# Domain Specific Languages for ad hoc Distributed Applications

Matthew Fuchs
Walt Disney Imagineering

# Domain Specific Languages for *ad hoc* Distributed Applications

Matthew Fuchs
*Walt Disney Imagineering*
*1401 Flower St., POB 25020*
*Glendale, CA 91221-5020*
*matt@wdi.disney.com*

## Abstract

*The Internet provides a medium to combine human and computational entities together for* ad hoc *cooperative transactions. To make this possible, there must be a framework allowing all parties (human or other) to communicate with each other. The current framework makes a fundamental distinction between human agents (who use HTML) and computational agents, which use CORBA or COM. We propose DSLs as a means to allow all kinds of agents to "speak the same language." In particular we adopt some ideas (and syntax) from SGML/XML, especially the strict separation of syntax and semantics, so each agent in a collaboration is capable of applying a behavioral semantics appropriate to its role (buyer, seller, editor). We develop the example of a card game, where the syntax of the language itself implies some of the semantics of the game.*

## 1 Introduction

The Internet is a large collection of entities, some computational, some human, each evolving independently, with its own goals, strategies, and capabilities. Our goal is to support cooperation among them, both *ad hoc* and institutionalized. This implies coordinating both human and computational agents. Popular current technologies, such as the WWW and CORBA, do not adequately support cooperation because they cannot deal with the heterogeneity among these agents and their goals. Domain specific languages (DSLs), however, can provide a framework for overcoming these difficulties

This kind of heterogeneity is a particularly difficult problem the Internet, and the Web, have not yet dealt with. The current approach, smoothing over differences through general protocols, such as the use of HTML for user interface and Java for applets, only works because of the server-centric nature of the Web. Essentially all computation occurs at the server (such as through CGI scripts) or is directed by the server (such as Java applets and even the display of HTML pages). There is no real communication between the information from the server and the client's environment beyond the browser.

When all code and all applications are developed under a central control, the heterogeneity problem doesn't exist. In our early work with mobile objects[5] we developed both display markup languages and object interfaces; as long as all the code was developed by a single source, all went well. However, our goal was to support *ad hoc* cooperation, and these approaches did not scale well across the range of agents.

If we wish to build distributed applications in a future Web, we cannot assume the reader of a Web page is a human staring at a browser. It may be an application developed entirely by the client. Yet there is no formal way to extract information from a Web page and store it in a database, or to connect forms and pages to workflows. Conversely, applets *are* objects with interfaces accessible to other objects on the client's machine, but this is only useful if their interface supports the client object's needs, as opposed to just other objects from the same server. As is the case in the database world, the server cannot know all the ways a client will want to use its information. Then there is the further problem of combining information across a variety of servers in a seamless fashion.

Domain-specific languages may hold the key to dealing with heterogeneity. As we shall see, they subsume both text markup languages and class interfaces in a powerful way. They are easily transmitted. If they are truly domain specific, they imply little about their implementation, leaving the client free to support a variety of implementations, each specific to a particular purpose.

The rest of this paper will expand on our approach of using DSLs as a means of supporting heterogenous agents in an open network. Section two will expand on the problem. Section three explains the significance of DSLs, while section four describes two short examples of DSLs in use. Section five describes the approach we have been taking. Other work is described in section six and we conclude in section seven.

## 2 How heterogeneity impedes distributed systems on the Web

The current World Wide Web architecture assumes the existence of three entities - the server, the browser, and the (human) user. The server sends information to the browser to be displayed to the use. Occasionally the user enters some information into the browser to be sent back to the server, which responds with some new information. If the user wants to enter the information in some local application it must be done manually. If the user wants to see two screens, or enter information from one screen into a form from another server (essentially creating an ad hoc distributed application), it must be done manually.

We want to replace "the browser and the user" with "a local agent," which may just be a browser and a user, but not necessarily. This local agent receives information – text or objects – from one or more servers and does something with it. What that something is may be to display it, store it, place it within new documents, email it, trigger some transaction, etc. But it is a local decision what to do (which might merely be executing code from the server).

We also want to enable clients to combine information from a variety of servers, or arrange for the servers to communicate with each other. This turns the current Web architecture on its head to treat the whole network in a peer-to-peer way.

Our approach to this question grew out of our experiences developing Dreme, a dialect of Scheme with mobile objects, and what appeared to be the basic asymmetry between communication with humans and communication with programs. In this scenario, suppose an object migrates to your desktop and wishes to communicate with you. If you happen to be a:

- Human, then the conventional mechanism is through a GUI. The GUI provided a sequence of pictures and simple responses for the human to interact with the object/agent. Software engineering "conventional wisdom" insists this will constitute the bulk of the object's code. Our earliest efforts looked at traditional toolkits, such as Athena, then moved to SGML-based markup languages in the quest for platform independence. HTML with forms would also seem a good choice here.
- Computer program, then the conventional mechanism is through some kind of functional interface, such as method invocation. The best candidate for this was the OMG's IDL. In conjunction with a distributed CORBA implementation, an object would be able to seemlessly communicate with other objects anywhere and bring its interface along with it.

These two approaches turn out to be very different on a number of levels. On the face of it, they are incompatible in any realistic way. Other than during program development, humans will not interface directly with objects through an IDL interface, especially naive users unfamiliar with the interface. Even more absurd would be the notion of humans communicating with each other through IDL defined interfaces. Conversely, it is highly unlikely objects will communicate with each other using HTML forms. This is, at present, a common way to support human/object communication (through CGI scripts), but even there it is unwieldy.

In an information-rich, networked universe, supporting both mechanisms is very onerous. For an object to communicate with both other objects and humans, it effectively needs two user interfaces. Just as a matter of limited resources, it is unlikely both will be complete, and there is also the problem of possible inconsistencies between them. Requiring two UIs is an undesireable feature from an engineering point of view.

The sender of an object may have a clear idea of the object's goals at the recipient, but to the recipient the object is also a resource to be manipulated for the recipient's purposes. A server may send a Java applet to a client to accept information for a purchase order, but the client will also want to log the transaction locally, and perhaps have the purchase order approved by local financial systems before it is finally submitted. And the client's interest does not end there; it is important to ensure the appropriate information is transmitted back to the server. Because the interface between the local applet and server is essentially private, and because a malicious applet can internally alter itself so the malicious code is garbage collected, the client has good reason to know exactly what purchase information passed to the server[4].

## 3 DSLs for distributed applications

We consider domain specific languages as the means to resolve the heterogeneity issue. In this paradigm, an object's interfaces are the languages in which it expresses the information it carries and its processing requirements.These domain specific languages are normally small, often much smaller than HTML, a popular domain-specific language for displaying simple hypertext. Whether a particular language is Turing-complete will depend completely on the particular language, but it is more likely the implementation language will be than the language itself. In our approach we separate language semantics into two levels:

- The *abstract* semantics, corresponding to the objects in the domain itself, without any regard for actual implementation. This corresponds to what is generally considered as semantics.
- the *operational*, or *concrete*, semantics, corresponding to however the recipient processes that message. We will discuss how this processing might be done, and how we have done it, but this does not presume that all message recipients will do likewise. This looseness is one of the strong points of our approach, as it allows a variety of different recipients to be swapped in and out of a distributed application.

In the "usual" programming language, the abstract semantics is usually straightforward (the syntax was created to reflect the semantics) and the operational semantics is simply the implementation of the abstract semantics on a particular machine.

In the world of DSLs, we can still consider the operational semantics to implement the abstract semantics on a machine, but we must enlarge somewhat our concept of machine. Mawl[9], for example, is implemented on an HTML/HTTP "machine." If we generally consider the receiving entity as a machine, our view of "machine" becomes quite broad. The DSL implementer may view a corporation as a machine with workflows, email to be sent, database entries to read, store, or update, etc., if that is the domain. Or the domain may be far smaller, like a card game. Any particular site might actually have many machines defined for the different ways they might want to handle the incoming information.

Our approach is a social model. Exchanged strings represent actions by the various parties. The closest analogy is with speech acts, as first discussed by Austin[1] and then extended by Searle[12]. Traditional logic has always considered sentences to be statements about the state of the world. As such they could be assigned truth values based on their correspondance with actual fact. Austin was the first to notice that some statements were actually acts themselves, and their utterance changes the state of the world. Austin called these speech acts. The classic example of a speach act is the marriage ceremony. At the end of the ceremony, both parties say "I do." These two utterances initiate the marriage, so they have, in fact, changed the state of the world.

According to Searle, for a speach act to be successful, it not only needs the right syntax, it also needs the appropriate context. The marriage speech acts, at least in the United States, can only be performed by unmarried individuals in the presence of an authorized individual, such as a justice of the peace. If performed by actors during a show, then it is not a valid as a marriage creating speech act. It is still a speech act, but one of a different kind.

We can take a similar approach using model theory. Although there may be a general domain model, each message is interpreted using a local model (and possibly more than one, if there are multiple operations to be performed). The net effect of interpreting a message using the local model, however, must be congruent with the abstract model. Accepting a purchase order should eventually lead to a shipment and a bill.

Model theory also leads, in a roundabout way, to justifying the application of DSLs. Just as there may be many models for a particular language, there may be many languages to express a particular model. For each of these languages there is an interpretation function to describe the mapping between it and the model. The interpretation functions for programming languages usually appear relatively straightforward. They describe how statements in the language function in general. However the interpretation function from a program in a traditional programming language to a particular idiosyncratic domain will be far more complex. Given that the interpretation function can vary in complexity, we would argue the appropriate language for any domain is the one with the simplest interpretation function. This is certainly the case where the language syntax corresponds directly with domain semantics so there is a one-to-one correspondance between language elements and the corresponding domains.

We've sidestepped the issue of implementing the language, or even of the variety of implementations corresponding to the different operational semantics we might require. But we've subdivided the initial problem – how to implement a variety of apparently unrelated applications in a particular problem domain – into two smaller problems:

- What is the language that best expresses the semantics of the domain.
- How can we implement this language to express the various operational semantics we need.

We maintain this is simpler. The first item is mostly a design issue, which needs to be addressed in any case. For the second one, each operational semantics corresponds roughly to an application, but it is now a mapping from a particular language to a particular implementation. Most of the operational semantics will fall into a small group, such as GUI, storage and retrieval, etc. Since we have subsumed the different problem domains under a particular structure (a language) and have now factored the different ways we'll need to manipulate those structures, we can attack each area separately (how to display a message, how to decompose and store a message, etc.). Our prefered method would be to use metalanguages to describe these mappings.
We can look at objects in message-passing based languagas, such as C++ and Java, as implicitly being language processors as well. In these languages, each object has one or more interfaces it presents to the rest of the world (C++ objects have one based on its class, Java objects can have several). Each interface describes a set of messages, called methods, accepted by the object. We can consider the set of methods as the alphabet of the object's language. After creation, an object receives a (potentially infinite) list of method invocations. Methods not in the alphabet are gibberish. (Dynamic languages, like SmallTalk, may have a default means to handle this; statically typed ones catch these at the compilation stage.) Otherwise the list of messages form a string in the language defined by the interface. Interface definition languages for object oriented languages currently specify no more that the alphabet, so any string (sequence of method invocations) is declared valid even though objects do not necessarily accept all sequences.

## 4 SGML/XML as DSL metagrammar

We have relied heavily on SGML, the Standard Generalized Markup Language, as the metagrammar for defining our various DSLs. SGML has some important characteristics which make it a candidate for the role:

```
<!ELEMENT element-name ((elem1, elem2)+ |
                        (elem3, elem4)*)>
```

Figure 1: Element definition

- It is an existing international standard already used to mark up terabytes of information, much of which may be interesting for the kinds of applications under consideration.
- Although rather complex, a number of parsers are available. XML, a simplified version of SGML designed for Web delivery is designed to be simple to parse.
- It is LL(1), as we will discuss later.
- Most important, SGML was designed to enable a complete break between syntax and semantics through its promulgation of logical, or descriptive, markup.
- SGML is also the metagrammar in which HTML is defined, so it will look familiar to people who have read Web document sources.

In descriptive markup, a tag designates what it is, not how it should be shown. For example, the `<date>` tag in the fragment `<date>10/29/1999</date>` indicates that the string is to be interpreted as a date not, for example, a part number. How it is to be displayed must be designated elsewhere, either in a display application or a stylesheet. However different applications - for database storage, for display, etc. - can all use the presence of the `<date>` tag.
SGML/XML does not use Backus-Nauer forms for defining grammars. An SGML rule is called an element definition. An example is given in figure 1. It has three parts:

1. The `ELEMENT` keyword to indicate this is an element definition.
2. The element name (the left hand side of a BNF production).
3. The *content model*, a regular expression what the internal contents of the element are. Where the element contains text, this is designted by the `#PCDATA` keyword.

We will give an example of a full grammar when explaining the Bridge application.

```
<!ELEMENT bridge (player+, deal,
                 bidding, dummy, play)>
<!ELEMENT player #EMPTY>
<!ATTLIST player position (north | south |
                           east  | west)
                                 #REQUIRED
                 name     cdata  #required>
<!ELEMENT deal (card+)>
<!ELEMENT card #empty>
<!ATTLIST card suit (spades | hearts |
                     diamonds | clubs)
                                #REQUIRED
               face cdata #REQUIRED>
<!ELEMENT bidding (bid | pass)>
<!ELEMENT bid #empty>
<!ATTLIST bid  suit (spades | hearts |
                     diamonds | clubs |
                     no-trump) #REQUIRED
               tricks cdata #REQUIRED>
<!ELEMENT pass #empty>
<!ELEMENT dummy (card+)>
<!ELEMENT play (trick+)>
<!ELEMENT trick (card, card, card, card)>
```

Figure 2: Grammar for Bridge Game

## 5 An example – the game of Bridge

A straightforward distributed application we have tried is a Bridge tournament. Bridge is interesting because it has both an interactive component and a multilayered architecture.
Bridge is the most popular of a number of games all based on the same basic procedure. In all of them:

1. Dealer deals the cards.
2. The parties bid to determine a *trump* suit and who will play the first card.
3. The "dummy," the partner of the highest bidder, lays his cards on the table.
4. The cards are played in a series of rounds. Each round is started by the winner of the previous round.
5. The play is scored.

Other families of card games are a variation on this, such as poker, where players can exchange some of their cards before bidding.
We can write a grammar to cover the context free aspects of this schema, as in figure 2. Some aspects of a game are necessarily context sensitive (such as not playing the same card twice) – these are handled by the agents, who or whatever they may be. It is also possible to create another meta-language to specify some of these context sensitivities (such as most steps proceed in a round robin fashion, or play is selection without replacement).
A correct and complete game of bridge 3 is a string in this language, although (due to context sensitivity) not all strings are correct games. In a distributed game of bridge:

```
<bridge>
<player position = ``north''
        name = ``author''> ...
<deal><card suit = ``spades''
           face = ``king''> ... </deal>
<bidding><bid suit = ``hearts''
           tricks = ``2''> ... <pass></bidding>
<dummy><card suit = ``clubs''
            face = ``2''>...</dummy>
<play><trick><card suit = ``hears''
                   face = ``6''>...</trick>
      ...
</play>
</bridge>
```

Figure 3: Schematic Brige Game

1. The actions of all the parties will, jointly, create a correct string in the language.
2. Each party will receive (and interpret) a correct bridge string. We can guarantee this because of the nature of bridge. There are other domains where only one party can determine that a correct string has been produced. There may also be domains where no party actually sees the final string (it may be distributed), but there is enough information to verify it is correct.
3. A knowledgeable party can view the resulting string and determine that it was, indeed, a correct game.

When playing bridge, the dealer generates the first part of the game by passing out the cards. Each player receives an open game start tag and then their 13 cards. If the entire game is automated, these may come in one message. On the other hand, if the dealer is human, the cards will be dealt one at a time, so the 13 cards will come in 13 messages. After this comes the bidding. Bidding terminates when three consecutive players pass. Each player, in turn, sends its bids to all the others, ensuring each player sees the same (correct) sequence. As the bids must escalate, this either requires some checking from the agent, or the grammar will drastically increase in size (since the number of possible games is actually finite, we could use a finite state machine, but this would not be terribly practical). It is easy enough to check the resulting string. Next comes the laying out of the dummy's hand – another 13 cards. These are sent by the dummy to the all the players (it also sends to itself for completeness). Finally the actual play begins. With each round, the leading player starts the round and plays its card, followed by the others. Again these are sent to all the players until the game terminates. At the end of the game, each player will have received a correct string. They will be the same except for the initial cards dealt.

As long as the players generate their pieces of the string correctly, it does not matter if they are computational objects or human. Non-participants or judges can listen in and follow that part of the game open to the public. The string can also be stored, printed, spoken, compared with other games, or otherwise manipulated.

If we want to compare computational and human agents, we can see that support in either direction is incremental. In the former case, the string only needs to be parsed into a data structure. Beyond that is the need to produce agents capable of playing bridge at whatever level is desired. This is outside the scope of this paper. In the human case, the interface can be as simple or complex as desired. A human could function with a simple, command line interface for Bridge, reading the messages directly and typing in bids and cards. This approach rapidly reaches a point of diminishing returns as the information becomes more complex, so a more sophisticated user interface is required.

Because the language is public, any client understanding the language is a valid participant. Whoever defines the Bridge language has created a public protocol. Bridge players can build their own clients or retrieve them from anywhere.

Maintaining the string representation down to the lowest level can facilitate *ad hoc* collaboration. Suppose, while playing bridge, I retrieve a bridge advisor program. Somehow I need to communicate the current state of the game to the advisor, and it needs to be able to give me feedback. If a traditional OO approach is taken, access to the GUI is encapsulated in the bridge client through a tangle of widgets and callbacks. Either the bridge client has a separate interface for other computational agents (such as a set of methods) or the information must be entered by hand. The second case is laborious and labor prone. The first, as we mentioned above, requires the client to have two interfaces - the GUI and the method. On the other hand, if the GUI, or some component of it, is really just a mapping from the current string to widgets, the advisor can retrieve the string directly from the GUI without requiring any communication with the bridge client. Ideally, the advisor can even borrow the display mapping to show the user alternative scenarios. Since the callback of playing a card is to send some tokens, it would even be possible to develop higher level tools to redirect the user's choice through the advisor (particularly if the advisor is also a tutor).

While the game string itself resulting from several processes interacting, that same string can be both program and datastructure to other applications. For example it is a program to a bridge game pretty-printer or DBMS storage routine. It is a data structure to any query facility trying to analyze the game

## 6 Implementation architecture

In discussing implementation, we will separate architectural issues into two parts, language considerations and application considerations.

We can also distinguish two broad application types - one in which processing is done continuous with the transmission of tokens (the bridge game) and one in which the entire language string is received by the client before any processing needs to be done (such as a purchase order).

### 6.1 The importance of top-down parsing

The flow of the bridge application – each player adding its piece to the construction of the final string – requires the use of an LL(1) grammar and parser for the language, as opposed to the more common LR(1) or LALR(1) grammars. These latter grammars lead to bottom up parses, while LL(1) grammars have top-down parses. In the LL parse, the parser knows which production it is entering as each token is encountered. This is imperative for the game, since the client always needs to know which state it is in to perform correctly. The LR parser, in contrast, knows which production it is leaving when the last token for the production has been seen. In other words, the parse will report to the application what *has* happened, not what *will* happen.

LL(k) parsing, in general, has been deprecated until recently because the need for a variable amount of lookahead. However, [11] has shown how to minimize the lookahead. As most rules seem to be LL(1), this makes LL(k) parsing very attractive.

An LR grammar, by contrast, appears able to support the purchase order application since the entire message should be available at the client before parsing begins. However this is not necessarily true; it might be that the document as sent contains references to other information which could be expanded in place if necessary. In other words, the sender leaves it to the receiver to determine if the additional information is necessary. For example, only a part of an item's record might be sent. If the parse is top down, the application will have more information about the message when it reaches the reference than if the parse is bottom up (this does not mean there will necessarily be sufficient information - it may be the information is necessary for something later in the document). However in a top-down parse, the additional information can be retrieved at the point the reference is encountered. In a bottom-up parse it is more likely that part of the parse would need to be discarded when it is time to retrieve the additional information. I would also argue that queries against a message are also easier in the LL scenario for the same reason. The more elaborate the path on a query, the more it resembles a parse in which most of the document is discarded. An LL grammar makes it more likely there will be a reasonable path through the document to the desired information.

### 6.2 Interpreter structure

The application architecture we have used is heavily influenced by our choice of Scheme for implementing our initial mobile object language. Lisp dialects are generally LL(1) because of the use of S-expressions. The feature facilitates the development of the various Lisp macro systems. A macro, in essence, is a function whose parameters are not evaluated. The body of the macro can rewrite the parameters (which may be a large chunk of code) and then have the rewritten code evaluated in place of the original code. This implies a top down evaluation. The other interesting aspect of Scheme is its support for nested lexical scopes (closures) and first class functions. By treating the parser itself as a coprocess, we have been able to write top down interpreters with several levels of nesting.

We essentially use an event-driven model, with events based on the generic identifiers of the the tags. Event handlers are grouped in lexically scoped groups. Each handler has three sections:

1. A pretraversal section for processing related to the current node in the parse tree. For example, suppose we had a purchase order and needed to keep track of the total cost for all the included items. Because of the lexical scoping, the code can simply define a variable here which will be accessed while the subtree is traversed. Also, because of the scoping, there is a possibility to use the same events recursively, so we can track prices of lists within lists.

2. A traversal section, in which the tree rooted at the current node is traversed. It is possible at this point to designate a new list of event handlers for nodes in the subtree. These event handlers are only required for events whose processing changes in the context of the current node. Since the event handlers are nested, if a handler is not found in the current scope, outer scopes are searched until one is located. In particular, as these handlers are also lexically scoped within the pretraversal section, they can have access to variables created at that level. So, for the purchase order example, the variable defined in the pretraversal state is incremented here.

3. A post traversal section for final processing after the subtree has been traversed. In the purchase order example, we would now have the total amount of the items and can print, transmit, store, etc., that value as we wish.

An example of this structure is given in figure 4. Note that due to the lexical scoping it is possible to have one event for a card when it appears during the `deal`, and another when it is part of `play`.

## 7 Related Work

Another approach to communicating among agents, either contrasting or complimentary depending on implementation, is given by KIF/KQML[7, 6]. This effort has grown out of the AI segment of the agent community. KIF is a predicate calculus based language for encoding ontologies – exhaustive analyses of the information in a particular domain. When used as a communication language, small Lisp-like programs are sent and executed remotely. KQML is a protocol for wrapping inter agent messages based on speech act theory. KIF seems to be very complimentary with our approach when considered as a domain specification language. The model must be described one way or another. However, while

```
('bridge
   (let ((var1 1)
 ...)
     (startup code)
     (event-list
       'deal (lambda (event)
               (let ((...))
                 ...
                 (event-list
                   'card
                     (lambda (event)
                       ...)
                   ...)
                 ...))
       ...
       'play (lambda (event)
               (let ((...))
                 ...
                 (event-list
                   'card
                     (lambda (event)
                       ...))
                 ...))
     (post-processing...)))
```

Figure 4: Interpreter Structure

the predicate calculus may be the best language for describing a domain, it is not necessarily the best syntax for making statements in that domain. The ideal DSL is the one with the most economical mapping between the syntax and semantics. The KIF community is concerned with producing ontologies - exhaustive analyses of domains. Our approach does not require that much overhead. KQML is a protocol for wrapping messages among agents heavily influenced by speech acts. Although it was developed with KIF in mind, it is agnostic concerning the language of the message and so could work with DSLs as well.

We see this approach as very compatible with *aspect-oriented programming*[8] and jargons[10]. Both of these look at decomposing a problem into a number of different aspects (or jargons), each with its own domain and language. A complete application is built by composing the program from statements in the different domain specific languages. The AOP effort weaves these together from separate programs, while the jargon approach allows the programmer to mix them in the source code. Both of these approaches are very implementation oriented. They represent possible alternative means of implementing local processing of a DSL message. The automatic weaving element of AOP is very attractive, as aspects could represent additional information the client could get from the server about the message without being completely dependent on the server for all aspects of processing.

The Shopbot[3] takes an explicitly AI view to integrating Web pages into a particular application. Shopbot is a shopping agent for the World Wide Web. Using a set of heuristics, it can learn how a shopping site is organized and help a user find products in a specific domain. This is a valid approach where:

- The problem domain is well structured but the messages are not.
- The difficulty of creating the agent is compensated by the number of uses.

Where the messags are structured in a domain appropriate way, the Shopbot approach is unnecessary. However, if it can convert unstructured messages to strucured ones, it could be integrated into such a system.

## 8 Conclusions

We have shown how DSLs can play an important role as the glue in multi-organizational distributed applications in the Internet. With the strong break between language definition and implementation semantics, we can map these languages into GUIs for human agents and functional interfaces for objects, so this approach subsumes both HTML and IDL and presents a unified communication paradigm.

This approach has many similarities to EDI[2]. In EDI the messages are standardized, so they could be considered a DSL, and each party is free to process them any way they require, so long as it is congruent with the abstract semantics, as determined by the international standards bodies. However, implementing EDI has been extremely difficult, even for large corporations. We suspect that our approach could simplify EDI implementation considerably.

We intend to apply this approach in a number of problem domains. The appearance and ready acceptance of XML indicates the Web requires this kind of approach.

## References

[1] J. L. Austin. *How to Do Things with Words.* Oxford University Press, 1962.

[2] Edward Cannon. *EDI Guide: a step by step approach.* Van Nostrand Reinhold, 1993.

[3] Robert Doorenbos et al. A scalable comparison-shopping agent for the world-wide web. In *Proceedings of the First International Conference on Autonomous Agents.* ACM, 1997.

[4] Matthew Fuchs. Beyond the write-only web. Technical report, 1995. http://cs.nyu.edu/phd_students/fuchs/in-long.ps.

[5] Matthew Fuchs. *Dreme: for Life in the Net.* PhD thesis, New York University, 1995.

[6] KQML Advisory Group. An overview of kqml: A knowledge query and manipulation language. Technical report, University of Maryland, 1992.

[7] Thomas Gruber. A translation approach to portable ontology specifications. Technical Report KSL 92-71, Knowledge Systems Laboratory, 1993.

[8] Gregor Kiczales et al. Aspect oriented programming. In *Proceedings of DSL '97.* University of Illinois Computer Science, 1997. http://www-sal.cs.uiuc.edu/ kamin/dsl.

[9] David A. Ladd and J. Christopher Ramming. Programming the web: An application-oriented language for hypermedia service programming. Technical report, 1997. http://www.bell-labs.com/project/MAWL/papers/Overview.html.

[10] Lloyd Nakatani and Mark Jones. Jargons and infocentrism. In *Proceedings of DSL '97.* University of Illinois Computer Science, 1997. http://www-sal.cs.uiuc.edu/ kamin/dsl.

[11] Terence Parr. *Obtaining Practical Variants of* LL(k) *and* LR(k) *for $k > 1$ by Splitting the Atomic* k-*Tuple.* PhD thesis, Purdue University, 1993.

[12] John Searle. *Speech Acts.* Cambridge University Press, 1969.