

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

Vinay K. Chaudhri<sup>1</sup>, Adam Cheyer<sup>1</sup>, Richard Guili<sup>1</sup>, Bill Jarrold<sup>1</sup>,  
Karen L. Myers<sup>1</sup>, John Niekrasz<sup>2</sup>

<sup>1</sup> Artificial Intelligence Center, SRI International, Menlo Park, CA, 94025

<sup>2</sup> Center for the Study of Language and Information, Stanford University, Stanford, 94301

**Abstract.** We present a case study in engineering a knowledge base to meet the requirements of an intelligent personal assistant. The assistant 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 supporting the user in performing tasks. We describe the knowledge base development process, the knowledge engineering challenges we faced in the process and our solutions to them, and important lessons learned during the process.

**Keywords:** Ontologies, Case Study, Intelligent Assistants, Semantic Desktop

## 1 Introduction

The Internet, electronic mail, and the Web have revolutionized the way people communicate and collaborate. Along with the increased amount of information and interaction that these tools enable comes the potential for problems such as information overload and thrashing that can result from too much multitasking. Fortunately, the information provided by these tools can also be exploited to provide intelligent assistance to a user. CALO (Cognitive Assistant that Learns and Organizes) is an intelligent personal assistant that can reason, learn from experience, and accept instruction in order to aid a user in dealing with information and task overload. In this paper, we present our experiences in constructing a knowledge base (KB) for CALO. We discuss the technical challenges, our solutions to them, and the lessons learned.

The CALO KB uses as starting points an upper ontology called the Component Library (CLib) [1] and off-the-shelf standards such as *iCalendar*, extending them to meet the requirements of CALO. The primary contribution of this paper is a comprehensive description of the process of engineering a KB for a personal assistant. CALO offers unique functionality by integrating an impressive array of AI technologies (more than 100 major software components that span machine learning, planning, and reasoning included); the effort of pulling them together into a semantic whole is unprecedented. The CALO KB effort 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 similar efforts.

We begin this paper by identifying the knowledge requirements for CALO, and then describe how we developed a KB to address those requirements. We then give an overview of the knowledge content, and discuss three knowledge engineering problems that we faced: knowledge reuse, representing meetings, and representing tasks. We give a review of tools that we used in the process and conclude by comparing to related work and identifying research directions and lessons learned through this experience.

## **2 Knowledge Requirements in Project CALO**

CALO's role is to know and do things for its user. As such, it must have knowledge about the environment in which the user operates and of the user's tasks. It is best to understand the knowledge requirements of CALO in terms of the six major functions it performs, which we describe below. Details of implementing the functions are not the focus of the present paper and can be found elsewhere [2-4].

### **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 email, contact records, chat discussions, calendar entries, information files, web pages, to-do lists, and people. Since much of the learned hypotheses about these entities is probabilistic, there is a need to provide a way to absorb and maintain this knowledge relative to a consistent background of symbolic knowledge [5].

### **2.2 Prepare Information Products**

CALO puts together a portfolio of information—for example, emails, files, and Web pages—to support a project, task, or meeting. CALO can also help the user create or extend PowerPoint presentations on a given topic. The vocabulary needed to support this functionality is similar to what is required to support the functionality to organize and manage information.

### **2.3 Observe and Mediate Interactions**

CALO observes and mediates human interactions whether they are electronic (in an email) or face to face (in a meeting). For example, it can summarize, prioritize, and classify an email. During a meeting, CALO captures the action items that were identified, and produces an annotated meeting record. To support this functionality, we need a vocabulary to specify priorities on emails, classifications for emails, and speech acts one may want to identify within an email thread. 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, for example, a Gantt chart depicting tasks, resources, and timelines.

### **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), requiring 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 must be tracked as a task is performed, by either the user or the system (e.g., task status, priority). Task automation 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 that information to answer 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 Knowledge Representation Framework

We chose the CLib as the primary upper ontology for the following reasons: (1) CLib provides a small set of carefully chosen representations that are domain independent, reusable, and composable, (2) CLib uses a STRIPS model for representing actions that is close to what was needed for supporting the function of monitoring and managing tasks [6,7], and (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 leveraged such existing vocabularies when possible. Specifically, we made extensive use of the *iCalendar* [8] standard for representing the calendar and to-do information. We leveraged the DAML-Time ontology as an inspiration for representing time [9]. 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 requirements. We accomplished the bulk of the knowledge representation work using the Knowledge Machine (KM) representation language [10]. The choice of KM followed from the choice of using the CLib.

We represent knowledge for performing automated tasks in the SPARK procedure language [11], which is similar to the hierarchical task network (HTN) representations used in many practical AI planning systems [12]. 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. As a result, the storage of the KB in the system is inherently heterogeneous. Portions of the KB reside in specific software modules that are best suited to store them. For example, we store the personal information of the user in an RDF store, process models in SPARK, and the weighted rules in a specialized reasoner.

Given such heterogeneity in representation, it did not make sense to support translations across each pair of representations. For example, even if we were to translate the process models in SPARK into some standard interlingua, most other modules in the system lacked the processing capability to reason with much of the resulting knowledge. Therefore, we had to look for a representation suitable to serve as a semantic interlingua for the system. The semantic interlingua had two parts: the vocabulary and the representation language. We used the vocabulary from the CALO KB, and OWL [13] as the representation language for the semantic interlingua. By vocabulary of the KB, we mean class-subclass structure, relations and their type constraints, individuals and their types, and slot values.

Thus, at the level of representation language, OWL served as the interlingua among various modules of the project. We could not simply use OWL natively for all modules as it does not offer all the representation features needed, for example, a process modeling language, rules, and uncertainty representation. However, it does offer suitable expressivity for our inter-module requirements of being able to represent only the vocabulary of the KB. We cannot, however, translate the detailed representation of process models into OWL. To exchange information between SPARK and the rest of the system, we use a triple-based notation called Portable Process Language [14] that naturally maps a subset of the SPARK representation into OWL and is adequate for information exchange between SPARK and other modules of the system. A comparison between the Portable Process language and OWL is available elsewhere [14]. Details of the translation of the KB into OWL are also available in a separate paper [15].

### **3.2 Knowledge Base Development Process**

The CALO development team is large and distributed with over twenty-five research groups across the country 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 list of concepts and relations, while others provided an axiomatic specification. Knowledge engineers implemented these requirements.

In the early development phase, we strove for large-scale reuse of existing knowledge representations. For example, we imported the whole *iCalendar* specification into our system. We did this because we needed a representation of calendars, and considerable thought had already been invested into designing the representation in *iCalendar*. A large scale reuse of *iCalendar* led to a multitude of problems. It made the KB very large, and the contributors complained that they had difficulty in finding the terms they were looking for. It also negatively impacted system performance. Therefore, the initial reuse phase was followed by a KB 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 code base.

Our approach to extending the KB was based on the CLib philosophy: create new representations by composing existing domain-independent representations [1]. The representation of meetings is a vivid demonstration of this approach.

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 doing a change in the KB without fully committing to the change. 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 they were incorporated throughout the system. We used Protégé [16] for doing the distributed knowledge engineering work. Protégé was selected because it offers the necessary editing functionality and was readily available off the shelf for free. [16]. At the time of this writing about 70% of the ontology originated via development in KM (and is automatically translated to OWL) and 30% originated via Protégé.

## 4 Knowledge Engineering Challenges

In describing the knowledge engineering challenges, we first give 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 KB represents multiple, interlinked aspects of office knowledge, for example, Persons, Organizations, Calendars, Meetings, Files, Contacts, Schedules, Tasks, and Processes.

We provide a basic representation of person that includes first name, last name, middle name, prefix, suffix, age, and sex. A person has *Contacts* that specify ways to contact a person, for example, postal addresses, phone number, and ZIP code. There can be multiple kinds of addresses, for example, home, work, primary, secondary, and emergency. For an *Organization*, we specify a collection of roles characterizing the functions that people can fill in an organization, such as manager, employee, program manager, job candidate, and vendor. For each of the roles, we specify its relevant properties, the person or organization playing the role, the time duration for which that role was played, and so on.

To represent *Calendar* information, we provide a vocabulary for specifying calendar entries, their start and end times, whether they repeat, and the attendees for each entry. We represent different kinds of meetings (e.g., job interview, conference), discussion topics, different roles in meetings (e.g., moderator, leader, listener), and different phases of a meeting (e.g., start, end, presentation, discussion). We represent 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 KB has about 1000 classes and 500 relations. The process model library contains about 50 process models that capture processes for common office tasks.

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

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 KB. 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 grouped together sequentially. 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 the KB, but at the same time be able to map this information to other information sources that use the *iCalendar* standard.

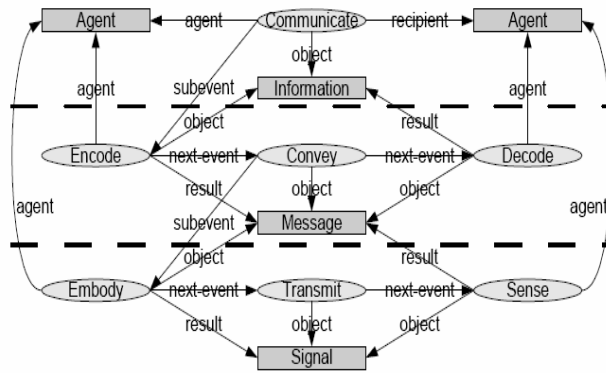


Figure 1. Model of Communication in CLIB.

### 4.3 Representing Meetings

We represent meetings in the CALO KB to support the requirement *Observe and manage interactions*. The meeting representation 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 of the representation of meetings is available elsewhere [17].

**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 *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.

**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 its own *Language*, and each of which can generate its own *Signal* in some *Medium*. Second, CLib provides the concept of *Message* between the physical signal and its domain interpretation. We extended *Message* by defining it to be a *LinguisticUnit* that is built out of *LinguisticAtoms*. For written language, examples of *LinguisticAtoms* are Words and

Sentences. Finally, we extended the communication roles in CLib that arise in meetings, for example, *Addressee* and *Overhearer*.

**Modeling Discourse Structure.** Discourse 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.

**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 representation is fairly standard, 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 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 for semantic integration.

We derived the representation for task instances from the VTODO construct for ‘to dos’ in the *iCalendar* representation. We reused some of the *iCalendar* constructs, dropped some, and added 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).

## 5 Deploying the KB

Recall that we developed the KB using the KM representation language, which was then translated into OWL for distribution. We distributed Javadoc-style documentation pages generated using OWLDOC, which is a Protégé plug-in [16]. There are two classes of usage of this KB. 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 KB. For both cases, we needed to provide ways by which users and system modules could access knowledge, update it, and deal with KB changes.

We support access to the distributed knowledge in the system through a query manager to which users and system modules can pose their queries using the KB vocabulary [18]. The query manager then decomposes the query into pieces that can be answered by the individual modules, queries them, and produces the final answer. Since distributed querying can be slow in general, for queries where the response by distributed querying was an issue, we cached the information locally in an information warehouse to avoid the network latencies. This turned out to be a useful service because it makes the information access within the whole system transparent to the users.

While it made good sense to provide transparent access for querying the knowledge, a similar model for updating the knowledge was not feasible because we cannot express all updates using the vocabulary from the KB. For example, to update a SPARK procedure, one needs to express the control structure of a process which requires the use of the full power of the representation language in SPARK. Therefore, for updating the knowledge in the system, we provide custom update modules.

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

During development, there are frequent changes in the vocabulary of the KB. 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 vocabulary as input, computes the differences, and updates the instances in the system as the vocabulary changes, migrating the KB forward through multiple versions.

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 KB at runtime. To support this, we implemented a KB 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 [19].

## **6 Lessons Learned**

We discuss some lessons learned in developing the CALO KB that can be of broader interest to others developing KBs for personal assistants, including semantic desktops.

### **6.1 Use of Upper Ontology**

In the CALO system, the vocabulary in the KB served multiple purposes: a framework for multiple modules to exchange semantic information, a schema in which to hold the data, some of which is learned by the system and some entered directly by the user, and a way to perform deductive inference.

A KB deductive question answering is usually designed with generality in mind, and so is grounded in an upper ontology. For the KB reported here, we used the CLib as the upper ontology. For semantic exchange of information, and for storing the user data, the upper ontology is not of much direct use. Furthermore, the reasoning modules in the current system are highly specialized. For example, the constraint reasoner for scheduling meetings primarily relies on the meeting constraints for inference and does not make use of the taxonomic structure in the CALO KB. Because of these reasons, the upper ontology part of the CALO KB was largely unutilized.

A general lesson we can draw from this experience is that one should carefully assess the advantage of linking to an upper ontology in the context of the reasoning requirements of a project, and not always assume that it will add significant value.

### **6.2 Common Representations**

While for semantic exchange and storing the ground facts in the system an OWL representation is adequate, some modules in the system use much richer representations. For example, the task execution engine uses an expressive process description language

that is not suitable for information exchange across modules. We designed a representation language called the Portable Process Language [14] that was limited to using triples, could be captured in OWL, and captured just enough information that was necessary for the task execution module to communicate relevant information to the rest of the system. In an analogous manner, a learning module in the system associates weights with the logical rules learned by various learners. It was not necessary for the CALO KB in the system or for the purpose of the semantic exchange to provide a representation of weighted logical rules.

A general lesson we can draw from this experience is that it is best not to try to devise with one language that can represent every feature of interest, but instead to look for shareable representations that can serve as the basis of semantic exchange within the system. Highly expressive languages can exist in the system, but they can be translated into weaker representations for communication and information exchange.

### **6.3 Design of Tools, Processes, and Languages**

We used a mixture of tools and processes during the project. For example, we did the bulk of the KB development work within KM and implemented a KM-to-OWL translator for making the vocabulary of the KB available to the project. Some team members contributed to KB directly in OWL by using the Protégé editor. While the use of Protégé enabled multiple contributors to add to the KB, it introduced a greater heterogeneity in the knowledge engineering style, and drift in the terminology. For example, in a section of the KB contributed by a distributed developer, there is a class called Create, and two classes with the same name exist elsewhere in the system with no apparent relationship between the two. We started the project assuming that we would be able to use OWL-DL to meet all the requirements, but as we went along, we realized that we needed to use some but not most features of OWL full. (Recall that OWL has three sublanguages: OWL-Lite, OWL-DL, and OWL full [13]).

The framework should also allow for the use of a heterogeneous set of tools and languages. The development process should provide for maximal participation from the team to contribute to and evolve the KB.

### **6.4 Scenarios for the Semantic Desktop Users**

The KB development effort was driven by the knowledge engineering needs of the individual elements of the project in a fairly bottom-up manner. For example, for one part of the project, we analyzed various semantic desktop applications to see which symbols they needed and designed the KB to support them. For another part of the project, we analyzed what was needed to support meeting interactions, and designed the KB to support that. As a result, different functions in the system used the vocabulary from the KB, but it was hard for each individual module to see the benefit of doing so. While the

approach we used was very pragmatic to stay focused and to be responsive to the needs, there was a missed opportunity on a higher level of semantic cohesiveness that is possible by the linking of concepts across different system modules.

A general lesson that we can draw from this experience is that it is vital to have system-wide semantic use cases that require bringing different functionalities into a cohesive whole. Such a framework leads to an enhanced user experience and leads to semantic-level integration within the system.

### **6.5 Combining with Informal Representations**

Certain aspects of a semantic desktop call for informality. For example, the user wants the ability to name an email folder or desktop folders the way he chooses, but to still have the benefit of ontological searches over that information. Users want to be able to use keywords or semantic tags on various information objects on their desktops. The representation that is designed to support the storage in the RDF-triple store is usually too low level, and too complex for a user to use for this purpose [20].

A general lesson that we can draw from this experience is that we need to provide a way to combine the formal representation in the KB with the less formal and more direct representations.

### **6.6 KB Evolution**

The SOUP was highly effective in maintaining the system as the system evolved. There were several significant changes in the representation design as we went along. For example, in our initial design, we did not support inference using the *subProperty* relation. Later when we started to support the *subProperty* relation, we needed to switch the existing representation without *subProperty* relation to the one with *subProperty* relation. The availability of SOUP to support this kind of migration addressed the resistance of the team to making changes in the vocabulary of the knowledge base, because otherwise, with ontology changes, some of the existing data would have become invalid.

A general lesson that we can draw from this experience is that it is worth taking the time to ensure that adequate tools are in place to allow for a graceful evolution of the KB.

## **7 Open Research Challenges**

The development work on the knowledge base is not yet complete. 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 KB.

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 KB. By making use of such links one can perform inferences using the KB 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 a representation of learning methods, and a way of reasoning with it.

## **8 Comparison to Related Work**

There have been several other projects on building cognitive assistants [21,22] but none of the prior efforts have had a scope as broad as that of CALO. For example, while the Electric Elves agents had the ability to work in an office environment, they were limited in their ability to execute tasks or personalize or improve their behavior through learning [21]. In a similar vein, several systems have been created to investigate the semantic desktop concept, including the Haystack system at MIT [23] and the Gnowsis system at DFKI [24]. These systems support functionality similar to the Organize and Manage Information capability of CALO, but do not address any of the other functionalities, for example, observe and mediate interactions.

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 knowledge bases 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.

CALO's KB is highly adaptable. As a specific example, the CLib generic communication model is a highly reusable chunk of CLib. We showed in this paper how we were able to adapt it to apply to a multimodal representation of meetings.

CALO's KB works across a large number of languages and platforms. 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, this design maximizes 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 KB. 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. The CALO KB embraces and integrates a broad range of heterogeneous elements.

The primary focus of our effort has not been to research and develop a new methodology for knowledge base construction, but we can still draw comparison to approaches others have used. The general structure of the development process we used involved requirement gathering, knowledge reuse, extension, implementation, evaluation, and refinement. This structure is very similar to the one advocated in the On-To-Knowledge methodology [25]. Even though we did not use a formal requirement specification language as advocated in the Methontology approach [26], we allowed the consumers to state the requirements either in plain English, or as lists of classes and relations that needed to be represented. An aspect of our requirements analysis process was governed by the competency questions that we expected the system to answer. This step is similar to the approach advocated in the TOVE project [27]. Finally, our collaborative processes were largely informal, and we did not use a formal structure of the sort advocated by the Diligent approach [28].

## **9 Summary and Conclusions**

We presented a case study in engineering a knowledge base to meet the requirements of a cognitive agent that organizes information and performs tasks. This effort generated a wide set of requirements for the system's vocabulary, whose satisfaction was complicated by a large distributed team using many different platforms. We provided a description of our development process, and discussed specific knowledge engineering challenges in reusing an existing ontology, in modeling multimodal meetings and tasks, and in representing 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 a contribution to the state of practice in engineering knowledge base systems for semantic desktops and cognitive assistants, 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 Contract No. 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 DARPA or the Department of Interior-National Business Center (DOI-NBC). We thank Chris Brigham, Sunil Mishra, and Guizhen Yang for their implementation effort on the project and Jack Park for his useful comments on the paper.

## References

1. Barker, K., B. Porter, and P. Clark, A Library of Generic Concepts for Composing Knowledge Bases, in Proceedings 1st Int Conf on Knowledge Capture (K-Cap'01). 2001. p. 14–21.
2. Cheyer, A., J. Park, and R. Guili. IRIS: Integrate, Relate, Infer, Share, in Semantic Desktop Workshop. 2005. Galaway.
3. Myers, K., et al., An Intelligent Personal Assistant for Task and Time Management. AAI Magazine, 2006.
4. Niekrasz, J., M. Purver, and J. Dowding. Ontology-based Discourse Understanding for a Persistent Meeting Assistant, in The AAI Spring Symposium on Persistent Assistants: Living and Working with AI. 2005. Stanford, CA.
5. Brigham, C., T. Dietterich, and T. Uribe, Combining Learners, Data and Probabilistic Inference. 2006. SRI International.
6. Fikes, R.E. and N.J. Nilsson, STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. Artificial Intelligence. 1971. 2: p. 189–208.
7. Clark, P. and B. Porter, KM - Situations, Simulations, and Possible Worlds. 1999.
8. Dawson, F. and D. Stenerson. Internet Calendaring and Scheduling Core Object Specification. 1998 [cited; Available from: <http://tools.ietf.org/html/2445>].
9. Hobbs, J. and F. Pan, An Ontology of Time for the Semantic Web. ACM Transactions on Asian Language Information Processing. 2004. 3:1.
10. Clark, P. and B. Porter. KM - The Knowledge Machine: Users Manual. 1999 [cited; Available from: The system code and documentation are available at <http://www.cs.utexas.edu/users/mfkb/km.html>].
11. Morley, D. and K. Myers. The SPARK Agent Framework. in International Conference on Autonomous Agents and Multi-agent Systems. 2004.
12. Erol, K., J. Hendler, and D. Nau, Semantics for Hierarchical Task-Network Planning. 1994, University of Maryland at College Park.
13. McGuinness, D.L. and F. van Harmelen, OWL Web Ontology Language. 2004.
14. Clark, P.E., et al. A Portable Process Language. in Workshop on the Role of Ontologies in Planning and Scheduling. 2005. Monterey, CA.
15. Chaudhri, V.K., B. Jarrold, and J. Pacheco. Exporting Knowledge Bases into OWL, in OWL: Experiences and Directions. 2006. Athens, Georgia.
16. Gennari, J., et al., The Evolution of Protege: An Environment for Knowledge-Based Systems Development. International Journal of Human-Computer Interaction, 2003. 58(1): p. 89–123.

17. Niekrasz, J. and M. Purver. A Multimodal Discourse Ontology for Meeting Understanding, in *Machine Learning for Multimodal Interaction: Second International Workshop, MLMI 2005*. 2005. Edinburgh, UK: Springer Verlag.
18. Ambite, J.L., et al. Integration of Heterogeneous Knowledge Sources in the CALO Query Manager, in *Proceedings of the Innovative Applications of Artificial Intelligence Conference*. 2006.
19. Chaudhri, V.K., et al., OKBC: A Programmatic Foundation for Knowledge Base Interoperability, in *Proceedings of the AAAI-98*. 1998: Madison, WI.
20. Park, J., and Cheyer, A., "Just for Me: Topic Maps and Ontologies", in Lutz Maicher and Jack Park (Editors), *Charting the Topic Maps Research and Applications Landscape: First International Workshop on Topic Map Research and Applications, TMRA 2005, Leipzig, Germany, 6-7 October 2005, Revised Selected Papers, Springer LNCS Volume 3873/2006* pp. 145-159.
21. Chalupsky, H., et al., Electric Elves: Applying Agent Technology to Support Human Organizations. *AI Magazine*, 2002. **23**(2).
22. Pollack, M., Intelligent Technology for an Aging Population: The Use of AI to Assist Elders with Cognitive Impairment. *AI Magazine*, 2005. **26**(2): p. 9–24.
23. Quan, D., D. Huynh, and D.R. Karger. Haystack: A Platform for Authoring End User Semantic Web Applications, in *International Semantic Web Conference*. 2003.
24. Sauermann, L., et al. Semantic Desktop 2.0: The Gnowsis Experience. in *International Semantic Web Conference*. 2006.
25. Sure, Y., S. Staab, and R. Studer, On-To-Knowledge Methodology, in *Handbook On Ontologies*, S. Staab and R. Studer, Editors. 2004, Springer Verlag.
26. Lopez, M.F., et al., Building a Chemical Ontology Using Methontology and the Ontology Design Environment. *IEEE Intelligent Systems*, 1999.
27. Fox, M.S. and M. Gruninger, Ontologies for Enterprise Integration, in *Proceedings 2nd Conference on Cooperative Information Systems*. 1994. p. 82–90.
28. Vrandečić, D., et al., The Diligent Knowledge Processes. *Journal of Knowledge Management*, 2005. **9**(5): p. 85–96.