A Case Study on the Evolution of Software Tools Selection and Development in a Large-scale Multiagent System

Eric Glover[†], Sunju Park[†], Anil Arora[†], Daniel L. Kiskis[‡], Edmund H. Durfee[†]

[†]Artificial Intelligence Laboratory, University of Michigan
Ann Arbor, Michigan 48109-2110, USA
{compuman, boxenju, aarora, durfee}@umich.edu

[‡]School of Information, University of Michigan
Ann Arbor, Michigan 48109-1205, USA
dlk@umich.edu

Abstract

The University of Michigan Digital Library (UMDL) is an open, evolving multiagent system, currently consisting of over a dozen different types of agents, that serves as a production system to real users and as a testbed for research ideas. Our open, evolving system requires a development environment that is also open and evolving. This paper explores the co-evolution of our system and the underlying tools.

We divide the agent development environment into two evolving layers—a layer that is supported by the UMDL system developers, called agentware, and a layer that is the set of tools used by agent designers, which falls outside the agentware, called otherware. The agentware layer facilitates the ease of agent design and integration by providing an agent shell, support for agent communication, and other useful libraries and classes. The otherware layer achieves diverse agent capabilities by allowing individual agent designers to use a variety of existing and newly developed tools. Furthermore, the division between the two layers is continually evolving as new types of agents are created in the UMDL. This division balances design flexibility and integration: agent designers can make local decisions to use whatever tools are available, and have a means to easily integrate new agents and services into a larger society of agents.

Introduction

In large-scale multiagent systems, there exists a spectrum of system requirements. This spectrum has two extremes, with highly specified requirements at one end and less specified, more open requirements at the other. At the first extreme are systems that start with a specific set of services and limit the types of agents and their interaction to support those services in the system architecture. The other extreme is a system that permits a variety of (including some yet unknown) services. In such a system, the system architecture cannot be fully specified without running the risk of precluding certain classes of services.

The University of Michigan Digital Library (UMDL) falls at this latter end of the spectrum. The UMDL is a large-scale multiagent system where agents represent a wide-variety of users, information sources, and services [1, 4, 13]. One of our goals when building the UMDL is to support diversity in terms of both the capabilities of agents and the technologies used to implement them. As we cannot anticipate all the technologies needed to provide useful information services (or even what services will be deemed useful in the future), our approach has been to provide an open, evolving architecture that can continually reconfigure itself as users, contents, and services come and go.

The open architecture, however, poses a challenge in balancing between design flexibility and design integration. In an open system, agent designers tend to have more freedom to choose the tools and technologies when developing their agents. With this freedom comes the challenge of ensuring interoperability with existing agents. To balance this tradeoff, we divide the agent development environment into two, evolving layers: a layer supported by the UMDL system developers (called agentware), and a layer of tools that are not available in the agentware but used in developing at least one agent (called otherware).

The agentware layer aims to achieve ease of agent design and integration. Tools at this layer provide common support for agent communication, agent design (e.g., agent shell), common functionality, and example agents for testing and verification. The otherware layer aims to facilitate design flexibility by allowing individual agent developers to use a variety of existing and newly-developed tools to create their own agents or services. Given the diversity of agent capabilities that are (and will be) in the UMDL, we believe that the design decisions including tool and component selections at the otherware layer should be almost entirely made by the agent designers, not predefined by the UMDL. The tools at the otherware layer may not necessarily be supported by the UMDL system developers, but by opening the otherware layer to any technology possible, we are able to accelerate the incremental growth of the UMDL.

Our purpose in dividing the tools into two layers is to support independent agent development, while preserving modularity and interoperability. Through standardization and software reuse, the tools at the agentware layer are able to significantly reduce both the design and integration time of an agent. The tools at the otherware layer support many different agent capabilities, such as ontology service and query planning, by using any technologies deemed

appropriate. The division between two layers is not fixed. For example, the agentware layer is continually evolving to accommodate widely accepted tools from the otherware layer. Similarly, a new set of tools may become a part of otherware as new agents and services are created.

Throughout the duration of our project, our tools have evolved to encourage and facilitate (1) creation of new instances of existing agents, (2) creation of wholly new types of agents, and (3) interoperability among agents. As a result, the UMDL contains many integrated but diverse services, each one independently designed and implemented, while retaining a high level of intra-agent activities.

In the following, we describe the UMDL system, the agent architecture, and the agent development environment. Then we examine the evolution of the UMDL and how the tools and agent components from both layers are used and evolved accordingly. Finally, we conclude with a summary of the lessons we have learned.

The UMDL System and Agent Architecture

In this section, we briefly describe the UMDL system and its agent architecture [1, 4, 10].

The UMDL is designed to provide digital library services in a distributed, heterogeneous information environment. It is structured as a collection of agents that collectively provide these services by dynamically configuring their interactions as needed. Higher level activities such as team formation are accomplished via intra-agent communication protocols.

We distinguish three broad classes of agents populating the UMDL: User Interface Agents (UIAs), Collection Interface Agents (CIAs), and Mediator Agents. The UIAs manage the presentation of information, maintain user models, and in general represent the end users of the digital library. The CIAs provide access and search services for the library collections, and represent the publishers and other owners of these collections. In between, many types of mediator agents perform a variety of services (e.g., monitoring the progress of the query, transmitting the results of a query, providing the ontology service, etc.). Among the mediator agents are specialized system agents called facilitators, such as the registry agent (that supports the location of relevant agents) and the auction agents (that facilitate information markets). As new services are desired, new types of agents are created to provide the services and added to the system.

Agents are implemented as Unix processes that communicate using common network protocols (i.e., TCP/IP). We choose distributed-object technology as the basis for agent communication; we use the CORBA (Common Object Request Brokerage Architecture) distributed-object standard [3], and in particular, Xerox's ILU (Inter-Language Unification) implementation of it [9]. The CORBA standard provides us with the basic mechanisms we need, and ILU is freely available, allowing us to take advantage of CORBA with little investment. Agent communication tools define for each agent a communication interface, embodied as an ILU object that handles communication with other agents' communication objects.

The interface exported by the communication objects implements KQML (Knowledge Query and Manipulation Language) performatives and their (possibly nested) argument lists [5]. KQML, being neutral to details of the "content" argument that varies with agents and services, makes it well suited for supporting the diverse uses of the UMDL.

To facilitate the agent design, we provide an agent shell—a C++ Agent class. The Agent class encapsulates the communication mechanisms. Individual agents are implemented by subclassing this class, thus inheriting the ability to communicate through our standard mechanisms.

For convenience, we have added to our Agent class some functionality tailored to particular services. For example, since a required capability of a UMDL agent is that it must register with the UMDL registry so that other agents can find it, we have embedded this functionality into our Agent class, thus relieving the duplicated programming of registration protocol for every agent created. Of course, any special functionality not inherent to the UMDL architectural requirements can easily be removed or ignored for different agents.

For each specific functionality of an agent, the agent designers may introduce additional components to the ones that are supported by the UMDL system developers. The UMDL places no restriction on the tools and components used at the otherware layer.

Agent Development Environment: Agentware and Otherware

In this section, we discuss the tools of the agentware and otherware layers and present an example of how an agent is developed. Agentware is the collection of various components and tools that are supported by the UMDL developers. Otherware is our name for third-party components and tools that are not integrated into the agentware, but have been involved in the building of at least one agent. In addition to the agentware and otherware, there are a number of basic tools, consisting of the compilation and basic libraries that provide the base for development. One such tool is C++, our primary programming language. We take advantage of C++'s object-oriented features to build the agent classes, as discussed in the next subsection. We also use the RogueWave Tools.h++ library [12]. The Tools.h++ library provides useful C++ classes such as a string class, container classes (e.g. vectors, hash tables) in both generic (Smalltalk-like) and template forms.

Agentware

The agentware is a collection of various component libraries and tools that are supported as a base for UMDL agents. The primary component of the agentware is the Agent class. This C++ class defines a set of

communication APIs (Application Programmer's Interfaces) for sending and receiving messages as well as encapsulating the underlying CORBA communication mechanisms. All agents subclass the Agent class and inherit these standard communication mechanisms. The agent class defines a set of KQML APIs for sending and receiving KQML messages. Each KQML performative has a unique send function and a unique receive virtual function. Each receive virtual function defaults to responding with "sorry" to indicate that the performative is not supported by the agent. To receive certain performatives, therefore, the agent designer overrides the appropriate virtual functions.

Along with the Agent class, the agentware has component libraries that are not required by the agents but provide useful functionality. One such library is the Agent Toolkit library. The Agent Toolkit library is a repository of C++ classes that are basic utility classes not found in the RogueWave or in the base agent classes. It contains various classes such as a thread class, a mutual exclusion lock, a semaphore class, and a class to register callback functions to respond to inputs on specified file descriptors.

The agentware also consists of support tools that facilitate agent development. An example is the Testing Client who reads in a file containing messages, and via a command line interface, allows agent developers to send those messages to a specified agent and to display the replies that the agent returns. The Testing Client provides agent developers with a way to verify the interoperability with other agents at the development phase.

The agentware also contains code that provides integration with the otherware. One example is the code that integrates the University of Michigan Procedural Reasoning System (UM-PRS) [7, 11] with the Agent class. This integration code was incorporated into the agentware so that other developers could take advantage of UM-PRS without rewriting that integration code.

Otherware

While the agentware provides a rich set of tools for building agents, it is often desirable to incorporate third-party software into an agent. It is this software that we refer to as otherware. A number of different software packages have been used in UMDL agents. We mentioned UM-PRS as one. Another example is Loom [2], a description logic classification system used in the Service Classification Agent (SCA) [15]. Additional otherware includes client libraries for Sybase, Z39.50 [8], and the FTL (Full Text Lexicographer) [6] search engine.

Developing an Agent

As an example, we examine how the SCA was developed. The SCA maintains a dynamic ontology of agent capabilities that allows for automatic classification of services within the UMDL. Instead of building a description logic classification system from scratch, the developer of the SCA decided to use Loom, the most expressive of the KL-ONE systems.

Integration of Loom with the existing Agent class, however, posed a challenge because Loom is written in Lisp[1] and the Agent class is implemented in C++. The solution was to build a driver that spawns two child processes, one the agent and one Allegro Lisp, which would communicate via a pipe. The agent translates messages received via the KQML API into messages that it then ships through the pipe. The Lisp process receives the data from the pipe and performs the classification service. The resulting classification is then sent back though the pipe and packaged up by the agent to reply to the original sender.

As in the example, there are tradeoffs in our two-layered development environment: the SCA developer was able to use the power of Loom, but needed extra work on integration.

Evolution of the UMDL and the Agentware

In this section, we examine the evolution of the UMDL, and the associated agentware used to create agents. Our original objective of an open architecture to facilitate integration of a variety of different services was quite ambitious. Nearly four years since the first production system, there are over a dozen different types of agents and dozens of different instances, each with unique services. This large set of services evolved over time. Paralleling this evolution was an improvement in the supported agentware available to agent designers.

The early versions of the agentware provided only the bare minimum set of libraries necessary for intra-agent communication. The basic KQML and lower level (CORBA-ILU) code are the only agent components mandated by the architecture. Over time, the agentware has grown to include some special purpose libraries, such as the ability to register, extra support for asynchronous communication through mailboxes, the ability to fetch web pages, and numerous others. The expansion of the agentware is one of many software related "evolutions" occurring within our project.

The collection of software libraries, or agentware, is our agent design toolkit. Unlike a monolithic toolkit, our agent designers are not restricted to using code and functionality found only in the agentware. Likewise, the agentware itself is not limited to a pre-defined set of architectural specifications. Over time, our agentware has expanded to fill the needs of a monolithic toolkit, without removing the individual designer's flexibility necessary to enable a cornucopia of services. These needs include the ability to quickly prototype an agent, and ease of integration with existing agents.

The growth of our system has demonstrated many simultaneous evolutions. First, is the gradual expansion of the agentware due to natural addition of new functionality.

[1] A C++ version of Loom has recently been released, but the system available when the SCA was created was written in Lisp.

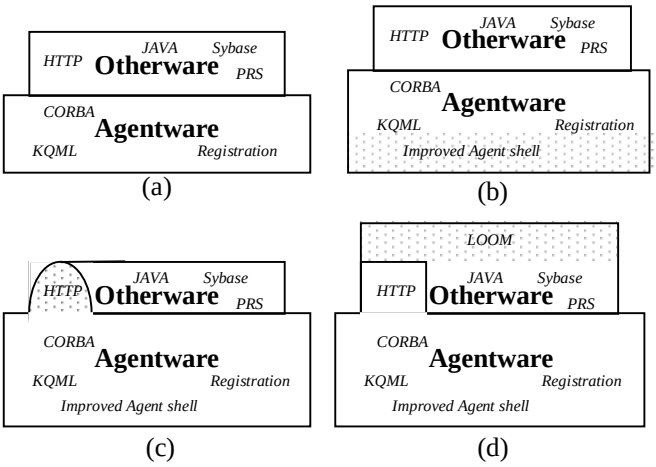

Figure 1. Evolution of Agentware and Otherware

The agentware has been slowly expanded to provide support demanded by the programmers: support both in terms of low-level functionality (e.g., modifying the low-level code to enable sending of typed elements, as opposed to only strings), and higher level demands (e.g., an improved agent shell). Second, is the expansion of the otherware due to the creation of new categories of agents and services. Each time a new agent type is created, the agent programmers have full flexibility to choose their own tools. Such tools include languages (compilers) or commercially available products, such as Sybase (used by the registry agent) or Loom (used by the SCA). This expansion on the otherware level in turn causes the agentware to change. A third type of evolution happens at the gradual shifting of agents from the research testbed[2] into the production system. As agents are moved over to the production system, the change in requirements often requires code changes, which affects the agentware. The resulting agentware simplifies and encourages new agent design.

Figure 1-(a) shows some of the components (APIs and libraries) of the agentware and otherware layers. When the agentware grows because of the addition of new functionality, such as the addition of an improved agent shell, as shown in Figure 1-(b), the ability to create new agents is also improved. When tools or libraries in the otherware layer become popular, they sometimes become part of the agentware as shown in Figure 1-(c). When new agents are created, new tools or libraries are often used, therefore expanding the otherware, as shown in Figure 1-(d). Each of these changes facilitates the others. Improving the agentware makes creation of new agents simpler, encouraging agent designers to create more agents. As the agentware grows due to the incorporation of code or tools, once used only by a single agent designer, newer services are available to expand the functionality of existing agents, as well as create new ones. When new agents are created, there are more opportunities to expand the otherware.

[2] The research testbed shares the same resources as the production system, but by convention agents choose not to communicate across these lines unless necessary. Research testbed refers to the collection of agents used for research-related activities, while the production system refers to the agents involved with service provision to real users (middle and high school students).

An Example: The Task Planner Agent

To illustrate these evolutions, we describe the evolution of the Task Planner Agent (TPA). The TPA is a mediator agent designed to act as a middleman between the User Interface Agent (UIA) and the registry by accepting queries and returning lists of relevant Collection Interface Agents (CIAs) [14]. In the earliest stages of the project, there were four types of agents, CIAs, UIAs, TPAs, and a registry. The TPA simply passed requests from the UIA to the registry with no additional processing. At present there are at least four wholly different "flavors" of TPAs, ranging from those with advanced learning to the production-system TPA which incorporates auction protocols to ensure proper system load balancing. The evolution of the TPA is one example of the expansion of the agentware, as well a demonstration of the importance of design flexibility provided by the two-layered design environment. In addition, the current TPAs show how the expanded agentware simplifies agent creation.

There are five critical stages in the evolution of the TPA. At each stage, there was either a direct or indirect agentware evolution occurring. The first TPA, the simplest one, was built prior to the incorporation of CORBA into our agentware, and like all new agents, was significantly less functional than the current varieties. The second evolution occurred as the research interests called for the incorporation of an advanced planning and reasoning

system (UM-PRS) to function as the core of the agent, allowing advanced AI functionality. The third stage demonstrated the flexibility of our system by incorporating the services of other mediator agents, a BSO (Broad System of Ordering) and a NASA Thesaurus. In addition, the agentware grew through the addition of mailbox communication support. At this stage, a break between the interests of research and user service provision occurred. This break demonstrates the flexibility of our agentware and architecture by allowing two wholly different development efforts to occur simultaneously. The fourth stage, inside the research testbed, demonstrated the ease of agent design (using agentware and UM-PRS), by creating different flavors of TPAs. At the fifth and final stage, the current production system version was created, which demonstrated the ease of integration. For example, the research on the auction protocols resulting in advancements of the agentware was incorporated into the production version of the TPA with ease.

Stage 1: The Initial TPA. The initial TPA was intended to demonstrate the concept of a mediator agent. Its functionality was limited, and it was built through manual coding in C++. At this point, the agentware provided only the most basic socket-level interfaces.

Stage 2: Integration of CORBA and UM-PRS. The experiments with the simpler agents at the first stage pointed towards the use of CORBA as the minimum agent requirements. Using CORBA as opposed to socket-level communication (KQML on TCP/IP sockets) enables more complicated structures to be communicated between agents, as well as providing a solid framework from which to build future agents.

Another major evolution in the TPA was the decision to use UM-PRS as the agent control language. UM-PRS is a proactive procedural reasoning system, which allows advanced reasoning about multiple goals implemented in a belief desire intention (BDI) framework. Incorporation of UM-PRS demonstrates the flexibility of our design environment as the agent designers were able to choose their own control language. The incorporation of UM-PRS allows for a much richer set of functionality. Since UM-PRS allows arbitrary knowledge areas and rules, the TPA was given more autonomy, and its functionality could be easily modified through changing of the UM-PRS knowledge areas. Our agent development environment is such that should some other agent control language be desired, it could be easily added.

Stage 3: Mailboxes, Thesauri, and New Objectives. The third point in the evolution of the TPA, with respect to changes in the agentware, was the modification of the nature of the interaction between UM-PRS and the agentware. For performance and reliability reasons, we integrated the UM-PRS into the same process as the agent. The UM-PRS and ILU control loops were run in different threads of the same process and communicated via a global mailbox. The mailbox communication scheme was rapidly incorporated into the agentware, and is currently used in all UMDL agents which use asynchronous communication.

In addition to the expansion of the agentware, we had the addition of two new agent classes, a BSO (Broad System of Ordering) agent and a NASA Thesaurus agent. With these agent services, the TPA could broaden or narrow queries and find alternate terms, aiding in the search process. The low-level agentware provided the necessary communication infrastructure for all the agents, and the decision to use CORBA and KQML enabled simple intra-agent communication. Since the KQML communication comes “for free” from the agentware, the ability to talk to new agents required no modification, except for the simple addition of the rules to speak the appropriate agent-specific protocols (which is simpler than making custom socket-level communication code). In addition, UM-PRS allowed developers to change the way the TPA processes each request with respect to the newly available services.

Stage 4: A Split between the Production System and the Research Testbed. Up until this point, the UM-PRS TPA was sufficient for both research and production needs. However, as the time to deploy the production system approached, the requirements of the production and research systems diverged. For the production system, the TPA needed to exhibit high performance and dependability, and only a fixed set of functions was required. The needs of research suggested an ideal opportunity to experiment with the flexibility of UM-PRS and to try many different experiments with regard to the unique intelligence of the TPA.

Using the UM-PRS TPA as a prototype, a smaller, more efficient TPA was built for the production system. It sacrificed the flexibility of UM-PRS in favor of run-time performance. The lineage of the UM-PRS TPA was continued for research purposes. Since both TPAs were implemented using the same agentware and supported the same performatives, the switch from one TPA to the other was transparent to the rest of the system.

The development lines of other agents had similar splits at this time. The result was the formation of two separate systems: the production system and the research testbed. Although there are two systems, the division is only a logical one. The agents share common resources and have the ability to interact across the testbed boundaries. The distinction is one of policy, not a physical partition. The fact that the two systems are only logically different facilitates moving of agents from one system to the other. An agent in the research testbed that has functionality desired by the production system can be accessible to the production system agents serving students using the UMDL. Likewise, new agents and improved agentware produced for the production system are immediately available to the research testbed.

Stage 5: Continued Evolution of the TPA. The current production-system TPA, which processes hundreds of queries a day, was built from the TPA of the third evolution stage. The generality of UM-PRS (and runtime overhead required to provide it) was not necessary given routine patterns of UMDL usage by students. Therefore, the TPA was re-implemented as a simpler, faster, but less flexible, multithreaded C++ program.

During the time this new TPA was being built, research was being done on the use of auctions as a means of load balancing and resource distribution. So that other agents could take advantage of this research, the code developed for the auctions was incorporated into the agentware. Once in the agentware, adding it to the production TPA was simple. This is an example where the evolving agentware simplified the addition of services to existing agents.

In the research testbed, three different types of TPAs have been developed. A TPA with a learning algorithm and a TPA with strategic reasoning capability have been developed to test individual graduate-student research, and a third, "dumb" TPA has been also made for comparison. UM-PRS and agentware make it easy to create new instances of the agent, each with wholly different goals and procedures. The ease by which new research ideas can be added to an agent demonstrates the power of the agentware and the flexibility of our development environment.

At present, the agentware has a set of optional libraries available for agent designers, and its flexibility is demonstrated by the ability to simultaneously develop different flavors of an agent, each with completely different purposes and requirements.

Lessons Learned

As the UMDL evolved, so did our development environment. The UMDL evolved in a path strongly affected by the architectural and policy decisions, as well as through the hard work and ideas of the many agent builders. The evolution of the UMDL with respect to the new services and new agents affected the tools such that future agent design was improved. Along this path, there were many challenges.

Throughout our project, we have learned that it is possible to have a large multiagent system that works. Our current system is actively used in the local middle and high schools. To date, over 1000 school children have used it as part of their regular science classes. It also provides a research testbed where new agent models and designs may be tried out.

Our decision to allow third party tools has provided extensive flexibility but has come at a price. Tools not made specifically for the agentware may have difficulties in integration and have no guarantee of support. Some examples include UM-PRS and Loom. UM-PRS required the addition of many new tools and libraries to the agentware, such as the mailbox code. Loom required the programming of Unix sockets as a means of communication between the LISP engine and the C++ agentware core. A benefit of this, however, is that the agentware has encompassed the mechanisms that were developed for such integration.

A second lesson is that an open system, sometimes, results in redundant work. Because we allow designers flexibility in tool choices, it may be the case that the same task is done by using two different tools. One example is the inclusion of HTTP support for our agents. At one point, there were three different agents which each utilized independent HTTP libraries. Eventually, one HTTP library was selected and added to the agentware. It has since been used to develop other agents, eliminating the need for the designers to maintain their own versions of the library.

A third lesson is that implementing a more general architecture requires more work. For example, since the architecture made no commitment with respect to the communication protocols, both synchronous and asynchronous communication were implemented. If the system were mandated that a single type was used, then the total work may have been less. However, current agent designers take advantage of the ability to use either model, and the generality is worth the extra effort.

A fourth lesson is that, in an open system, the core set of functionality evolves slowly over time instead of being specified from the start. Many of the current agents were designed by graduate students for their research. In the early part of the project, the agentware was much smaller and poorly documented, making creation of new agents difficult. If a monolithic toolkit were available, more agents may have appeared sooner, but those agents would have been limited by the capabilities of the toolkit. Currently, the agentware contains libraries for almost every task a new agent designer would desire. In the event a particular functionality cannot be realized by the agentware, the designer is always free to choose third party tools while providing the necessary means of integration.

Despite the problems encountered while learning these lessons, the high level of flexibility provides many opportunities that might not have otherwise been present. By separating the agentware and otherware layers, agent designers can easily create agents that interact within a large society of agents and can make local decisions on tools related to a specific functionality of each agent. The current mix of agents is diverse in ways not imaginable during the early days of the project. We currently have agents capable of re-ordering search results, crawling the web, planning queries, recommending results, suggesting alternate terms, or topics, using advanced learning to choose the best resources, automating service classification, auctioning services and goods, and providing a plethora of diverse content. This list is growing every day, as is the underlying set of tools to support it.

Acknowledgments

We would particularly like to thank José Vidal and Peter Weinstein for sharing their development experiences with UM-PRS and Loom. We would also like to thank the rest

of the UMDL Service Market Society group (Bill Birmingham, Tracy Mullen, and Bill Walsh) for their comments and ideas. This project has been funded in part by the joint NSF/ARPA/NASA Digital Libraries Initiative under CERA IRI-9411287.

References

[1] Atkins, D.E., Birmingham, W.P., Durfee, E.H., Glover, E.J., Mullen, T., Rundensteiner, E.A., Soloway, E., Vidal, J.M., and Wellman, M.P., 1996. Toward Inquiry-Based Education Through Interacting Software Agents. *IEEE Computer* 29(5):69-76.

[2] Brill, D., 1993. Loom Reference Manual, Version 2.0, University of Southern California.

[3] The Common Object Request Broker: Architecture and Specification, Revision 1.1. Framingham, Massachusettes, USA: Object Management Group.

[4] Durfee, E.H., Kiskis, D.L., and Birmingham, W.P., 1997. The Agent Architecture of the University of Michigan Digital Library. *IEE British Computer Society Proceedings on Software Engineering* 144(1).

[5] Finin, T., Fritzson, R., McKay, D., and McEntire, R., 1994. KQML - An Information and Knowledge Exchange Protocol. In Knowledge Building and Knowledge Sharing,

[6] Full Text Lexicographer Homepage. `http://kalex.engin.umich.edu/ftl/`.

[7] Huber, M.J., Lee, J., Kenny, P.G., and Durfee, E.H., 1994. UM-PRS V3.0 Programmer and Reference Guide, University of Michigan, Artificial Intelligence Laboratory.

[8] Information Retrieval (Z39.50): Application Service Definition and Protocol Specification, ANSI/NISO, Z39.50 - 1995, Library of Congress. `http://lcweb.loc.gov/z3950/agency`.

[9] Inter-Language Unification Homepage. `ftp://ftp.parc.xerox.com/pub/ilu/ilu.html`.

[10] Kiskis, D., 1995. UMDL Architecture Requirements Analysis, Internal Report, School of Information and Library Studies, University of Michigan.

[11] Lee, J., Huber, M.J., Durfee, E.H., and Kenny, P.G., 1994. UM-PRS: An Implementation of the Procedural Reasoning System for Multirobot Applications. *Proceedings of the AIAA/NASA Conference on Intelligent Robotics in Field, Factory, and Space*.

[12] Tools.h++ Homepage. `http://www.roguewave.com/products/tools/tools.html`.

[13] University of Michigan Digital Library Homepage. `http://www.si.umich.edu/UMDL/`.

[14] Vidal, J.M., and Durfee, E.H., 1995. Task Planning Agents in the UMDL. *Proceedings of the Fourth International Conference on Information and Knowledge Management (CIKM) Workshop on Intelligent Information Agents.*

[15] Weinstein, P. and Birmingham, W.P., 1997. Runtime Classification of Agent Services. *Proceedings of theAAAI-97 Spring Symposium on Ontological Engineering.*