ORINCON's Agent Workbench: A Graphical Framework for Building BDI-Based Intelligent Software Agents

Marcus J. Huber [*],
Jaeho Lee, Cherilyn Michaels, Cheryl Zenor, Vivek Samant

ORINCON Corporation
9363 Towne Centre Drive, San Diego, CA 92121
Phone: (619) 455-5530
marcush@home.com
{jaeho,michaels,czenor,samant}@orincon.com

Abstract

Designing, implementing, and debugging agent–based applications can be a long and laborious process. ORINCON's Agent Workbench simplifies and speeds up this process of designing and instantiating applications with one or more intelligent software agents using a collection of graphical tools and underlying representations all implemented in Java for maximum portability. The Agent Workbench graphical user interfaces provide an easy to use integrated graphical and textual programming environment that facilitates functionality development and reuse at multiple abstraction levels (native–code primitive functions, plans, and multi–agent processes) for Belief–Desire–Intention (BDI) based agents. The Agent Workbench also provides facilities for monitoring and controlling the runtime state of executing agents to aid in debugging and refining agent–based applications.

[*]Now with Intelligent Reasoning Systems, 240 Belflora Way, Oceanside CA 92057, http://members.home.net/marcush/irs.html

1 Introduction

The use of intelligent software agents within applications has been grown increasingly popular in the last few years. Multiagent systems, where a number of agents cooperate to perform tasks, have also received considerable attention lately. Our definition of an agent is an autonomous, intelligent software entity that can make decisions and perform actions based on perceived inputs in order to achieve some goals [13]. Agents exhibit a number of useful characteristics, including varying levels of autonomy, reactive and proactive behavior, mobility, communication, and cooperation [15]. Agents also typically have declarative representations for tasks and other attributes, making them much more flexible than more traditional software approaches that use implicit representations. Research related to agent theories and architectures has reached a maturity level at which people are confident enough with the technology to incorporate it into applications outside of purely academic systems.

It is safe to say that a large majority of all agent–based applications to date were constructed using standard software implementation processes and standard software development tools. Although many general development tools can work across a wide range of problems and paradigms, in many cases tools designed specifically for a particular domain can simplify and streamline development. Such is the case with intelligent software agents. Generic software development tools are typically adequate for building agent–based systems, but tools *optimized* for such a task, that address the particular issues and perspectives required for agent–based systems, should result in faster, easier, and more focused development. We have designed and implemented such a specialized tool, which we call the Agent Workbench Environment, (or simply, Agent Workbench) in Java.

The Agent Workbench is a graphical framework and a set of underlying representations designed to facilitate the design, construction, and execution monitoring of single and multi–agent systems. It consists of a Multi–Agent Editor, an Agent Functionality Editor, and an Agent Execution Monitor. It is implemented in Java and therefore inherits all of the benefits associated with Java; in particular, the Agent Workbench should run on any platform that has a Java virtual machine.

The Multi–Agent Editor (MAE) is a graphically–interfaced tool used to create and edit multi–agent process specifications (i.e., tasks requiring more

than one agent). Its underlying representations are being modified to be based upon the IDEF3 process modeling standard [10],[1] which we extended in order to capture information required for instantiating the multiple executable agents required to perform the process. This new scheme provides support for multiple perspectives, or views, into the process model. These views include: the *role view*, which depicts a process from an agent–centric perspective, which highlights which agents are responsible for which tasks within a process; the *activity view*, which depicts a process from a task–centric perspective, which highlights which tasks must be performed, and in which order, to accomplish the objective of the process; the *object view*, which depicts a process from one of the (possibly many) objects being managed within the process, and shows the transformation of the object as the process is executed; and the *schedule view*, which indicates the temporal sequencing of each of the tasks required for the process independent of the role required for each task. We describe the MAE in more detail in Section 2.

The Agent Functionality Editor (AFE) provides a number of graphical tools for specifying agent capabilities at multiple levels of abstraction, from primitive functionality to abstract plans to multi–agent plan specifications. The AFE is closely coupled with the MAE so that process specifications created within the MAE can be further refined within the AFE. The AFE's primary use is for development and refinement of plan specifications for single agents. This typically amounts to specifying one or more goals for the agent, an initial set of beliefs for the agent, and a set of capabilities in the form of one or more plans that the agent can use to achieve its goals. None of these specifications are formally required however, as we recognize that an agent may have the ability to generate plans, sense or infer beliefs, create goals on the fly, etc., or an agent might acquire these from other agents through communication. We describe the AFE in more detail in Section 3.

The Agent Execution Monitor (AEM) provides several graphical user interfaces for observing and changing agent behavior during agent execution. Separate interfaces exist for the beliefs, pending and active goals, and intentions (instantiated, executing plans) of an agent. For example, the interface for the agent's beliefs provides the ability to view, modify, add, and delete beliefs during runtime. Executing agents interface to the AEM dis-

[1]The current implementation was not based upon IDEF standards but does provide fully functional multi–agent process specification capabilities.

plays through sockets, a communication mechanism commonly supported on many operating systems on many hardware platforms. CORBA (Common Object Request Broker Architecture [14]) support is planned for the near future. We describe the AEM in more detail in Section 4.

Before describing the Agent Workbench in more detail, clarification of its intent and scope is in order. The Workbench is designed to accommodate a wide range of agent architectures based around the general Belief/Desire/Intention (BDI) theoretical framework [12]. These, in the categorization of Wooldridge and Jennings [15], are a form of "hard" agent architecture, which have explicit conceptual and/or implementational models of such concepts as knowledge about the world (beliefs), goals (desires), and commitment to action (intentions), among others. That is, the BDI agents constructed with the Agent Workbench have a stronger notion of *agency* than their category of "soft" agents. Such soft agents may exhibit external behavior similar to their more theoretically founded counterparts, but do not have the explicit, usually declarative modeling of such concepts and therefore tend to be more specialized and less flexible and dynamic, with implicit, hard-coded capabilities. The initial implementation of the Agent Workbench currently fully supports the UMPRS agent architecture – the University of Michigan implementation of the Procedural Reasoning System [5, 9] and is easily extensible to alternative BDI agent architectures such as the Java-based Jam agent [4].

2 Multi–Agent Editor

The Multi–Agent Editor (MAE) component of the Agent Workbench models the interactions between autonomous agents that may be engaged in cooperative and non–cooperative tasks. These agents are possibly geographically distributed and intentionally divergent (disparate goals and priorities).

The MAE provides the functionalities to construct and edit multi–agent interactions at multiple levels of abstraction (connectivity, language, protocol, communication mechanism). In addition, the activities specified in the process can be automatically mapped to executable individual agent plans. The individual agent plans also manage interdependencies between activities such as adequate sequencing and synchronization of activities to coordinately accomplish the goals identified in the multi–agent process.

The representational and functional requirements for the MAE are similar to that found in the field of workflow [3, 6] and we attempt to leverage the insights and previous work from this work and such standard representations as IDEF3 [10] as much as possible.

A workflow management system is an active system that manages the flow of business processes performed by multiple persons. A comprehensive definition of workflow management is given by Hales and Lavery [2] as follows,

> Workflow management software is a proactive computer system which manages the flow of work among participants, according to a defined procedure consisting of a number of tasks. It co–ordinates users and system participants, together with the appropriate data resources, which may be accessible directly by the system or off–line, to achieve defined objectives by set deadlines. The coordination involves passing tasks from participant to participant in correct sequence, ensuring that all fulfill their required contributions, taking default actions when necessary.

In other words, workflow management can be considered as a specification coordination problem, and thus we find closely related issues with multi–agent systems such as coordinated activity and resource sharing between agents.

The current implementation was not based upon IDEF3 standards but provides fully functional multi–agent process specification capabilities (See Figure 1). In this implementation, the MAE graphically depicts a multi–agent interaction model represented in a declarative manner and results in workflow execution procedures that are distributed among the agents possessing roles in the interactions specified in the interaction specification.

Our underlying representations are being modified to be based upon the IDEF3 process modeling standard, which we extended in order to capture information required for instantiating the multiple executable agents required to perform the process. IDEF3 is designed to help document and analyze the process of an existing or proposed system. Proven guidelines and a language for information capture help users represent and organize process information for multiple downstream uses [10]. By switching to the IDEF3 process modeling standard, we can take advantage of established *knowledge engineering methodology* and existing extensive *process libraries.*

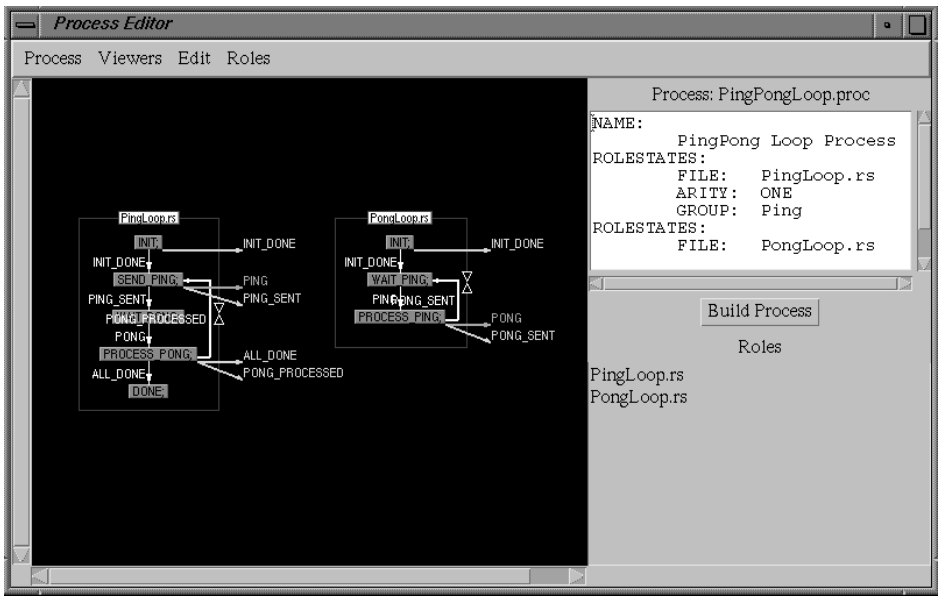

Figure 1: An example of the current Multi–Agent Editor display. This process consists of two roles: one role sends "ping" messages to the agent fulfilling the other role; the second role receives "ping" messages and sends "pong" replies to the agent fulfilling the first role.

The MAE supports multiple *views* of the multi–agent process that the user can switch between to get different insights into the process. The *activity view* and the *object view* corresponds to the *process-centered view* and the *object-centered view* of IDEF3, respectively. Activity views focus on assertions about the processes that occur and their ordering. Object views, on the other hand, focus on a participating object or set of objects.

In addition, the MAE supports the *role view* and the *schedule view*. The role view depicts a process from an agent–centric perspective, which highlights which agents are responsible for which tasks within a process. The schedule view indicates the temporal sequencing of each of the tasks required for the process independent of the role required for each task.

Activity View

The activity view helps users to represent knowledge about events and activities, the objects that participate in those occurrences, and the constraining relations that govern the behavior of an occurrence. The activity view, or task view, in the MAE is a graphical layout explicitly indicating each of the tasks required for the process independent of the role required for each task. The graphical elements, adopted from IDEF3 process representations, include Unit of Behavior (UOB) boxes, links, and junctions.

A UOB is a distinguishable packet of information about an event, decision, act, or process. Links specify relationships (constraints) between UOBs. Precedence links express temporal precedence relations between instances of one UOB and those of another. Relational links (dashed links) carry no predefined semantics, thus users can use it to highlight the existence of any relationship between two UOBs.

A junction is a point in the process flow where a process flow path branches into multiple paths (fan-out junctions), or multiple process flow paths merge into one (fan-in junctions). Junctions thus describe the flow logic of the process.

In the MAE, UOBs and links also have additional attributes to support other views. The UMPRS agents that we can build with this model necessarily are limited to a subset of junctions however. Specifically, instantiable agents support AND and OR fan-in and XOR and synchronous AND and OR fan-out. UOBs and links have attributes associated with them that are not standard IDEF3. For UOBs, these may include (but are not limited to):

Label: a unique ID

Subprocess specification: label or other unique identifier of a sub–graph that specifies how to accomplish the UOBs action.

Cost: Numeric value or function that specifies how expensive the action is.

Failure condition: situations in which this action fails.

Duration: how long it takes to perform that action. This may be something like a constant, a min/max range, or a function.

Role(s): one or more labels specifying either a requirement of the "services" that the performer of this action must have in order to do it, or the label applied to a performer of this action.

Annotation: miscellaneous domain/application–specific information.

Links between nodes have conditions specified for them that represent when the transition between nodes will occur. The link condition representation may include (but is not limited to) the following relationships:

Temporal transitions: simple sequencing of actions.

Event transitions: messages or other types of a temporal occurrence.

Arity constraints: how many of a particular event, message, recipient or source, etc. need to be involved.

Resource constraints: requirements of resources. The idea of how this representational information is transformed into actual agent–specific code is that each UOB (and failure block) corresponds to an agent operator (plan) and that the conditions on the links represent preconditioning contextual information on when the successive states are executed.

Object view

An object state in the IDEF3 process method is any physical or conceptual thing that is recognized and referred to by participants in the domain as a part of their description of what happens in their domain. The object view depicts information about how objects of various kinds are transformed into other kinds of things through a process, how objects of a given kind change states through a process, or context-setting information about important relations among objects in a process.

In the MAE, the object view is displayed using a graphical layout explicitly indicating the states one particular object takes throughout execution of a process. Each object involved in a process would have a view from its own perspective.

Role view

The role view is a graphical layout explicitly indicating each of the tasks for each role involved in the process, independent of which agent (software or human) actually performs the role. A more specific role view, where individual role interactions (e.g., messages, signals), and state transitions and conditions can be viewed and modified.

Schedule view

The schedule view is a graphical layout explicitly indicating the temporal sequencing of each of the tasks required for the process independent of the role required for each task.

3 Agent Functionality Editor

The Agent Workbench's Agent Functionality Editor (AFE) is used to specify an individual agent's capabilities, goals, beliefs, and other attributes. This specification can be performed at several levels of abstraction, ranging from definition of low–level primitive functionality (e.g. C/C++ functions) to specification of one or more predefined process roles that the agent will need to be capable of fulfilling, and the libraries of primitive functionality it will need to perform those roles.

At the highest level, an Agent Workbench agent consists of roles and libraries. Roles are specifications of behavior, often the behavior needed to play a part in a particular process. Each role has capabilities, goals, and beliefs associated with it, and an agent needs only one role in order to function. The Agent Functionality Editor, however, supports the combination of more than one role in a single executable agent, allowing an individual agent to simultaneously support a number of different and possibly unrelated behaviors. For example, a single agent may play more than one automatically generated role in a process, or a role specifying some particular useful behavior, such as the ability to report on internal state to some monitoring process, may be added to an agent which already has a role specifying its primary behavior. Roles are stored as separate files, organized by area of functionality, making it easy to locate and reuse useful roles when automatically or manually defining a new agent. The other high level components

of Agent Workbench agents are libraries. Libraries are collections of primitive functionality, and, like roles, can be quickly added to agents. Libraries can contain any sort of primitive functionality, though libraries supporting inter-agent communication capabilities are especially useful.

The Agent Editor, shown in Figure 2, is used to specify an agent at this high level. The agent developer simply uses pull–down menus and browsers to specify the roles the agent is to perform, and the libraries of primitive functionality to be made available to it. With this information, an executable instance of the agent, complete with a makefile for linking the generated primitive function code to the main agent engine, is automatically generated. The Agent Workbench is designed to accommodate multiple agent architectures in a modular fashion, so that an instance of the agent in any of a number of agent architectures will be able to be generated in this fashion. At the moment, however, only the UMPRS agent architecture is supported.

Figure 3 shows the AFE's Role Editor being used to define a particular role within an agent. The role-specific beliefs, goals, and plans (which we also call *operators*) can be specified by the agent developer here, although they are often predefined or automatically generated. Like roles and libraries at the agent level, plans can be easily stored individually and the later reused in new agents that need the same functionality.

Figure 4 shows the Operator Editor with a fairly complex plan under development. The left half of the figure shows the graphical representation of the plan while the right half of the figure shows a textual representation. The textual portion of this editor is in the syntax of what we call the Agent Description Language (ADL), an amalgam of plan constructs and attributes from the representations of the Procedural Reasoning System of Georgeff, Rao, et al. [1] (more specifically, from the University of Michigan implementation, [5]), SRI's ACT plan interlingua [11], and the Structured Circuit Semantics (SCS) representation of Lee and Durfee [8, 7]. ADL is explained in more detail below.

Within the Operator Editor of the AFE, modifications to the graphical portion of the window results in automatic revisions of the text portion, while modifications to the text portion results in revision of the graphics representation when the "Create Graph from Text" button (lower right corner) is pressed by the developer. The developer can therefore use either graphical drag–and–drop procedures, text editing, or a combination of the two to define a plan.

The AFE also greatly facilitates development of domain–specific primitive functions through use of the Primitive Library Editor, shown in Figure 5. A primitive function library contains a collection of native–code functions with a CORBA Interface Definition Language–like argument specification for each function. Along with this are specified system– and architecture–specific information, such as the C++ include and load library paths to be used by the system in automatically generating a makefile to build the library. Based on the IDL–like specification, the Agent Functionality Editor automatically generates code specific to an agent architecture, such as the "wrapper" code required to interface between UMPRS representations and standard C++ representations. This automation makes it very easy to create new primitive functions or change the IDL of existing ones. The non-wrapper contents of the primitive functions are defined in a separate Primitive Function Editor. This component of the AFE is also agent–architecture specific in that the generated implementation code must be in a form compatible with a particular target agent architecture. However, the Primitive Library Editor has been designed to accommodate analogous functionality for other agent architectures.

A useful area for primitive definition is the communications domain. We have defined libraries of primitives to support inter-agent communication using plain sockets, TCX (a socket abstraction library from Carnegie Mellon University), and CORBA. In the case of CORBA, we have implemented the ability to automatically generate primitive libraries from CORBA IDL, thus providing automatic binding between Agent Workbench agents and CORBA. At the moment, these bindings have been completed for only a small subset of the total CORBA IDL standard, but already they make it possible for the developer to choose a (simple) CORBA IDL file, generate a library from it, and then use the generated library primitives in agent plans, thus creating a CORBA-enabled agent without the need to manually write primitive function wrappers for each CORBA method.

The Agent Description Language (ADL) used to model agents within the Agent Workbench is an amalgam of plan constructs and attributes from the representations of the University of Michigan implementation of the Procedural Reasoning System [5, 9], SRI's ACT plan interlingua [11], and the relatively recent Structured Circuit Semantics (SCS) representation of Lee and Durfee [8, 7]. The ADL representation for goals, beliefs, and plan libraries are modeled identically to UMPRS; these areas require substantial

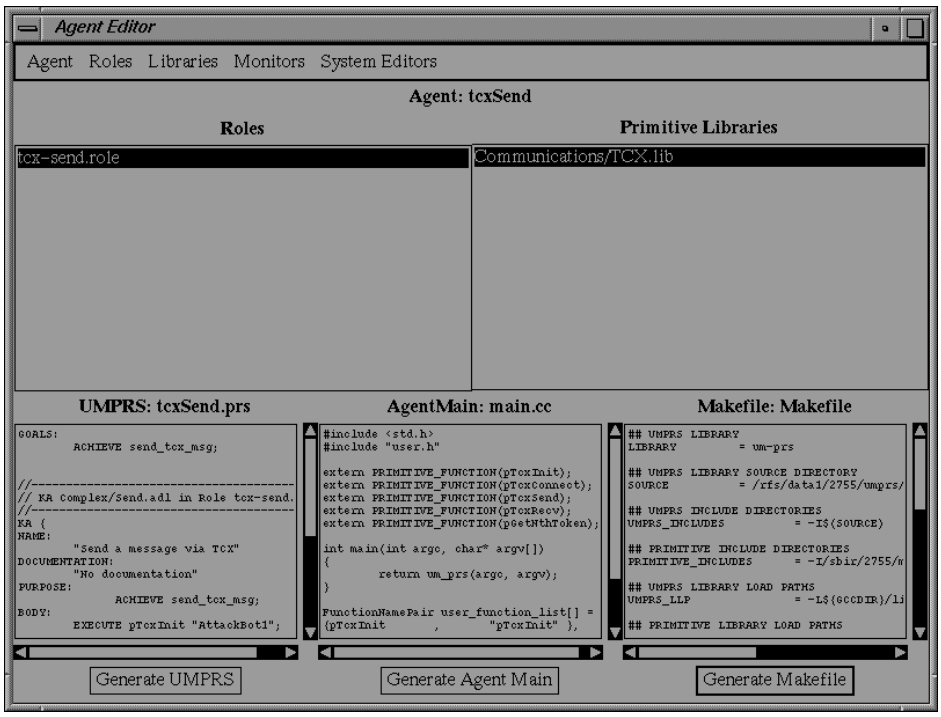

Figure 2: Example of the Agent Functionality Editor's Agent Editor interface. An agent developer only needs to specify roles and primitive function libraries to build a complete agent. A great deal of the details (bottom panels) of agent construction is taken care of automatically.

effort to generalize them to support alternative models. A goal is defined to specify a state of the world to achieve and is represented simply as a relation name with arguments. Facts in the world are represented as propositional relations, as mentioned above. A plan library (a common, but not universal, agent attribute) is simply a list of operators (plans) that represents a goal hierarchy, with one or more procedural methods for achieving each of the goals and subgoals.

An ADL plan contains the following elements, some of which are optional: a unique string identifier; a goal specification indicating the goal state to be achieved by the plan; pre–, post–, and runtime–conditions; a utility function which calculates the overall utility of the plan; resource requirements; a procedural specification for how to achieve the plans' goal; and a procedural

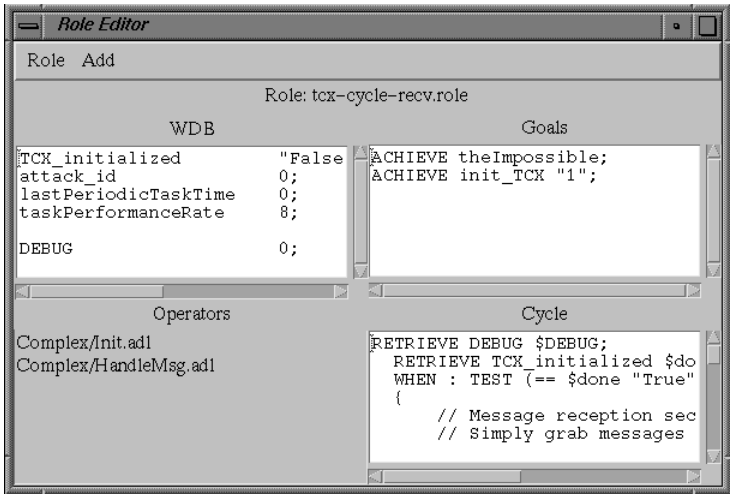

Figure 3: Example of the Agent Functionality Editor's Role Editor display.

specification for how to deal with plan failure. The Operator Editor supports a number of simple and complex plan constructs that can be included in the plan and failure procedural specifications. These include: subgoaling, invoking native–code primitive functions; actions for adding, removing, and modifying the agent's beliefs and top-level goals; conditional branching; iterating; specification of atomic, non–interruptible procedure sections; and parallel and non–deterministic execution.

4 Agent Execution Monitor

The Agent Execution Monitor (AEM) provides the capability to monitor and control executing agents. This is a particularly useful ability during development and debugging of a multi–agent system, where each agent can be a distinct process running on a different machine and where the agents might even move from machine to machine over time. The AEM is a set of graphical tools for viewing and editing the beliefs, desires, and intentions (executing plans) for distributed agents. The AEM displays are relatively simple and the information is displayed primarily in a textual manner. Editing, adding, and removing entries within the displayed information is supported through graphical means (buttons, edit windows, etc.). The Goals Monitor display,

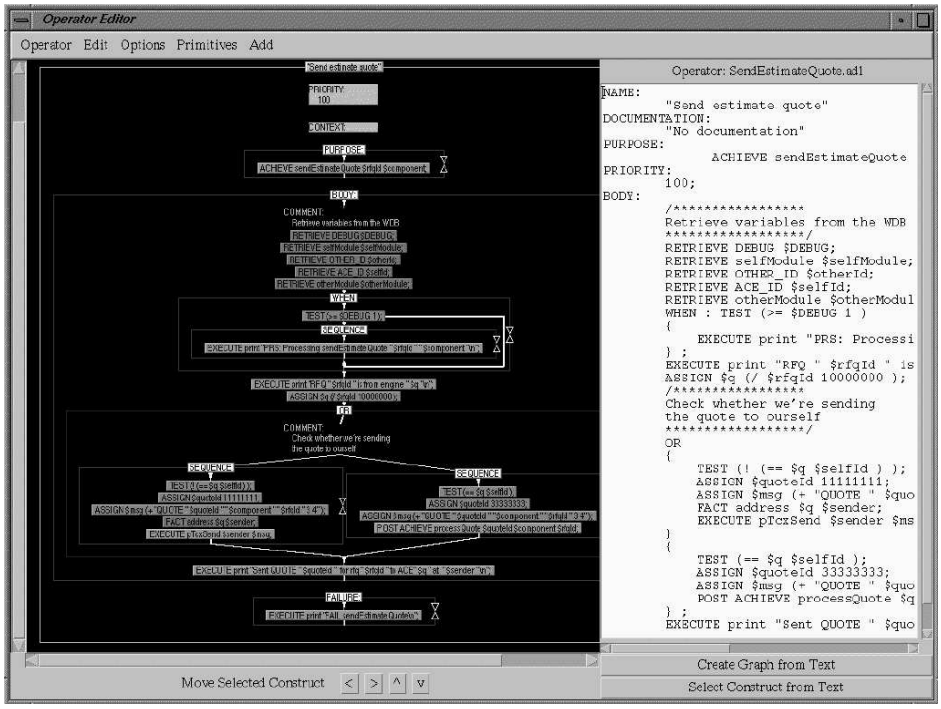

Figure 4: Example of the Agent Functionality Editor's integrated text and graphics Operator Editor display.

WDB Monitor (for World Data-Base) display, and the Intentions Monitor display are shown in Figure 7, Figure 6, and Figure 8, respectively.

The current interface for the Goals Monitor provides the ability to view, modify, add, and delete an agent's goals. For example, one of the agent's goals can be removed simply by highlighting the desired entry and then pushing the `Remove` button. The current WDB Monitor interface provides the ability to view, modify, add, and delete the agent's beliefs, with similar simple interaction support. And, the current Intentions Monitor interface currently provides only the ability to view which plans the agent is currently executing.

Executing agents interface to the AEM displays through standard sockets. This communication capability can be added to the agent simply by specifying the `Reporter` role within the `Agent Monitoring` process within the MAE. The examples shown in Figures 7-8 demonstrate a C++ agent

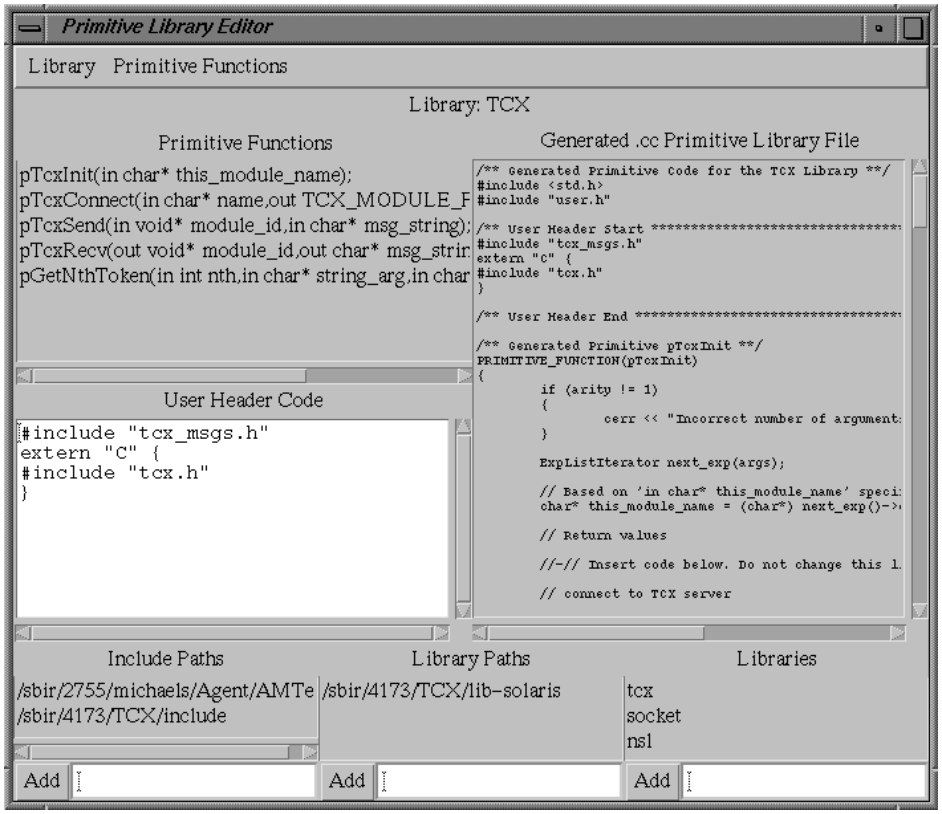

Figure 5: Example of the Agent Functionality Editor's Primitive Library Editor display, where the developer can specify an API's interface, implementation, and library–related agent building parameters.

based on UMPRS communicating to the displays. The agents created by the Agent Workbench do not rely upon the AEM's displays to be present and active, and so may operate with or without the Agent Workbench components in their environment. When using sockets, the agents look for the displays at a known location (host and port) when they first start executing. CORBA–based communication between agents and the AEM displays is planned but, since it requires agent developers to have access to an Object Request Broker (ORB), was deferred in favor of the more widespread socket mechanism.

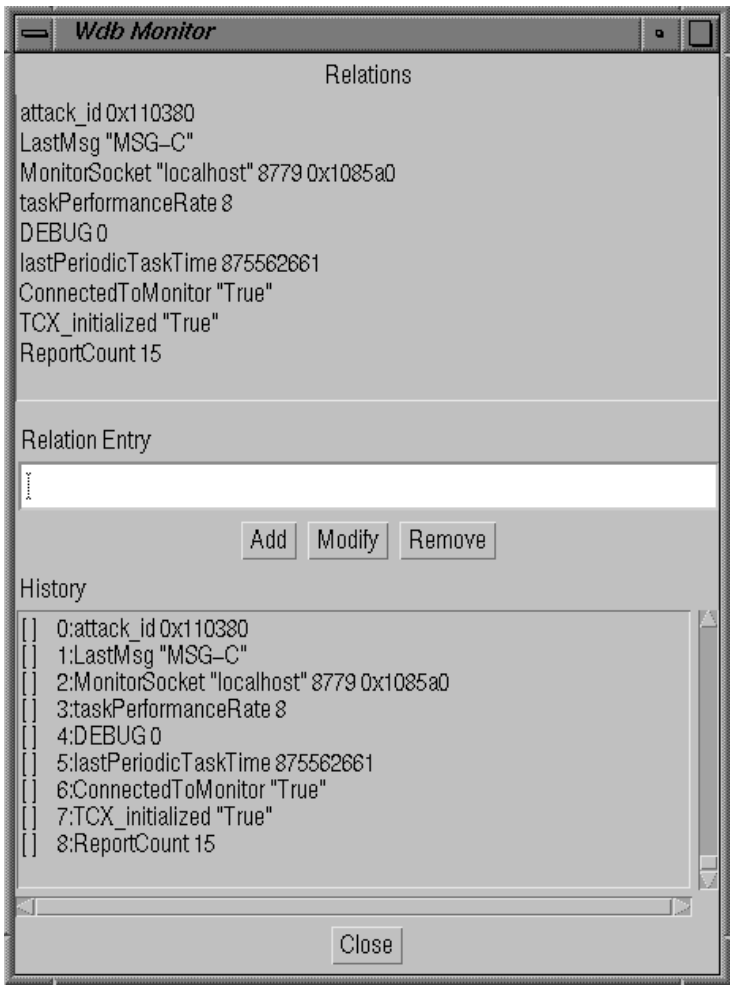

Figure 6: Example of the Agent Execution Monitor's facts display.

5 Summary and Future Work

The Agent Workbench represents a tremendous improvement in the ability to develop and deliver agent–based applications. The Agent Functionality Editor and Multi–Agent Editor provide easy to use, intuitive, graphical interfaces for specifying agents at multiple levels of abstraction. These tools greatly reduce an agent developer's burden by automating many of the more mundane tasks, leaving more time for more important high–level issues. Once

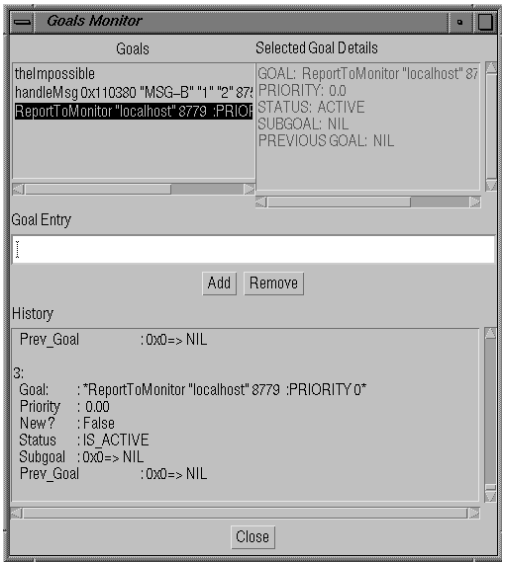

Figure 7: Example of the Agent Execution Monitor's active and pending goal list display.

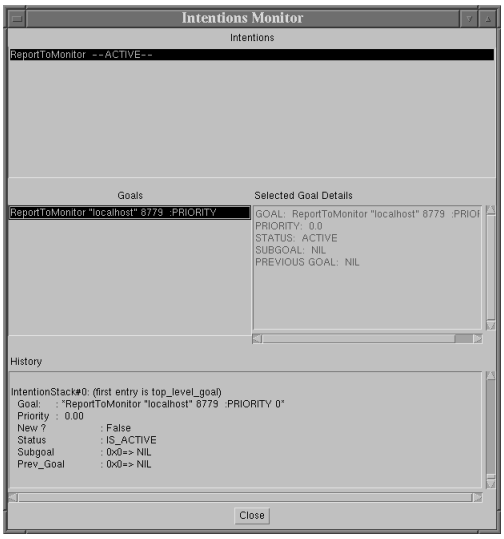

Figure 8: Example of the Agent Execution Monitor's intentions display.

low–level details of agents have been created the first time, an agent developer can simply browse through and select functionality and behaviors. The Agent Execution Monitor provides basic but highly useful runtime interfaces for single and distributed multi–agent system evaluation, debugging, and refinement. The Agent Workbench is also extremely portable due to its implementation using Java.

The Agent Workbench is complete and fully functional and the people using it recognize distinct advantages over their previous, more general software development tools and processes. The Agent Workbench does not have a lot of maturity at this point, however, and there are undoubtedly a number of areas where the Workbench needs to be changed to be more robust and user friendly.

More fundamental research in such areas as single and multiagent plan representations, domain knowledge representations, and agent architectures would all benefit the Workbench. Extension and development of the current Workbench representations and interfaces to accommodate other agent architectures (other than UMPRS), agent frameworks (other than BDI–based

systems), capability representations (other than simple IDL specifications of primitive actions), and process models (other than IDEF) would also be interesting and highly useful.

References

[1] Michael P. Georgeff and Amy L. Lansky. Procedural knowledge. *Proceedings of the IEEE Special Issue on Knowledge Representation*, 74(10):1383–1398, October 1986.

[2] K. Hales and M. Lavery. *Workflow Management Software: the Business Opportunity*. Ovum Ltd., London, UK, 1991.

[3] David Hollingsworth. The workflow reference model. Technical Report TC00-1003, The Workflow Management Coalition, Brussels, Belgium, November 1994.

[4] Marcus J. Huber and Jaeho Lee. The Jam! BDI Agent Architecture. http://members.home.net/irs.html, 1997.

[5] Marcus J. Huber, Jaeho Lee, Patrick Kenny, and Edmund H. Durfee. *UM-PRS Programmer and User Guide*. The University of Michigan, 1101 Beal Avenue, Ann Arbor MI 48109, Oct 1993.

[6] Stefan Jablonski and Christoph Bussler. *Workflow Management: Modeling Concepts, Architecture and Implementation*. International Thomson Computer Press, 1996.

[7] Jaeho Lee. *An Explicit Semantics for Coordinated Multiagent Plan Execution*. PhD thesis, University of Michigan, Ann Arbor, Michigan, January 1997.

[8] Jaeho Lee and Edmund H. Durfee. Structured circuit semantics for reactive plan execution systems. In *Proceedings of the Twelth National Conference on Artificial Intelligence*, pages 1232–1237, Seattle, Washington, July 1994.

[9] Jaeho Lee, Marcus J. Huber, Patrick G. Kenny, and Edmund H. Durfee. UM-PRS: An implementation of the procedural reasoning system for

multirobot applications. In *Conference on Intelligent Robotics in Field, Factory, Service, and Space*, pages 842–849, Houston, TX, March 1994. American Institute of Aeronautics and Astronautics.

[10] Richard J. Mayer, Christopher P. Menzel, Michael K. Painter, Paula S. deWitte, Thomas Blinn, and Benjamin Perakath. Information integration for concurrent engineering (IICE) IDEF3 process description capture method report. Technical Report AL-TR-1995-XXXX, Knowledge Based Systems, Incorporated, September 1995.

[11] Karen L. Myers and David E. Wilkins. *The Act Formalism.* Artificial Intelligence Center, SRI International, Menlo Park, CA, version 2.1 edition, May 1997.

[12] A. S. Rao and M. P. Georgeff. Modeling rational agents within a BDI-architecture. In J. Allen, R. Fikes, and E. Sandewall, editors, *Proceedings of the Second International Conference on Principles of Knowledge Representation and Reasoning.* Morgan Kaufmann Publishers, San Mateo, CA, 1991.

[13] S. J. Russell and P. Norvig. *Artificial Intelligence. A Modern Approach.* Prentice-Hall, Englewood Cliffs, NJ, 1995.

[14] Jon Siegel. *CORBA fundamentals and programming.* John Wiley & Sons, 1996.

[15] Michael Wooldridge and Nicholas R. Jennings, editors. *Intelligent Agents — Theories, Architectures, and Languages*, volume 890 of *Lecture Notes in Artificial Intelligence.* Springer-Verlag, 1995.