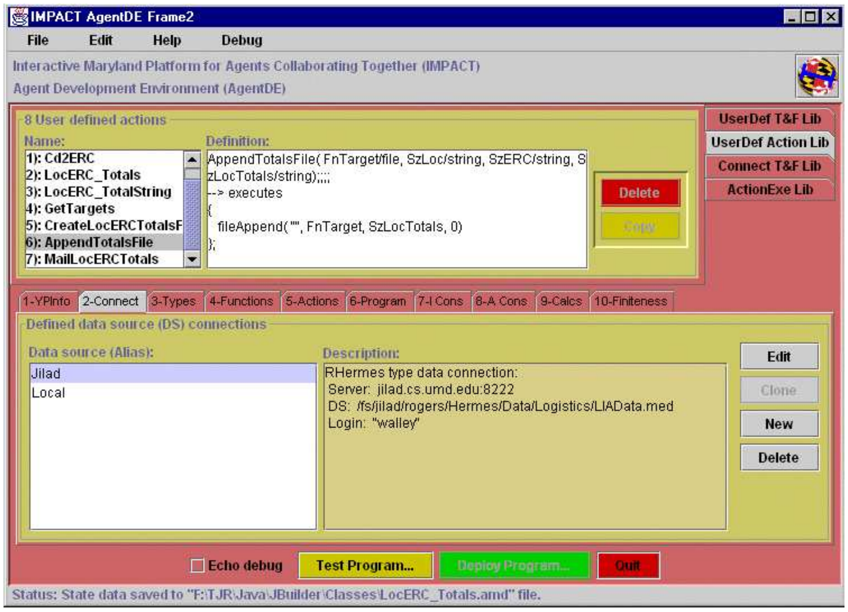

Figure 3. Actions in the **AgentDE**

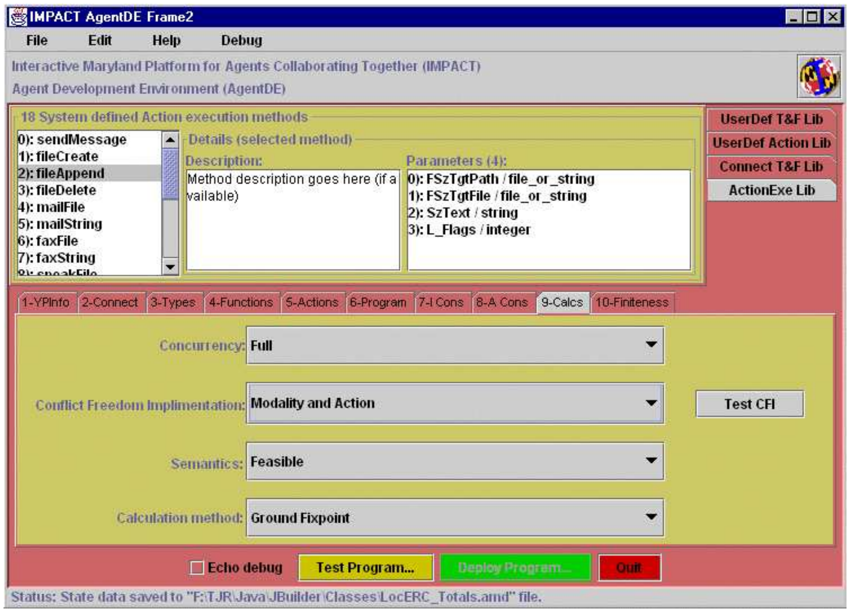

Figure 4. Computation Parameters in the **AgentDE**

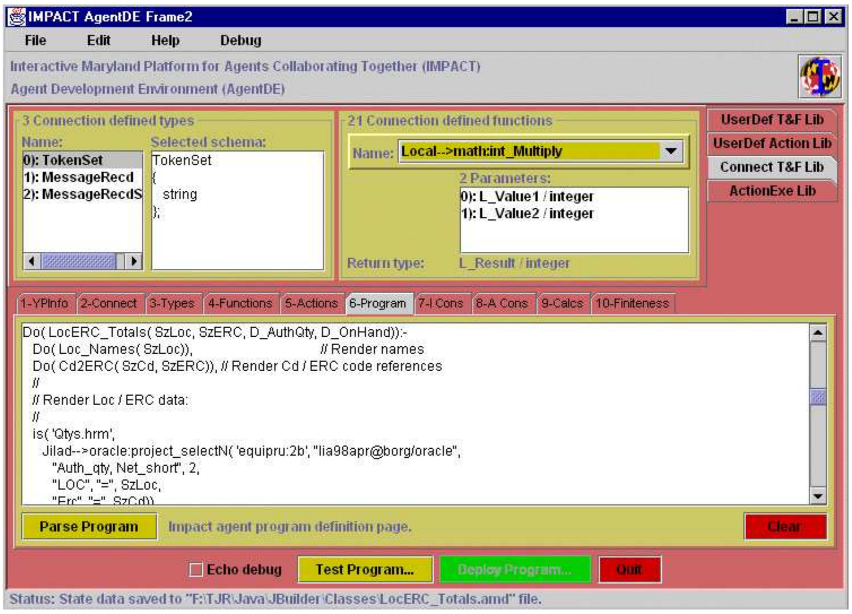

Figure 5. Agent Programs in the **AgentDE**

2. Performing a variety of compile time checks such as:

    (A) **Strong safety checking:** this check ensures that the evaluation of code call conditions in the body of a rule is feasible (i.e., there is a way of re-ordering code call atoms so that each code call's arguments are instantiated prior to an attempt to execute it) and finite (i.e., to ensure that no code call atom has an infinite answer).

    (B) **Deontic stratification:** this check ensures that an action status atom is never defined recursively in terms of its own negation. For instance, a rule such as $\mathbf{O}\alpha \leftarrow \neg\mathbf{P}\alpha$ is a rule that defines $\mathbf{O}\alpha$ cyclically (implicitly whenever $\mathbf{O}\alpha$ holds, $\mathbf{P}\alpha$ must hold as well).

    (C) **Boundedness:** this check ensures that the rules in an agent program can be "unfolded" so that after a finite number of unfolding steps, no positive action status atoms occur in the body of any rule.

The **AgentDE** contains algorithms to unfold out an agent program at compile time. Once this is done, every rule in the unfolded program has the form

$$Op\alpha(\vec{t}) \leftarrow \chi \wedge \neg A_1 \wedge \ldots \wedge \neg A_n$$

where the A_i's are action status atoms. At run time, each such rule is readily evaluable because the deontic stratification condition requires that each A_i's definition does not depend upon $Op\alpha(\vec{t})$. Now, computing a status set merely involves

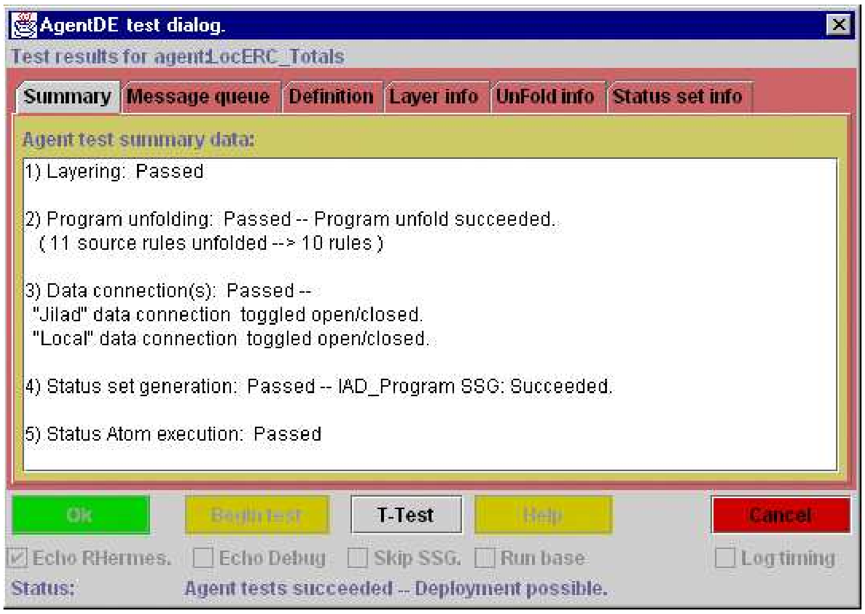

Figure 6. Screen Shot during Testing of an Agent Program

computing all these rule bodies — no inference (modus ponens) is performed at run time.

Figure 6 shows the screen the developer sees when she has finished testing her program. When the test is started, the first step is unfolding (which requires layering the rules of an agent program). Next, the data connections requested by the program are established. Finally, status sets are created and executed.

5. Impact Server

Figure 7 shows the architecture of the IMPACT Server. The IMPACT Server provides a variety of services that are required by a society of agents as a whole, rather than by a single agent. Such services include:

- **Registration Services:** When an agent is deployed by an agent developer using the **AgentDE**, it automatically provides the agent developer with the option of registering the agent with an IMPACT registration server. This server provides a graphical interface through which the services provided by the agent can be registered. This registration causes certain data structures to be updated with information on the services provided by the agent in question. Changes to service offerings may also be registered with this server (after undergoing an authentication check).

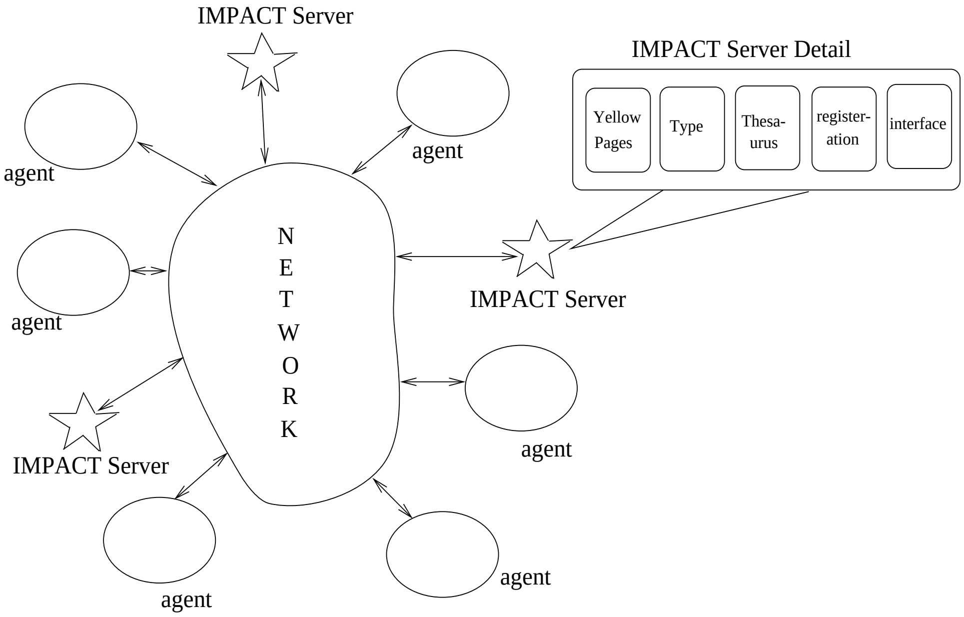

Figure 7. Architecture of IMPACT Server

- **Yellow Pages Services:** The IMPACT Yellow Pages Server accesses the data structures created by the registration server in read-only mode. When an agent wants to find another agent offering a desired service, it contacts the Yellow Pages Server and specifies one of two kinds of queries.
  - **Nearest Neighbor Queries** which are queries of the form "Find me the k best matches for a desired service, e.g. `sell:tickets(opera)`?" or
  - **Range Queries** which are queries of the form "Find me all services which are within a given distance from a desired service, e.g. `sell:tickets(opera)`."
- **Type Services:** The IMPACT Type Server allows agent developers to specify types used by them together with relationships between their types and existing types. For example, if an agent developer creates a type called `japanese_cars` and an existing type called `cars` is declared in the IMPACT Type server (either by this agent developer or someone else), then the agent developer can state that `japanese_cars` is a subtype of the type `cars`.
- **Thesaurus Server:** The IMPACT Thesaurus Server allows the Yellow Pages Server to relax queries to related queries. For example, suppose an agent wants to find an agent that offers the service `sell:cars`. Then an agent that offers the service `sell:automobiles` is probably a good match for this request, but to know this, the Yellow Pages Server needs access to a thesaurus.

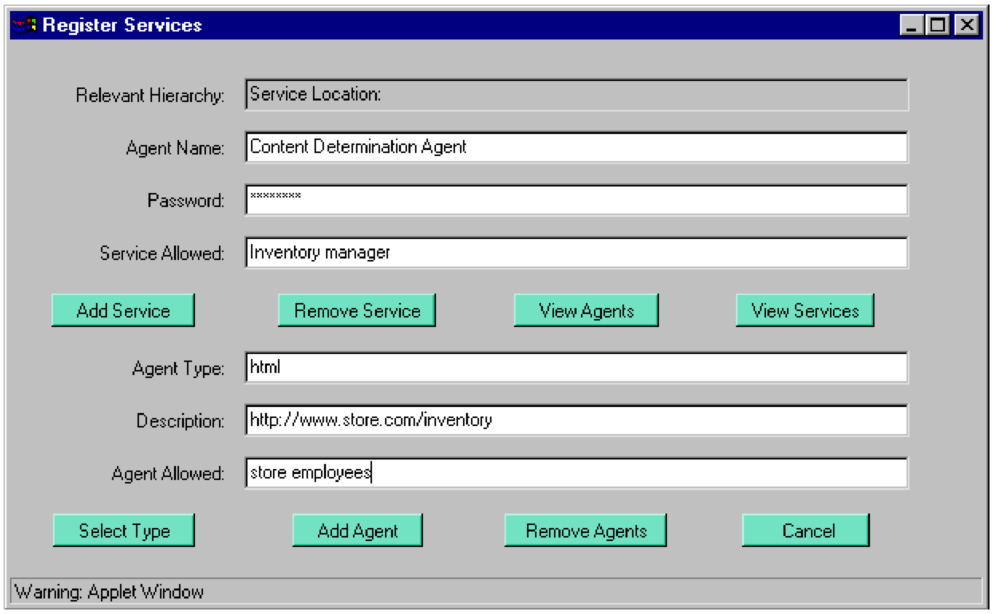

Figure 8. Agent/Service Registration Screen Dump

- **Ontology/Translation Services:** The IMPACT Server contains facilities to provide ontology services and assumptional information. For example, an agent whose data is stored in English may have trouble responding to a query from another agent phrased in German. The aim of the IMPACT ontology server is to allow the latter agent to reformulate the query in terms it can understand.

Figure 8 shows a screen dump of the IMPACT registration server.

6. IMPACT Connections

There are many cases where IMPACT agents need to interact with programs that are not IMPACT agents. Such programs may include database servers, agents built in other agent platforms, web servers, and the like.

In IMPACT, we have built an IMPACT Connections library that allows IMPACT agents to access third party platforms. Some currently implemented examples include IBM Aglets, Oracle servers, and a large variety of Hermes data sources. We are also working on establishing connections via ODBC (Open Database Connectivity), JDBC (Java Database Connectivity), Corba (Common Object Request Broker Architecture), Microsoft's TAPI (Telephony Application Programming Interface), etc.

Regardless of the underlying data server, IMPACT uses a generic interface with methods for establishing/terminating connections, executing code calls over these data sources, and processing the returned responses. Furthermore, IMPACT supports "reflection" methods which retrieve a list of data methods, types, and/or sources associated with a connection (to show what's available). Note that connections are useful for both bringing data in (along with appropriate type conversions) and for sending data out (e.g., for posting to web pages or for sending messages, files, and email).

To establish a connection, IMPACT may need several pieces of information such as URLs, host names, port numbers, file names, user IDs, passwords, etc. Nonetheless, once this connection is defined, agents may use a connection by simply executing a code call. Note that IMPACT can also create connections to internal data sources so functions related to arithmetic, time, message processing, etc. can be accessed via regular code calls as well.

7. Agent Roost

An Agent Roost manages all incoming messages for a set of agents. It behaves like a duty officer since it assigns duties to these agents based on messages it has received and processed. Initially, these agents are "asleep" (i.e., inactive). When one of these agents receives a message, the Agent Roost fills this agent's message box, "wakes" the agent (i.e., launch it if it's not already awake), and lets it run. If this agent sends an outgoing message to another agent managed by the roost (i.e., an *internal* agent), then the roost repeats the process above for this internal agent. Likewise, if the outgoing message is sent to an agent which is not managed by the roost (i.e., an *external* agent), then the roost routes the message to an "appropriate" roost which will then pass this message along to the specified external agent.

To determine which roost is "appropriate", an Agent Roost may contact the IMPACT Server to determine where an external agent is located. If the name of an external agent is known, then its location information will be returned by the Registration Server. Note that several locations may be returned if the external agent is *mirrored*; this allows our agent to choose from several alternate locations if the external agent at some primary location is not responding. Now suppose that our agent only knows the name of an external agent's service (e.g., "predict weather(DC)"). Then the Yellow Pages Server can be used to return the names of all registered external agents which provide this service. After determining the locations for these agents, this information can then be passed back to the Agent Roost.

We have not yet specified how agents can return to "sleep". By default, an agent puts itself back to sleep once it has finished processing its action policy. Alternatively, an agent could decide to stay awake indefinitely or to awaken at regular intervals, e.g., on every Saturday at 4:00am. When an agent is not asleep, it is *active*. We can determine which agents are currently active by observing the Agent Roost GUI (see Figure 9). Here, in the "Agent listing" box, an agent's name will be red if it is currently active. This information, along with data from

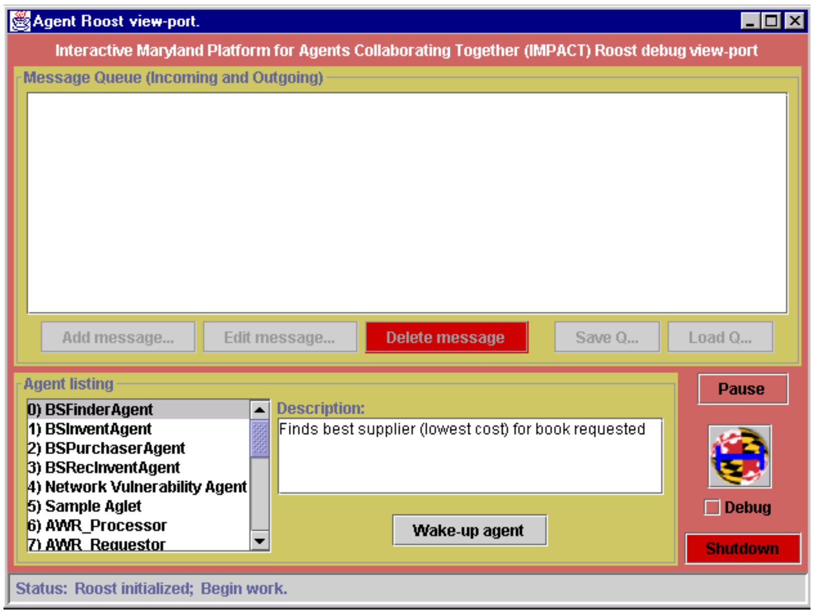

Figure 9. Agent Roost Screen Shot

the Agent Log, helps agent developers understand how their agents are interacting with others.

8. Agent Log

The IMPACT Agent Log behaves like a bulletin board where agents can post messages/information and can read (non-restricted) information posted by other agents. It is a centralized message manager which allows us to maintain a record of inter-agent communication. The log is useful for many purposes including record keeping (which is vital for system security), usage statistics (which are used in performance analysis/optimization), and process visualization (to better understand agent states/messages when constructing and debugging agents).

Entries in the Agent Log have the following format:

```
(Subject, From, To, Date, Time, Contents)
```

Here, `Subject` is the name of the entry (or alternatively, the subject of the agent's message). A string in the `From` field indicates where an entry came from (i.e., from which agent/user/machine, etc.) and a string in the `To` field indicates the target audience for the entry (i.e., which groups of agents/users/etc. would be interested in the entry). Usually, the `Date` and `Time` fields are used to indicate when an entry was added to the Agent Log. Note however that this is not required so dates could pertain to past events (e.g., the time a suspicious user entered the system)

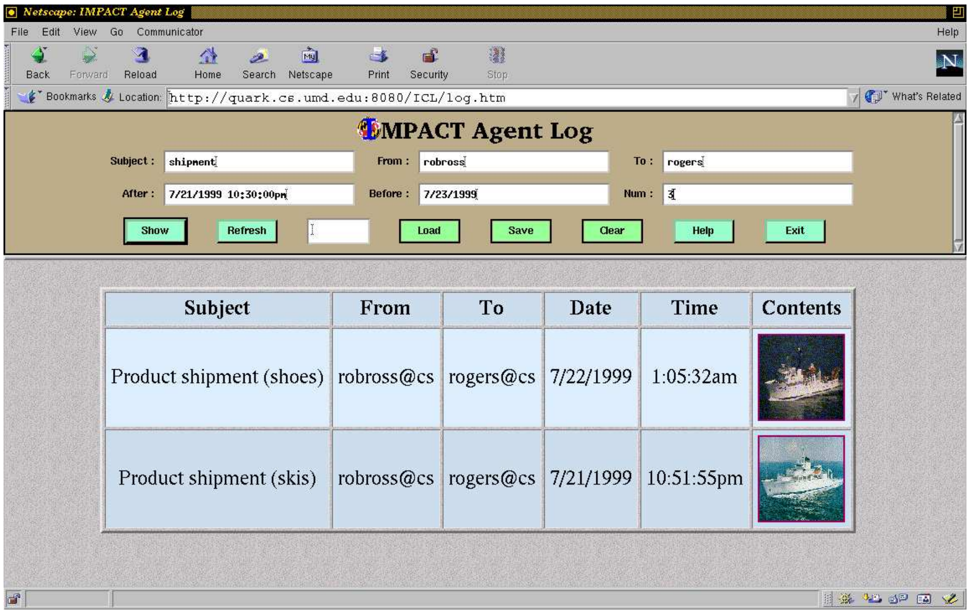

Figure 10. Agent Log Screen Shot

or future intensions (e.g., our agent's prediction of the time when a shipment will reach Chicago). Finally, the `Contents` field holds the contents of the entry and can be any HTML object such as hyperlinks, text, images, movies, sound files, etc.

A sample screen shot of the Agent Log is given in Figure 10. Notice that here, a user has entered field values which filter out unwanted entries from the log's display. Specifically, this user has asked for "at most `3` `shipment`s from `robross` to `rogers` which occurred after `10:30:00pm` on `7/21/1999` but before any time on `7/23/1999`. Agents requesting entries from the log can provide their own instantiations for these filter fields.

By clicking on the `Show` button, all entries currently in the log which match the given criteria will be displayed (and ordered so the most recently received entries are shown first). To automatically update the display every n seconds (where n is the number after the `Refresh` button or a default of "5"), click the `Refresh` button. The `Load` (`Save`) button allows us to store (retrieve) log entries to (from) non-volatile memory, and the `Clear` button allows users with appropriate permissions to remove all entries from the log.

9. Conclusions

There are numerous important agent applications around today, accompanied by relatively few theoretically well grounded *and* practically realizable frameworks for implementing agents and multiagent applications. The IMPACT effort is one attempt at this. There are many other important efforts.

The InfoSleuth project [3] develops a set of agents to support interoperability — these include "user" agents, ontology agents, broker agents, task execution agents and monitor agents and agents communicate with each other in KQML. Likewise, Shoham [11] was perhaps the first to propose an explicit programming language for agents, based on object oriented concepts, and based on the concept of an agent state. In Shoham's approach, an *"agent is an entity whose state is viewed as consisting of mental components such as beliefs, capabilities, choices, and commitments"* [11]. He proposes a language, Agent-0, that provides a mechanism to express actions, time, and obligations. There are several differences between our approach and Shoham's. Our language builds on top of arbitrary data structures, whereas Shoham's language is more or less logical (though it uses a LISP-like syntax). For us, states are instantiations of data structures managed by the program code associated with agents, while for Shoham, the agent state consists of beliefs, capabilities, choices and commitments. This allows Shoham to focus on reasoning about beliefs, whereas our focus is on decision making on top of arbitrary data structures. Clearly both paradigms are needed for successfully building an agent. An excellent overview of a set of agents is provided by Hendler [8] who describes a set of projects funded under DARPA's Control and Agent Basesd Systems program. He classifies agent communication languages as falling into four categories – *performative* which includes `Ask`, `Tell` style primitives describing simple service invocation between agents, `service` languages that need semantics expressed perhaps via classes, relations, etc., *content* based languages that require domain-specific ontologies, and *control add-in* languages that vary from system to system (another excellent overview of agent communication languages is provided by Labrou et. al. [9]). In contrast to all these works, our framework focuses on tools to build an agent on top of existing legacy code so that (i) agents can continuously engage in the cycle of "observe state change — think — act" in accordance with the operating principles of the agent.

Towards this end, we have developed a comprehensive methodology to build agents on top of legacy/specialized software by "adding on" to the code, a description of the data types involves, the application program interface functions involved, the actions the agent may take, the integrity constraints the agent must preserve, the notion of concurrency used by the agent, the action constraints that limit what the agent may do concurrently, and the agent program that encodes under what conditions the agent is permitted/obligated/forbidden to execute an action. We have provided a description of the architecture of the IMPACT system, describing its various components. This paper intentionally focuses on the system, rather than on the underlying mathematical theory — the reader may consult [5, 6] for that.

Acknowledgments

Different parts of this work were supported by the Army Research Office under Grants DAAH-04-95-10174, DAAH-04-96-10297, DAAG-55-97-10047 and DAAH04-96-1-0398, by the Army Research Laboratory under contract number DAAL01-97-K0135, and by an NSF Young Investigator award IRI-93-57756.

References

1. Adali, S., Candan, K.S., Papakonstantinou, Y. and Subrahmanian, V.S., "Query Processing in Distributed Mediated Systems," Proceedings of ACM SIGMOD Conference on Management of Data, Montreal, Canada, June 1996.
2. Arisha, K., Eiter, T., Kraus, S., Ozcan, F., Ross, R. and Subrahmanian, V.S., "IMPACT: Interactive Maryland Platform for Agents Collaborating Together," IEEE Intelligent Systems magazine, March 1999, pp. 64–72.
3. Bayardo, R. et. al. "Infosleuth: Agent-based Semantic Integration of Information in Open and Dynamic Environments," Proc. ACM SIGMOD Conf. on Management of Data, 1997.
4. Dix, J., Subrahmanian, V.S. and Pick, G., "Meta Agent Programs," Journal of Logic Programming, to appear.
5. Eiter, T., Subrahmanian, V.S. and Pick, G.. "Heterogeneous Active Agents, I: Semantics," Artificial Intelligence, March 1999, pps. 179–255.
6. Eiter, T. and Subrahmanian, V.S., "Heterogeneous Active Agents, II: Algorithms and Complexity," Artificial Intelligence, March 1999, pps. 257–307.
7. Subrahmanian, V.S., Bonatti, P., Dix, J., Eiter, T., Kraus, S., Ozcan, F. and Ross, R., "Heterogeneous Agent Systems: Theory and Implementation," MIT Press, 2000, to appear.
8. Hendler, J. "Making sense out of agents," IEEE Intelligent Systems, March 99, pps. 32–37.
9. Labrou, Y., Finin, T. and Peng, Y. "Agent Communication Languages: The Current Landscape," IEEE Intelligent Systems, March 99, pps. 45–52.
10. Lange, D.B. and Osjima, M. "Programming and Deploying Java Mobile Agents with Aglets," Adison Wesley, Massachusetts, USA, 1998.
11. Shoham, Y. "Agent oriented programming," Artificial Intelligence, 60:51–92, 1993.
12. Ullman, J.D. "Principles of Database and Knowledge Base Systems," Computer Science Press, 1989.