On Using KQML for Matchmaking

Daniel Kuokka Larry Harada *
Lockheed Research Labs, O/96-20, B/255
3251 Hanover Street, Palo Alto, CA 94304
kuokka@aic.lockheed.com, harada@aic.lockheed.com

Abstract

As agents see more use as entry points to increasingly complex distributed information networks, agent communication technologies such as the Knowledge Query and Manipulation Language and the SHADE Matchmaker will play an important role. We describe our experiences with these technologies as applied to two applications: collaborative engineering and satellite image retrieval. Based on these experiences, we outline the major observed benefits of KQML and matchmaking. In addition, we discuss several problematic issues and potential solutions, including representational challenges in advertising complex databases, the need for persistent requests in information brokering, the dilemma between explicit vs. implicit brokering, problems in error recovery and response timing, consistency among information providers, and efficiency.

Introduction

Interest in information agents has undergone explosive growth as wide-area information networks outpace traditional information sharing and retrieval techniques. A precise definition for "agent" is still elusive, but some commonly cited elements of agency include autonomous or semi-autonomous operation, production and consumption of application-specific information, and communication and interaction with other agents to help fulfill goals.

We view the focus on communication and interaction as being particularly important. We anticipate future information networks to be extremely large, highly dynamic collections of information agents—so dynamic, in fact, that traditional approaches centered around querying and updating static, central databases will not work. The lifetime of information will be too short-lived and the sources will be too unpredictable. This situation is already the norm in large engineering efforts, where many different tools consume and produce data (Cutkosky *et al.* 1993).

In order to cope with the dynamic nature of information networks, agents will depend much more heavily on expressive communication not only to answer queries, but also to describe their information capabilities and needs. In order to enable the expressive communication required, a number of researchers have been working on knowledge sharing languages (Patil *et al.* 1992). One of the major efforts is the definition of a Knowledge Query and Manipulation Language, or KQML (Finin *et al.* 1993), a language in which agents can express their beliefs, needs, and preferred modalities of communication.

The SHADE project (Kuokka & Harada 1995a; McGuire *et al.* 1993) has been exploring the use of KQML to define an agent infrastructure for collaborative engineering. SHADE has several major foci, including techniques for defining and using ontologies, and creating an application programmer interface (API) for KQML. SHADE is also defining facilitation agents—*middleware* agents that assist *end-user* agents in information sharing. One of the major facilitation agents defined by SHADE is called a *matchmaker* (Kuokka & Harada 1995b).

The SHADE infrastructure has served as the foundation for agent-based services being developed by several projects. Therefore, we have compiled a significant body of experience in using agents and KQML. In this paper, we outline some of our initial experiences gained while using KQML and matchmaking for interactive agent applications. Where shortcomings have been identified, we propose solutions and discuss alternatives.

Matchmaking and KQML

KQML defines a set of message types (called performatives) that describe the sender's "attitudes" about knowledge. KQML messages types include simple queries and assertions (e.g., `ask`, `stream`, and

*This work was supported by ARPA Contract DAAA 15-91-C0104 (Shared Knowledge-Based Technology for the Re-Engineering Problem), monitored by the U.S. Army Research Laboratory, Advanced Computational and Information Systems Directorate. The views, opinions, and/or findings contained in this report are those of the authors and should not be construed as an official Department of the Army position, policy, or decision, unless so designated by other documentation.

`tell`), routing and flow instructions (e.g., `forward` and `broadcast`), persistent queries (e.g., `subscribe` and `monitor`), and information brokering requests (e.g., `advertise`, `recommend`, `recruit`, and `broker`), which allow information consumers to ask a facilitator to find relevant information producers. The knowledge carried by a KQML message is referred to as the content, and may be in any language. For this discussion, we assume a content language of KIF (Genesereth & Fikes 1992). Also, even though KQML defines many parameters such as `:language`, `:ontology`, `:reply-with`, we include only those parameters vital to the examples.

Much of the power of KQML stems from its brokering performatives, which must be implemented by a facilitator. Thus, our discussion of agent cooperation must necessarily include facilitators. The SHADE matchmaker (Kuokka & Harada 1995b) is one such facilitator, and others such as the ABSI facilitator (Singh 1993) have also been developed. The matchmaker serves as a central clearinghouse to which other agents can advertise their specific information capabilities, request pointers to providers of information, or even ask to keep them informed of changes to classes of information. The matchmaker is accessed via standard KQML messages. Advertisements are sent using the KQML `advertise` performative. Requests are sent using the `recommend`, `recruit`, and `broker` performatives. The matchmaker also supports a variety of other KQML performatives, such as `tell` and `subscribe`.

Experience with KQML and Matchmaking

The SHADE infrastructure, including KQML and the matchmaker, has been used as the basis for several applications including a collaborative engineering testbed and a satellite imagery clearinghouse. The engineering testbed is a collaborative environment in which engineers can dynamically locate and share heterogeneous data. It consists of agent-wrapped versions of several commercial engineering tools, such as SDRC's I-DEAS solid modeler, and research prototypes, such as Lockheed's Parameter Manager (Kuokka & Livezey 1994).

In this testbed, KQML and matchmaking have been shown to be very effective integration mechanisms. For example, any number of engineers could have a Parameter Manager (ParMan) application running, in which they state constraints on specific product parameters. Without the SHADE infrastructure, if the systems engineer decides to add a system-wide constraint, he has no way of knowing exactly which other ParMan agents are running, let alone which should be notified of the new constraint. KQML and matchmaking allow each ParMan agent to post advertisements and subscriptions for specific parameters of concern. The system engineer, in turn, need only ask the matchmaker for those agents having expressed interest in the newly constrained parameter. Thus, agents can locate the sources and sinks of information, even though they change dynamically.

A second application of matchmaking is based on a satellite imagery retrieval system, which allows users to locate and retrieve variable-resolution satellite imagery from multiple dynamic sources. A prototype has been built using KQML and the matchmaker (as well as other elements of the SHADE infrastructure). In this application, the matchmaker has proven invaluable because there are multiple sources of data, which are constantly being updated as satellites circle the earth. Only an automated system can offer the up-to-the-minute location of data required. Furthermore, the image databases have complex schemata and overlapping data availability. KQML, with its flexibility of content languages, allows these databases to be characterized for the matchmaking process, thereby directing the user to appropriate data sources without excessive exploration.

KQML and matchmaking have been used by several other projects as well, with similar positive results. The Cosmos project (Mark & Dukes-Schlossberg 1994), which is creating a knowledge-based commitment reasoner to determine impacts of engineering changes, uses KQML for all message traffic, and depends on the matchmaker to provide indirection between a set of dynamic clients and the server. The ARPA Simulation Based Design project (Davis *et al.* 1993) uses the matchmaker to provide change subscription and notification services over its large, object-oriented product model. In this application, if an object for which a subscription has been issued changes, the user will receive automatic notification. Other applications of the matchmaker, such as its use to locate relevant pages in a large distributed engineering notebook, are in earlier stages of development.

Finally, several informal contrapositive examples have been identified from discussions held with architects of non-agent-based systems, which depend on ad hoc communication among the components. In these cases, the architectures have been found to be a barrier to maintenance and enhancement, since the complex information flow among components is hardwired and implicit. In fact, several opportunities to add needed functionality available for free from other projects had to be scuttled since the affected subcomponents were too hardwired to accept input from new sources. These cases provide compelling examples of the cost of not using an open, dynamic architecture, such as that enabled by KQML and matchmaking.

Issues in using KQML and Matchmaking

Even though KQML and matchmaking have shown great promise in several applications, a number of open issues and problems have reoccurred. These issues have been grouped into several classes, which are described below. When promising solution have been identified, these are also presented.

Persistent Requests

We use the term *request* to refer to performatives that ask the matchmaker to locate another agent that can provide needed information (e.g., `recommend`, `recruit`, and `broker`). Our experience with these messages has revealed a significant shortcoming in the current specification: the need for persistent requests.

For example, when a consumer agent sends a `recommend` to a matchmaker, the matchmaker responds with the name of one or more producer agents that have advertised a relevant capability. This works when all producers have already sent their advertisements, such as when producers are traditional static databases that run continuously. However, this assumption is often not valid, especially when the information shared by agents is dynamic. In this case, information providers may appear dynamically, or an existing agent may augment its information library dynamically. Returning only the currently known producers in response to a request clearly limits the potential for information sharing.

The problem is compounded when the assumption of a consumer/provider dichotomy is relaxed. As in the case of a set of interoperating ParMan agents, information consumers can also be providers, in which case, the network is more a collection of peers rather than of clients and servers. In such a network, a scheme without persistent requests would require that all agents first advertise their capabilities, and only then issue requests. However, this is impossible, since peer agents can startup and shutdown independently. Furthermore, even within a running agent, specific information capabilities may be added dynamically (for example, when the user adds a constraint over a new parameter in ParMan).

To provide a persistent request capability, several approaches were considered: make requests persistent by default, add a persistence parameter, or wrap requests within a `subscribe`. The first option was ruled out since non-persistent requests are sometimes needed. The second option is to add an additional parameter that specifies the persistence of the request. The `:until` parameter can assume three values:

now The request performative will be answered with respect to the current set of advertisements. If there's no match, a `sorry` will be returned to the sender. This is the current KQML behavior, and is the default.

once The request is kept until it can be matched; then it's answered and removed. The request performative will never return a `sorry`.

forever The request performative is kept forever, resulting in the request being sent to each agent advertising a relevant capability.

This approach provides great flexibility, but requires an addition to the KQML specification that violates the desired orthogonality of parameters.

The third approach is to wrap the request in a `subscribe`, which requires no extension to KQML and appears to be an elegant application of KQML's orthogonality. Unfortunately, upon closer consideration, it is unclear whether the semantics of KQML support this. `Subscribe` "indicates that the sender wishes the recipient to tell it about future changes to what would be the response(s) to the KQML performative in the `:content` parameter." This would work for `recommend`, since the matchmaker simply returns the name of an agent capable of satisfying the content request. However, `recruit` and `broker` return only an acknowledgment—their main purpose is to perform a side effect, namely to forward the content request to a provider. An agent could use a two-part protocol, first issuing a subscribe-of-recommend, and once a suitable agent is found, issue either a `recruit` or `broker` to actually get the data, but this was seen as too cumbersome.

Since an immediate solution was desperately needed, the second approach has been implemented within the SHADE matchmaker, resulting in a marked change in system utility and robustness. With this addition, agents must no longer be carefully started in the correct order, and restarted when any agent goes down. Instead, the network of advertisers and requesters organizes itself. Based on this experience, we conclude that persistent requests are critical to the utility of the broker performatives. If the semantic uncertainties of the current specification can be resolved, KQML may be found to support this as is. But future matchmakers must provide support for this feature.

Generality of Advertisements and Requests

In traditional information retrieval and knowledge-based query applications, the knowledge base tends to be a large set of specific assertions, and queries tend to be relatively specific. In order to advertise such a database in KQML, the naive approach would be to send a message such as:

```
(advertise :sender P :content
  (ask :content (and (mass payload 12)
                     (mass bus 7) ...)))
```

A consumer might, in turn, send a request like:

```
(recommend :sender C :content
  (ask :content (mass payload ?mass)))
```

In this case, determining whether or not the request matches the advertisement can proceed as a traditional logical database query.

However, it is impractical for the provider to advertise its whole database. Instead, information producers might send advertisements such as:

```
(advertise :sender P :content
  (ask :content (mass ?component ?mass)))
```

Unfortunately, this advertisement is certainly over general, since the provider does not know the mass of all components. A solution would be to include further constraints on the advertisement:

```
(advertise :sender P :content
  (ask :content
    (and (mass ?component ?mass)
         (subcomp ?component satellite))))
```

However, this approach requires that the matchmaker has sufficient knowledge to determine if the subcomponent condition is true, which either forces the matchmaker to contain domain-specific knowledge, or to request domain specific knowledge at a significant performance cost. Thus, the cost of performing more careful matchmaking may outweight the benefit. There is also a representation problem in that the single `:content` field does not make it clear that the matchmaker should view the subcomponent predicate as a constraint over valid values for `?component`, as opposed to an advertised pattern. The second problem has resulted in a proposal for a `:content-constraint` parameter, which separates the content pattern from the constraints, but the ramifications of this proposal are still under investigation.

These problems can be partially addressed by requiring the constraints to be universal predicates, such as basic mathematical relations. This would allow advertisements of the form:

```
(advertise :sender P :content
  (ask :content
    (and (mass ?component ?mass)
         (date ?component ?date)
         (>= ?date 941201))))
```

In this case, whereas the producer cannot limit the advertisement to those subcomponents of the satellite, it can limit its advertised information to components manufactured after a certain date, since the matchmaker can verify this simple mathematical constraint. The advertisement is still over general, so spurious matches are still possible, but constraining advertisements in this way has proven very useful in characterizing certain databases, especially in the satellite imagery domain.

Finally, even if the advertisement is not over general, seemingly spurious matches may still be found in the case of `recommend`. Consider the following advertisement and request:

```
(advertise :sender P :content
  (ask :content (mass payload 12)))

(recommend :sender C :content
  (ask :content (mass ?comp ?mass)))
```

In this case, the advertisement is extremely specific, but since the consumer made a very general request, the match succeeds. Unfortunately, the consumer might conclude that the producer can answer queries about any component, and so might ask the producer about the mass of other components (notice that this confusion is less likely in the case of `recruit` and `broker`, since an answer is returned to the consumer based on the original request). This underscores that matchmaking locates *possible* sources of information, not *guaranteed* sources.

Only partial solutions to the problems of overgeneral advertisements and requests have been identified. However, based on our experiences, the advantages of succinctly specifying a large database generally outweight the costs of extra matches. When spurious matches are returned, the consumer will, at worst, receive a sorry from the producer.

Error Recovery

In a dynamic environment, agents may intentionally or unintentionally stop receiving and sending messages for arbitrary lengths of time. Ideally, the matchmaker should support clients that want to suspend their operations or recover from crashes. Likewise, the matchmaker should allow agents to reconnect if the matchmaker itself crashes.

There are three classes of transients that the matchmaker should handle. First, if an agent suspends itself or otherwise gracefully shuts down, the matchmaker should be notified of this event so it can take appropriate action. For example, in the ParMan application, the `deny` performative is sent to cancel advertisements whenever a ParMan agent shuts down. This is adequate when the agent is shutting down permanently, but if the agent is only suspending temporarily, some messages that would have been delivered may get lost.

In the Simulation-Based Design application, agents can send a `suspend` to the matchmaker, which then queues subscription notifications for the suspended clients. These notifications are later forwarded to the client when an `update` performative is received by the matchmaker from the re-awakened client. This mechanism is somewhat unsatisfactory since it relies on two ad hoc extensions to KQML: `suspend` and `update`. However, it does illustrate the usefulness of a simple high-level protocol to prevent messages from being lost.

The second class of transients are those where an agent unexpectedly dies or loses communication with the matchmaker. In our experiences with SHADE, this class of transients is very common, and introduces many problems. To cope with this situation, two alternatives present themselves. A high-level solution would require each agent to acknowledge received messages with a `reply` performative. Failure to acknowledge would be taken as an indication that the message should be resent. A low level solution would require the transport level protocols to indicate that a message had failed to be delivered. For example, the SHADE KQML API returns an error code when a message can not be sent. The former approach places additional burdens on the agents and on the communi-

cation bandwidth, while the latter makes assumptions about the communication medium.

In either case, the action to be taken by the matchmaker is unclear. Should the matchmaker queue messages for the affected agent, preventing any messages from being lost (this assumes that the recovering agent sends a message to the matchmaker upon restart). Or should the matchmaker assume that the agent has shutdown for good. As a corollary, what if an agent comes up with the same name as an agent that was previously employing the matchmaker's services? Did the original agent crash and restart, is this a new instantiation of the same agent not interested in the old services, or is this a name conflict? Our experience with these issues has revealed significant holes in the KQML specification relating to transactions and reliable services.

Finally, the third class of transients is when the matchmaker, itself, gracefully shuts down or crashes. Since the matchmaker acknowledges all KQML messages that have `:reply-with` tags, client agents that do not receive acknowledgements can safely assume that the message was not received. Thus, unexpected matchmaker crashes are noticed by each client agent. In the case of a graceful shutdown, the matchmaker should send a suspend message to its current clients so they don't have to determine the matchmaker's unavailability for themselves. But KQML does not provide a convenient mechanism for doing this; the matchmaker must explicitly cancel each of its services. Finally, the matchmaker must certainly maintain a persistent knowledge base of clients, which can be recovered upon restart. Otherwise, all the advertisement and request context would be lost.

A related issue is that of timeouts. When a consumer sends a request to a producer, the time to compute the answer can be arbitrarily long, potentially causing the consumer to assume the producer (or matchmaker) went down. In general, consumers may want to know how long it will take to satisfy a request, or at least determine its status. There are several potential solutions. First, an advertisement could include the computation time required for the advertising agent to satisfy the request. Of course, such a value will be an estimate, since the actual value may depend on the parameters of the request, machine load, and other factors. Second, producers might support queries about the timing and status of their processing. This would require developing an ontology of terms to talk about queries.

The third alternative involves establishing a request initiation protocol, managed by a matchmaker. Instead of simply forwarding the request to the producer, the process would proceed as follows: 1) The matchmaker would "ping" the producer with the request, asking "Are you prepared to answer this request?" and "How long will it take?" 2) If the producer is prepared to answer the request, the time to complete the request is returned to the consumer. 3) If the consumer approves the request time, the request is finally forwarded to the producer. 4) The answer is returned to the consumer (or matchmaker). A protocol such as this involves developing an ontology to talk about requests. Since both agents have approved the query prior to its start, there should be no need to cancel the request. However, the actual computation may take significantly longer than estimated; thus, status request and cancellation mechanisms would be useful.

The above proposals not withstanding, based on our experience, KQML does not include sufficient provisions for error recovery and suspending operations. Furthermore, it is not clear how an atomic transaction model can or should be implemented. Before KQML can be used extensively in real applications, these issue must be addressed.

Specification of Content-Based Routing

The request performatives allow the consumer take a proactive role in information exchange, in that the consumer ultimately takes responsibility for querying for the data. Another useful paradigm, however, is for information providers to take the lead and assert information as it changes. KQML supports this truth-maintenance capability via the `subscribe` and `monitor` performatives, which allow a consumer to be kept informed automatically about information. As conceived, the consumer sends a `subscribe` directly to a producer, since the producer knows when its information changes. Unfortunately, this requires that the consumers know all possible producers, and that the producers send individual messages to possibly many interested consumers.

A useful variant of this approach is for the consumer to send a `subscribe` to a central facilitator. Producers, in turn, send updates to the facilitator. This allows each agent to be concerned with only one clearinghouse. This approach is called *content-based routing*, and has proven extremely useful in many applications. A content-based router does not have a knowledge base like other agents since it does not store the union of all it hears—it just forwards `tell`s on to other interested agents. In most cases, this does not matter, since to a consumer, the matchmaker will appear as if it does contain in its knowledge base all the information. In fact, the KQML specification explicitly allows such behavior, since it uses the term "*virtual* knowledge base."

However, if the consumer sends an `ask` message, the matchmaker will not be able to respond since it only passes along messages. Thus, the matchmaker has the peculiar property of being able to satisfy subscribe-of-ask-x but not ask-x! (In fact, it has been the subject of debate as to whether or not a subscribe-of-ask should implicitly include the first `ask`.) To address this problem, the content-based router could send an `ask` to the real information producers in response to an `ask` from a consumer. With this enhancement, the content-

based router is presenting a facade that it can answer queries, but it really gets the information from other agents and simply takes credit (much like managers and politicians).

One might view such an agent as a useful abstraction, but note that identical behavior can be requested explicitly by wrapping the `asks` and `subscribes` inside of a `broker`. Since this transparent mechanism exists, should a content-based router that presents an opaque facade be used? Beyond a moral concern about misleading advertising, if agents take credit for others' information, loops could easily develop in which agent A depends on agent B which depends on agent C which depends on A. Rather than attempting to restrict the behavior of agents, the network could rely on additional facilitators that detect when another agent is really providing second hand information at a markup, or detect when information services are being abused or participating in a loop.

A second more practical concern is that content-based routing, as implemented to date, depends on producer agents "babbling" about about updates. This raises the questions: when do producers babble, and what causes them to babble? As agent communities grow in size and dynamiticity, unnecessary messages could easily bog down the network. Conversely, new agents with information of interest to a content-based router may not know about the router, and therefore wouldn't know to babble. Both of these problems can be fixed by requiring the content-based router, upon receipt of a subscription, to issue a `subscribe` to the original providers, which must have advertised their relevant information. Thus, only those agents that have been requested to provide information babble.

To summarize, content-based routing has been found to be an extremely useful agent communication paradigm, but the messages required are somewhat more complex than initially believed. In addition, there is an interesting philosophical dilemma as to whether a content-based router should make explicit its behavior or present an opaque facade for other agents.

Other issues

Several other issues and techniques have been identified with respect to matchmaking. First, consistency may become a problem as agents rely on second-hand information sources. If multiple agents can answer a query, it is possible that their answers will be inconsistent. If producers assert overtly contradictory facts, problems are, of course, inevitable. However, in complex domains, even if producers are correct, it is likely that they will be subtly inconsistent. For example, different simulations may produce slightly different answers. If the same simulation agent is used throughout a computation, such errors may not be significant. However, since matchmaking does not guarantee that the same agent will answer subsequent requests, normal mutual inconsistencies may cause problems for a consumer (e.g., "why did this value change so much when I only changed this one parameter?"). Such problems can be avoided by using the `recommend` performative, which gives the consumer control over which producer is used, but this may be overly restrictive.

Another issue that will become critical as agent networks increase in size is efficiency. As many agents depend on a central matchmaker to route messages and broker information, the matchmaker could easily become overwhelmed, creating a bottleneck. There are several potential solutions to this problem. The first solution is the use of performatives, such as `recommend`, that force the bulk of the communication burden on individual agents. This approach uses matchmaking to make the initial connection only, not for subsequent higher-bandwidth communication. Another solution is to construct communities of matchmakers in which different individual matchmakers are used to serve different clients. Thus, as demands on the matchmaker grow, the capacity of the distributed matchmaker also grows. Matchmaking is ideal for distributed processing since its workload is naturally partitioned according to the client agents. Finally, the granularity of messages can make a large difference in efficiency. Rather than issuing many very specific advertisements or requests, agents should take advantage of the expressivity of the content language to issue fewer, more general advertisements and requests.

An inconvenience that has become apparent in our experiments is that there are common co-occurring sets of messages, but KQML does not provide for their convenient grouping. For example, when an agent advertises its willingness to fulfill `subscribes`, it typically also advertises its willingness to deny (or cancel) that `subscribe`. This can only be done currently via a second verbose message whose content is largely a duplication of the first. A mechanism that permits multiple performatives to refer to the same content would reduce the volume of message traffic as well as reduce redundancy and errors.

Whereas the above is focused on the efficiency of matchmaking, the efficacy of matchmaking, i.e., identifying relevant matches and avoiding spurious matches, is also an issue. As discussed previously, matchmaking efficacy depends on the producer and consumer agents carefully crafting correct content fields. However, an intriguing feature being investigated is to allow matchmaker clients to provide feedback about the matches found, e.g., a consumer might tell the matchmaker that this was a bad recommendation. Based on this feedback, the matchmaker could learn better descriptions of the various producers' capabilities and consumers' needs. Even if client agents don't provide explicit feedback, the matchmaker could infer when incorrect matches are made based on required messages such as the KQML `sorry` performative, which is sent when an agent cannot satisfy another's request.

Conclusions

In their use as core infrastructure elements supporting several diverse applications, KQML and the SHADE matchmaker have proven extremely valuable. KQML provides a much needed language for encapsulating domain-specific information. Without KQML, distributed applications, at best, create their own language of messages, and at worst, depend on completely ad hoc or hardwired exchanges. The specific performatives defined by KQML support many important and useful mechanisms allowing distributed agents to discover and share information dynamically. We have found the brokering performatives (e.g., `subscribe`, `advertise`, `recommend`, `recruit`, and `broker`), as implemented by the matchmaker, to be particularly important in the dynamic network of agents support by the SHADE infrastructure.

However, based on our experiments with matchmaking and KQML, there are numerous areas in which additional work is needed. Persistent brokering requests must be supported, there is inadequate support for transactions, error recovery and status checking, the use of `subscribe` to implement content-based routing carries several dangers, and consistency of information is placed at risk when the actual sources are abstracted from the consumers. Also, the efficiency and efficacy of KQML message traffic has not been analyzed sufficiently for large systems of agents. Finally, representational adequacy for advertising large, complex information bases is a continuing challenge. (Since KQML is indifferent to content language, this is not really an indictment, but it is an important concern in the use of KQML).

Before KQML can grow into a widely used language for agent interaction, and before matchmaking can become a useful service in emerging information networks, these and other issues must be addressed. In addition, KQML must be actively merged with other common messaging standards such as CORBA. Otherwise, the global information infrastructure may pass by, noting an interesting but irrelevant attraction on the side of the road. Based on our initial trials, however, KQML and matchmaking promise to fulfill a very important need as the information landscape grows and becomes populated with agents.

Acknowledgments

We gratefully acknowledge the insights and assistance of Jim McGuire, Brian Livezey, and our colleagues in PACT and the ARPA Knowledge Sharing Initiative. This work was supported by ARPA prime contract DAAA15-91-C0104, monitored by the U.S. Army Research Laboratory. We thank Morton Hirschberg for his invaluable support.

References

Cutkosky, M.; Engelmore, R.; Fikes, R.; Gruber, T.; Genesereth, M.; Mark, W.; Tenenbaum, J.; and Weber, J. 1993. Pact: An experiment in integrating concurrent engineering systems. *IEEE Computer* 26(1).

Davis, M.; Evans, R.; Davis, G.; and Jones, G. 1993. Simulation based design for submarines. In *Proceedings of the Submarine Technology Symposium, JHU/APL.*

Finin, T.; Weber, J.; Wiederhold, G.; Genesereth, M.; Fritzson, R.; McKay, D.; McGuire, J.; Pelavin, R.; Shapiro, S.; and Beck, C. 1993. Draft specification of the KQML agent-communication language. Technical report, The ARPA Knowledge Sharing Initiative External Interfaces Working Group.

Genesereth, M., and Fikes, R. 1992. Knowledge Interchange Format, version 3.0 reference manual. Technical Report Logic-92-1, Computer Science Department, Stanford University.

Kuokka, D., and Harada, L. 1995a. A communication infrastructure for concurrent engineering. *Journal of Artificial Intelligence in Engineering, Design, Analysis, and Manufacturing.*

Kuokka, D., and Harada, L. 1995b. Matchmaking for information agents. In *International Joint Conference on Artificial Intelligence.*

Kuokka, D., and Livezey, B. 1994. A collaborative parametric design agent. In *Proceedings of the National Conference on Artificial Intelligence*, 387–393. Menlo Park, CA: AAAI Press.

Mark, W., and Dukes-Schlossberg, J. 1994. Cosmos: A system for supporting engineering negotiation. *Concurrent Engineering: Research and Applications* 2(3).

McGuire, J.; Kuokka, D.; Weber, J.; Tenenbaum, J.; Gruber, T.; and Olsen, G. 1993. SHADE: Technology for knowledge-based collaborative engineering. *Concurrent Engineering: Research and Applications* 1(3).

Patil, R.; Fikes, R.; Patel-Schneider, P.; McKay, D.; Finin, T.; Gruber, T.; and Neches, R. 1992. The DARPA Knowledge Sharing Effort: Progress report. In *Proceedings of the Third International Conference on Principles of Knowledge Representation and Reasoning.* Morgan Kaufmann.

Singh, N. 1993. A CommonLisp API and facilitator for ABSI (revision 2.0.3). Technical Report Logic-93-4, Stanford University Computer Science Department Logic Group.