# A Case Study in Engineering a Knowledge Base for an Intelligent Personal Assistant

Vinay K. Chaudhri[1], Adam Cheyer[1], Richard Guili[1], Bill Jarrold[1], Karen L. Myers[1], John Niekarsz[2]
1. Artificial Intelligence Center, SRI International, Menlo Park, CA, 94025.
2. Center for the Study of Language and Information, Stanford University, Stanford, 94301

**Abstract.** We present a case study in engineering a large knowledge base to meet the requirements of a personal assistant. The agent is designed to function as part of a semantic desktop application with the goal of helping a user manage and organize his information as well as support the user in performing day today tasks. We discuss our development methodology and the knowledge engineering challenges we faced in the process.

## 1 Introduction

The use of ontologies, metadata annotations, and semantic web protocols on desktop computers will allow the integration of desktop applications and the web, enabling a much more focused and integrated personal information management as well as focused information distribution and collaboration on the Web beyond sending emails. In this paper, we present our experience in constructing a large knowledge base (KB) designed specifically to support a personal assistant called CALO (Cognitive Assistant that Learns and Organizes). CALO is a multidisciplinary project funded by DARPA to create cognitive software systems that can reason, learn from experience, be told what to do, explain what they are doing, reflect on their experience, and respond to surprises.

CALO KB uses an upper ontology called Component Library (CLIB) (Barker, Porter et al. 2001) and off-the-shelf standards such as *iCalendar,* as starting points and extends them to meet the requirements of CALO. The primary contribution of this paper is in providing a comprehensive description of the process of engineering a large knowledge base. CALO offers unique functionality by integrating an impressive array of AI technologies (more than 100 major software components spanning machine learning, planning, and reasoning written in about 10 different programming languages spanning Lisp, Prolog, and Java), and the effort of pulling them together into a semantic whole is unprecedented. The CALO KB effort as presented here has been a key ingredient in accomplishing the goal of semantic integration in CALO, and much can be learned from the description of this experience by others interested in undertaking efforts to construct large knowledge bases.

We begin this paper by identifying the knowledge requirements of CALO and then describe how we developed the KB. We then give an overview of the knowledge content, and discuss three ontological challenges in some detail. We give a review of tools that we used in the process and conclude with a comparison to related work and by identifying research issues suggested by this experience.

## 2 Knowledge Requirements in Project CALO

CALO's role is to know and do things for its user, and therefore it must have knowledge about the environment in which the user operates and the user's tasks. It is best to understand the knowledge requirements of CALO in terms of the major six functions it performs. Details of implementing the functions are not the focus of the present paper.

### 2.1 Organize and Manage Information

From a user's information about emails, contacts, calendar, files, and to-do lists, CALO learns an underlying relational model of the user's world that provides the basis for higher-level learning. The relational model contains information such as the projects a user works on, which project is associated with which email, the people a user works with, and in what capacity. Providing such functionality requires vocabulary to relate information across the email, contact records, calendar entries, information files, to-do lists, and people. Since much of the learned information is probabilistic, there is a need to provide a way to absorb and maintain this knowledge with the symbolic knowledge.

### 2.2 Prepare Information Products

CALO puts together a portfolio of information, for example, emails, files, and Web pages to support a project or a meeting. The vocabulary needed to support this functionality is similar to that required support the functionality to organize and manage information.

### 2.3 Observe and Mediate Interactions

CALO observes and mediates interactions whether they are electronic (in an email) or direct (in a meeting). For example, it can summarize, prioritize, and classify an email. CALO identifies the action items, and produces an annotated meeting record. During a meeting, CALO captures the action items that were identified, and produces an annotated meeting record. To support this functionality, we need vocabulary to specify priorities on emails, classifications for emails, and speech acts one may want to identify within an email. To support observation during a meeting, we need to provide representations for the dialog structure of a meeting, and for objects mentioned in the dialog, e.g., a Gantt chart.

### 2.4 Monitor and Manage Tasks

CALO aids the user with task management in two ways. First, it provides tools to assist a user in documenting and tracking 'to do' tasks for which she is responsible. Second, it can automatically perform a range of tasks that have been delegated to it by the user. This automation spans both frequently occurring, routine tasks (e.g., meeting scheduling, expense reimbursement) and tasks that are larger in scope and less precisely defined (e.g., arranging a client visit) and that require ongoing user interaction.

Support for both types of task management requires a vocabulary for specifying tasks that includes representation for the parameters, their types, and the roles that they play in achieving a task. In addition, 'life-cycle' properties of tasks must be tracked as the task is performed, by either the user or the system (e.g., task status, priority). Task automation further requires an explicit representation of the processes to be performed to achieve a task.

### 2.5 Schedule and Organize in Time

At a user's request, CALO can schedule meetings for the user by managing scheduling constraints, handling conflicts, and negotiating with other parties. To support this function, we need a representation of the schedule and scheduling constraints as well as a model of preferences that a user may have over individual scheduling requirements.

### 2.6 Acquire and Allocate Resources

CALO can discover new sources of information that are relevant to its activities. For example, it is capable of discovering new vendors that sell a particular product. It can also learn about the roles and expertise of various people and use them for answering questions. To meet this requirement, CALO must be able to extend its vocabulary as new sources of information are discovered.

## 3. Developing CALO Knowledge Base

The requirements identified in the previous section illustrate the range of knowledge that needs to be captured in the KB. Here, we describe the development process we used to meet these requirements.

### 3.1 Choosing a Starting Point

We chose the Component Library (CLIB) as the primary upper ontology for the following reasons: (1) CLIB provides a small set of carefully chosen representations with broad coverage of required concepts. (2) CLIB uses a STRIPS model for representing actions that seem close to what was needed for supporting the function of monitoring and managing tasks. (3) CLIB provides a well-thought-out model of communication that seems to be a good fit for an *Observe and Mediate Interactions* capability.

Numerous specialized vocabularies exist for modeling information such as emails, contacts, and calendars. Instead of reinventing, we reused them. Specifically, we made extensive use of the *iCalendar* (Dawson and Stenerson 1998) standard for representing the calendar and to-do information. We leveraged the DAML-Time ontology as an inspiration for representing time (Hobbs and Pan 2004). To develop ontologies for office products, we drew inspiration from online Web stores such as Gateway.com and CompUSA.com.

Given the heterogeneity in the system, it was clear that no single knowledge representation language was going to be adequate to meet all the needs. We accomplished the bulk of the ontology representation work using the Knowledge Machine (KM) representation language (Clark and Porter 1999). The choice of KM followed from the choice of using the CLIB as the upper ontology. Once developed, the ontology was exported into OWL via a KM to OWL translator. We chose OWL as an interchange language because several tools in the system were Java based, and several off-the-shelf tools were available to read OWL.

We represent knowledge for performing automated tasks in the SPARK procedure language (Morley and Myers 2004), which is similar to the hierarchical task network (HTN) representations used in many practical AI planning systems (Erol, Hendler et al. 1994). The SPARK language extends standard HTN languages through its use of a rich set of task types (e.g., achievement, performance, waiting) and advanced control constructs (conditionals, iteration). The expressiveness in SPARK was essential for representing the complex process structures necessary for accomplishing office tasks. We represent the uncertain knowledge extracted by the learning algorithms using weighted first-order logic rules that are processed using a Max-SAT solver. The weights are assigned by the learning modules, and the rules use vocabulary drawn from the CALO KB.

Thus, the CALO KB uses a mixture of representations each of which is well-suited for a specific function. These representations are semantically linked together by the vocabulary provided in the KB.

## 3.2 Knowledge Base Development Process

The CALO development team is large and distributed with over twenty research groups contributing to the project. Some contributors had never before worked with a formal KB. To initiate KB development, we solicited requirements from the contributors. The requirement specifications varied in form, depending on the background and experience of the contributors. Some contributors specified their requirements as a listing of concepts and relations, while others provided an axiomatic specification of their requirements. Knowledge engineers implemented these requirements.

In the early development phase, we stove for a large-scale reuse of existing ontologies. For example, we imported the whole *iCalendar* specification into our system. This led to a multitude of problems. It made the ontology very large, and the contributors complained that they had difficulty in finding the terms they were looking for, and it negatively impacted the system performance. Therefore, the initial reuse phase was followed by an ontology simplification phase where we retained only terms that were of direct use to the system. We determined such terms by looking for the usage of each term throughout the system's code base.

As the project proceeded, it became clear that the centralized KB development model needed to be relaxed as it was not possible for the knowledge engineering team to keep up with all the requests. Furthermore, members of the software development team needed to try out ontologies without yet fully committing to the ontology changes. Therefore, we switched to a two-stage model: Individual contributors took responsibility for a section of the KB; their changes were then reviewed by the knowledge engineering team before incorporating them throughout the system. We used Protégé for doing the distributed knowledge engineering work (Gennari, Musen et al. 2003).

Portions of the KB were localized to specific modules in the system and were accessed only via the shared vocabulary. For example, the KB includes a library of SPARK-based process models that provide a range of capabilities in the areas of visitor planning, meeting scheduling, expense reimbursement, and communication and coordination. These models were private to SPARK, but could be queried using the vocabulary provided by the ontology. (We discuss the query tools in a later section of the paper.)

## 4. Knowledge Engineering Challenges

In describing the knowledge engineering challenges we faced in developing the ontology, we begin this by giving an overview of the knowledge content and then explain in detail three specific technical problems that we addressed.

## 4.1 Overview of the Knowledge Content

The CALO ontology consists of a collection of ontologies that reference each other. Some of the key ontologies in the collection are ontologies for People, Organization, Calendar, Meetings, Contacts, Schedules, Tasks, and processes.

The *Person* ontology provides a basic representation of person that includes first name, last name, middle name, prefix, suffix, age, and sex. The *Contact* ontology specifies the vocabulary for ways to contact a person, e.g., postal addresses, phone number, and ZIP code. There can be multiple kinds of addresses, e.g., home address, work address, primary, secondary, and emergency. The *Organization* ontology is a collection of roles specifying the variety of roles people can play in an organization, such as manager, employee, program manager, job candidate, and vendor. For each of the roles, it also specifies its relevant properties.

The *Calendar* ontology provides a vocabulary for specifying calendar entries, their start and end times, whether they repeat, and the attendees for each entry. It references the *People* ontology and the *Time* ontology.

The *Meeting* ontology provides vocabulary for specifying different kinds of meetings (e.g., job interview, conferences), discussion topics, different roles in meetings (e.g., moderator, leader, listener), and different phases of a meeting (e.g., start, end, presentation, discussion). The *Task* ontology specifies different states a task could be in (e.g., initiated, terminated) and a specification of roles that different entities might play in a task.

The current ontology has about 1000 classes and 500 relations. The process model library contains about 50 process models that capture the execution of office tasks.

### 4.2 Leveraging Off-the-Shelf *iCalendar* Ontology

The *iCalendar* format is a standard (RFC 2445 or RFC2445 Syntax Reference) for calendar data exchange. The standard is sometimes referred to as "iCal", which also is the name of the Apple Computer calendar program that provides one implementation of the standard.

To incorporate the iCalendar standard into CALO KB, we undertook three steps: (1) pruning the relations needed by the application, (2) defining symbol name mappings, and (3) linking with the rest of the ontology. We begin with an overview of the content in the *iCalendar* standard, and then give more detail on each of the steps in the process.

The top-level object in *iCalendar* is the Calendaring and Scheduling Core Object. This is a collection of calendaring and scheduling information. Typically, this information will consist of a single *iCalendar* object. However, multiple *iCalendar* objects can be sequentially grouped together. The body of the *iCalendar* object (the icalbody) consists of a sequence of calendar properties and one or more calendar components. The calendar properties are attributes that apply to the calendar as a whole. The calendar components are collections of properties that express a particular calendar semantic. For example, the calendar component can specify an event, a to-do, a journal entry, time zone information, free/busy time information, or an alarm.

It is straightforward to define mappings from the *iCalendar* standard into classes, relations, and properties, which gives 6 classes, 35 relations, and 14 property values.

To support uniformity and usability, we use naming conventions in the CALO KB. For example, the *iCalendar* standard defines a slot called *calendar-dtstart* to denote the starting time of a meeting. If we use the naming conventions in the KB, this slot will map to *calendarEntryDTStartIs*. We defined mappings for the relation names in the KB so that we can retain the uniformity within KB, but at the same time be able to map this information out to other information sources that might use the *iCalendar* standard.

In several places the *iCalendar* ontology references things that are defined elsewhere in the ontology. For example, the *range* of the relation representing the attendee of a meeting is a *Person* defined elsewhere in the KB. The *range* of the relation defining the start of a meeting is *Time-Instant* that is defined as part of the DAML-Time ontology.

### 4.3 Representing Meetings

We designed the meeting ontology in the CALO KB to support the requirement *Observe and manage interactions*. The meeting ontology extends the model of communication in CLIB to support the needs of meetings that involve multimodal dialog. We review the model of communication in CLIB, and then discuss how we extended it. A more detailed description is available elsewhere (Niekrasz and Purver 2005).

#### 4.3.1 Model of Communication in CLIB

The model of communication in CLIB consists of three layers representing physical, symbolic, and informational components of individual communicative actions. The events in these three layers occur simultaneously, transforming the communicated domain-level *Information* into an encoded symbolic

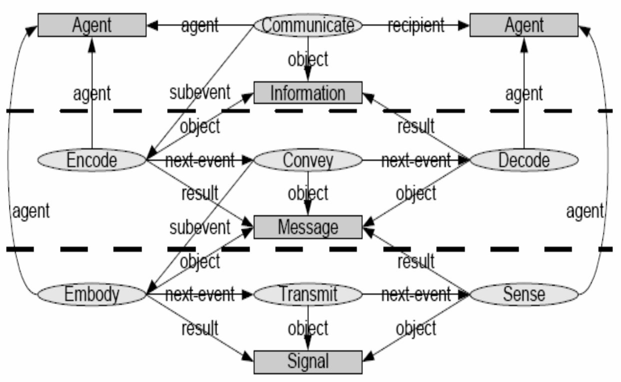

**Figure 1. Model of Communication in CLIB**

*Message*, and from this *Message* into a concrete signal. We show a graphical representation of these layers in Figure 1. Events are depicted using ovals and entities using darker rectangles. Arrows signify relations. The events Communicate, Convey, and Transmit correspond to the informational, symbolic, and physical layers.

### 4.3.2 Modeling Multimodal Communication

To represent a meeting with multimodal communication, we had to extend the basic model of communication in CLIB. First, the CLIB model assumes a one-to-one correspondence across the three layers. This assumption breaks down when there is multimodal co-expression of speech. To support this, we extended the Encode concept to produce multiple messages each in their own *Language*, and each of which can generate their own Signal in some *Medium.* Second, CLIB provides the concept of *Message* between the physical signal and its domain interpretation. We extended this to represent by defining a *Message* to be a *LinguisticUnit* that is built out of *LinuisticAtoms.* For written language, examples of *LinguisticAtoms* are Words and Sentences. Finally, we extended the communication roles in CLIB that naturally arise in meetings, for example, *Addressee* and *Overhearer.*

### 4.3.3 Modeling Discourse Structure

*D*iscourse structure allows us to express relationships among individual communication acts —both at the level of modeling the dialog structure and at the level of argumentation and decision making. To represent the dialog structure, we consider individual *Communicate* events as *dialogue moves,* expressed via membership of particular subclasses and with their interrelation expressed via the properties associated with these subclasses. For example, we define classes such as *Statement, Question, Backchannel,* and *Floorholder.* Each *Communicate* event can have an *antecedent.* The graph structure on *Communicate* events defined by the *antecedent* relation is limited to a tree. The argument structure is modeled at a level coarser than the individual *Communicate* acts considered in the discourse structure. For example, the argument structure is represented using actions such as *raising an issue, proposal, acceptance,* and *rejection.* Each action in the argument structure consists of a series of individual *communicate* acts.

### 4.3.4 Modeling the Meeting Activity

A meeting consists of subevents, the majority of which are Discourse events. Meetings may include non-Communicative acts e.g., note taking) and multiple discourses (e.g., simultaneous side conversations). Therefore, we provide two ways to segment a Meeting activity in a top-down, coarser-grained way: along a physical state or an agenda state. The physical state depends only on the Physical activities of the participants (e.g., sitting, standing, talking). The agenda state refers to the position within a previously defined meeting structure, whether specified explicitly as an agenda or implicitly via the known rules of order for formal meeting types.

## 4.4 Representing Tasks

The task ontology is represented in a fairly standard way, with an individual task class modeled in terms of a task type, a set of input and output parameters for the task, whether they are required or optional, and constraints on allowed input tasks.

Information about task instances that are currently active in the system, whether they are performed by the user or automated, is important for a wide range of functions within CALO. For instance, knowledge about the current set of user task instances is employed to focus activity recognition modules that seek to understand what the user is trying to accomplish at any point in time. Knowledge of overall user task workload can be used to inform the scheduling process. Information about status on task instances and resource usage is used to support execution monitoring and dynamic task reallocation. For this reason, it was necessary for us to define a system-wide vocabulary for representing task instances that could serve as the basis of semantic integration.

We derived the representation for task instances from the VTODO construct for ‘to dos’ in the *iCalendar* representation. We reuse some of the *iCalendar* constructs, drop some, and add new ones to meet our requirements.

For a task instance, we represent its intrinsic properties, its relationship to other tasks or entities elsewhere in the system, and information about its status and dynamics. The heart of the representation consists of *descriptive* properties of a task instance (such as a formal specification of the task, priority, documentation, source, location, and resource allocation and usage), *temporal* properties (such as creation time, start/completion time, deadline), and *state* properties (such as status).

We extended the *iCalendar* representation to include expected duration (to enable reasoning of projected task completion times), resources consumed (to enable effective resource management), creation information for a task such as the source (i.e., the person who created task) and the context (e.g., a meeting or an email), a possible result for a task, information about the delegation of tasks to other individuals, and *change management* properties (such as a modification history).

### 4.5 Ensuring Interoperability

The representation language KM is broadly expressive. To ensure interoperability for the information that had to be shared across modules, we had to limit the knowledge engineering efforts to use only the representation constructs offered by OWL. We consider two such examples here.

OWL does not support n-ary relationships. We need to represent signatures of the procedures used in the SPARK library in our ontology. Since the order of arguments to a procedure was significant, an n-ary relation representation would have been the most natural. But, because of the limitation of OWL, we had to reify each procedure parameter to indicate its position in the list of arguments to that procedure (Noy and Rector 2004).

In OWL, one cannot define subclasses of primitive data types such as STRING. There were several compelling situations where such a feature was critical. For example, a learning algorithm may deduce that a string such as “94025-3493” is an instance of the class of strings representing US Postal Codes, and that a string such as “650-555-1212” is an instance of the class representing phone numbers in United States. Once deduced, there is a need to enforce it as a constraint on the legal values of postal code and phone numbers. To support this requirement, we introduced a collection of classes termed as *Pseudo Ranges* that were not a subclass of the built-in OWL class String. For example, *PostalCodeString* is a pseudo range class that is a string, and defines legal strings for US Postal Code.

## 5. Deploying the Ontology

Recall that we developed the ontology using the KM representation language that was then translated into OWL for distribution. We distributed Javadoc-style documentation pages generated using OWLDOC, which is a Protégé plugin. There were at least two different classes of users of this ontology. First, the users simply loaded the whole OWL file into their modules. Second, the users did not load the OWL file, but simply made references to the terms in the ontology. For both cases, we needed to provide ways using which users and system modules could access knowledge in the system, update it, and evolve the system as the ontology changes.

We supported access to the distributed knowledge in the system a query manager using which users and system modules can pose their queries to the knowledge system using the KB vocabulary (Ambite, Chaudhri et al. 2006). The query manager then decomposes the query into pieces that can be answered by a module, queries them, and produces the final answer. This turns out to be a useful service because the information access within the whole system is transparent to the users.

While it made good sense to provide transparent access for querying the knowledge in the systems, a similar model for updating the knowledge in the system was not feasible because the updates require access to knowledge that is at a level deeper than what can be exposed using the vocabulary in the ontology. For example, to update a SPARK procedure, one needs to operate at the level of full expressiveness of the native language and SPARK, and thus a transparent solution is infeasible. Therefore, for updating the knowledge in the system, custom update modules needed to be provided.

Even though the updates are decentralized, there is a need for modules to know about changes in other modules. To support that requirement, we devised a publish-and-subscribe scheme, using which the modules can advertise their updates. The encoding of messages in the publish-and-describe facility uses the vocabulary from the ontology.

During the course of the development, there are frequent changes in the ontology. These changes will impact the instance data stored in the system. Therefore, we implemented a program called Simple Ontology Update Program (SOUP) that accepts old and new ontology as input, computes the differences, and updates the instances in the system as the ontology changes.

Recall that one of the functions of CALO is to *Acquire and Allocate Resources.* In the process of doing this, the system may learn new classes and relations that must be added to the ontology at runtime. To support this, we implemented an ontology update module that can add new classes and relations to the knowledge base. The API for this update was modeled after the OKBC tell language.

## 6. Open Research Challenges

The development work on the knowledge base is not yetcomplete. In fact, the engineering and infrastructure work done so far has laid the foundation for investigating compelling research questions that have not been possible so far. We consider here a few such challenges that also suggest directions for future work.

In recent years, we have seen the phenomenon of tagging information resources become very popular. Users tag an information resource with keywords describing their content, which can be used by a search tool for retrieval of those documents. Such tagging is used by several tools within CALO. Furthermore, the machine learning algorithms annotate the documents on a user's desktop using keywords. Currently, there is no relationship between the tags or keywords and the ontology. Furthermore, for understanding meetings, there is need to go from the natural language utterances and the terms in the ontology.

Tags and keywords are usually words in natural language. The CLIB has links from Wordnet to its concepts that gives us mappings from the words in natural language to the concepts in the ontology. By making use of such links one can perform inferences using the ontology that could not be done using the words alone.

In the current CALO system, most of the learning components perform a predefined set of tasks. The learning capability in the system is, however, much more general than that to a point where the system could identify what it should learn and then go about learning it. A natural progression of such a capability in increasing order of difficulty is as follows. (1) A CALO developer can apply a learning method to a new problem by writing a specification of the learning problem. (2) A CALO user specifies a high-level goal, and CALO can identify how to realize that goal by applying the learning methods at its disposal. (3) While observing the user, CALO can determine what it should learn, and it learns it. Achieving such capabilities requires developing an ontology of learning methods, and a way of reasoning with it.

## 7. Comparison to Related Work

One of the distinguishing features of the work reported here is that its development is strictly driven by the needs of a cognitive agent designed to function in an office environment. In some sense, it makes it less general than other ontologies such as Cyc or SUMO, but on the other hand, given the wide applicability of the office work, its content is of interest to a large number of end users.

Another important distinguishing feature is the CALO ontology's adaptability. As a specific example, the CLIB generic communication model is an example of a highly reusable chunk of CLIB. We showed in this paper how we were able to adapt this representation to apply to a multimodal representation of meetings.

Another distinguishing feature of the CALO KB from others is the variety of languages and platforms used. Some of the knowledge engineering work is done in KM, and this gets automatically translated to OWL. Because OWL is the universal language of the semantic Web, we maximize accessibility. We develop in KM, so we can allow ourselves maximum expressiveness—for example, rules, which OWL does not have—where we need them. Multitudes of reasoning platforms access this ontology. These reasoning platforms include such languages as Lisp, Java, Prolog, and C. The reasoning platforms also have many different styles of reasoning ranging from declarative reasoning to procedural reasoning and process execution. In short, the CALO ontology embraces and integrates broad range of heterogeneous elements.

## 8. Summary and Conclusions

We presented a case study in engineering a large knowledge base to meet the requirements of a cognitive agent. The cognitive agent is designed to perform tasks such as *Organize and Manage Information* and *Monitor and Execute Tasks.* This created a wide set of requirements for the vocabulary and was further complicated by a large distributed team using many different platforms. We provided a comprehensive description of our development process. We described specific knowledge engineering challenges in reusing an existing ontology, and in modeling multimodal meetings and tasks. The knowledge base has served as the basis of semantic integration in the CALO system built out of more than 100 artificial intelligence (AI) modules written in about 10 different programming languages. We believe that this work is an important contribution to the state of practice in engineering large knowledge systems, and can be instructive to others striving toward similar goals.

**Acknowledgments**

This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under ContractNo. NBCHD030010. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the DARPA or the Department of Interior-National Business Center (DOI-NBC).

## References

Ambite, J. L., V. K. Chaudhri, et al. (2006). Integration of Heterogeneous Knowledge Sources in the CALO Query Manager. Proceedings of the Innovative Applications of Artificial Intelligence Conference.
Barker, K., B. Porter, et al. (2001). A Library of Generic Concepts for Composing Knowledge Bases. Proc. 1st Int Conf on Knowledge Capture (K-Cap'01): 14--21.
Clark, P. and B. Porter. (1999). "KM -- The Knowledge Machine: Users Manual." from The system code and documentation are available at http://www.cs.utexas.edu/users/mfkb/km.html.
Dawson, F. and D. Stenerson. (1998). "Internet Calendaring and Scheduling Core Object Specification." from http://tools.ietf.org/html/2445.
Erol, K., J. Hendler, et al. (1994). Semantics for Hierarchical Task-Network Planning, University of Maryland at College Park.
Gennari, J., M. A. Musen, et al. (2003). "The Evolution of Protege: An Environment for Knowledge-Based Systems Development." International Journal of Human-Computer Interaction 58(1): 89-123.
Hobbs, J. and F. Pan (2004). "An Ontology of Time for the Semantic Web." ACM Transactions on Asian Language Information Processing 3:1.
Morley, D. and K. Myers (2004). The SPARK Agent Framework. International Conference on Autonomous Agents and Multi-agent Systems.
Niekrasz, J. and M. Purver (2005). A Multimodal Discourse Ontology for Meeting Understanding. In Machine Learning for Multimodal Interaction: Second International Workshop, MLMI 2005, Edinburgh, UK, Springer Verlag.
Noy, N. and A. Rector. (2004). "Defining N-ary Relations on the Semantic Web." from http://www.w3.org/TR/swbp-n-aryRelations/.