Lessons Learned in Building a Knowledge Base for a Cognitive Assistant
Vinay K. Chaudhri, Adam Cheyer, Bill Jarrold, Karen L. Myers
Artificial Intelligence Center
SRI International, Menlo Park, CA, USA

Abstract
We present our experience in developing a knowledge base (KB) in the context of an intelligent personal assistant called CALO. The goal of building the intelligent assistant was to aid a knowledge worker in managing information on their desktop, enable the worker in managing time commitments and performing tasks, and assisting during meetings. While we leveraged state-of-the-art best practices in constructing a KB for CALO, we faced unique challenges as the system evolved from its initial conception to deployment and raised the following questions: How well do the traditional KB development methodologies and upper ontologies work in a context in which the goal is a deployed system rather than ontology research? What ontology support is needed for an information management application that leverages statistical machine learning? How can a KB support cognitive task assistance? The primary contribution of our work is in giving a detailed account of successes and failures in addressing these questions and the general lessons that can be drawn for future research and development of knowledge base systems.

Introduction
The technological goal of the CALO project is to develop an integrated cognitive system with the ability to acquire knowledge about the world through experience (to learn) and to use that knowledge to solve problems and deal with unanticipated circumstances (to reason) [1]. The CALO system is aimed at a knowledge worker who routinely performs the following functions: working with electronic mail, performing tasks, planning meetings, attending meetings, etc. Assisting such a knowledge worker is a compelling application for creating a new class of robust, adaptable and collaborative software systems which integrate traditionally disjoint user data such as email, contacts, calendars, documents and projects.

Several papers give an overview of the implemented capabilities of CALO [2, 3]. The present paper is about the knowledge base (KB) and the ontology used in CALO. A primary goal of the KB development effort was to provide a semantic foundation for the integration of multiple technologies that contributed to the CALO system. To meet that challenge, we first turned to off-the-shelf tools and practices. As these efforts were met with varying degrees of success we developed customized solutions where necessary. An important contribution of the present paper is an analysis of what worked and what did not.

Another important goal of KB development was to support the needs of learning research. This is a research area that is largely unexplored. The need to support learning research raised the following questions: what kind of a KB or ontology is appropriate for statistical machine learning? How can a KB enable learning of tasks? While we cannot
claim to have definitive answers to these questions, we do believe that the experience we report provides important data points towards answering these questions.

We begin our discussion by giving an overview of the capabilities of the CALO system. We then describe the KB content development, and the associated tools and services that we implemented to use the KB. We conclude with an evaluation of our approach and the general lessons that can be drawn from it for future research and development efforts.

**Overview of CALO Capabilities**

We identify six broad classes of functions that CALO performs to provide the background context for what a KB must represent. The project team defined these functions as a concrete realization of a near-term capability that could be realized within the five-year time frame of the project.

1. **Organize and Manage Information:** Organize the information on a user’s desktop by learning 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, organizing user’s emails and files, etc.

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.

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.

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.

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. Supporting this function requires a representation of the schedule and scheduling constraints as well as a model of preferences that a user may have over individual scheduling requirements.

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.
KB Development

It is helpful to think of the KB content development process over the five-year duration of the CALO project in terms of three phases: Initial Design, Simplification and Refinement, Deployment for End Users. Our approach for KB development in all the three phases of the project involved: requirements gathering, knowledge reuse, knowledge extension, and then implementation. This structure is very similar to the one advocated in the On-To-Knowledge methodology [4]. We will now explain in detail the challenges faced in each phase of the development.

Initial Design Requirements

In ambitious research projects such as CALO, it is hard to have a precise set of requirements from the very beginning. But, it was obvious that the KB would need representations for people, projects, emails, documents, tasks, meetings and other office related objects and activities. The KB also needed to support the various types of learning as well provide sufficient reasoning to support the cognitive functions of CALO. The KB should also be developer friendly so that the ontology can be easily incorporated into the system code base. Throughout the 5-year project, the latest version of the CALO system was subject to a yearly evaluation which measured its ability to show an enhanced ability to answer questions of interest to users. Thus, an additional requirement is that the KB support such query answering.

Knowledge Representation Framework

It is an accepted best practice to start an ontology development effort from an upper ontology. There are several upper ontologies available, for example, Cyc [5], DOLCE [6], SUMO [7], etc. We chose the Component Library (CLib) as the primary upper ontology for the following reasons [8]: First, CLib provides a small set of carefully chosen representations that are domain independent, reusable, and composable. We believed that these three properties were important to guaranteeing a flexibly extensible ontology which was important to a cognitive architecture that integrated a broad array of capabilities. Second, CLib uses a STRIPS model for representing actions that is close to what was needed for supporting the function of monitoring and managing tasks [9, 10], and (3) CLib provides a well-thought-out model of communication that seemed 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 [11] standard for representing the calendar and to-do information. We leveraged the DAML-Time ontology as an inspiration for representing time [12]. 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 [13]. The choice of the KM language followed from the choice of using the CLib because CLib is native to KM.
It is worthwhile at this point to illustrate a sample class definition from KM to illustrate the flavor of this language. We see below a KM representation of the class *Express*. This class was important to the above mentioned communication model. It is a generic concept fundamental to all kinds of communication. There are some OWL expressible assertions such as “Express is a subclass of Action” and “Any value on the recipient slot of an instance of express must be an instance-of Tangible-Entity”. However, there are a number of assertions that require writing rules. For example, “For every instance of Express, there exists a Tangible-Entity that is its agent, an instance of Information that is its object, and a Message which is its result.” In addition we see the use of the STRIPs style pre-conditions list with an assertion that says “Every instance of Express requires as a precondition that the agent must know the information being expressed and the agent must know the language in which the information is expressed.”

```xml
<Express has (superclasses (Action)))

<every Express has
  (recipient ((must-be-a Tangible-Entity)))
  (agent ((a Tangible-Entity)))
  (object ((a Information)))
  (result ((a Message with (result-of (Self)))))

  ;; Preconditions: The agent must know the object
  ;; the agent must be experiencer of a Be-Knows with object the information
  [pcs-list ((:triple (:the object of Self)
                 :object-of
                 (:a Be-Known with
                  :experiencer (:the agent of Self))))

][Express pcs-1])

(:triple (:the agent of Self)
  :experiencer-of
  (:a Be-Known with
   :object (:the Language information-language of
            (:the result of Self)))
  :experiencer (:the agent of Self))

)
```

We represent knowledge for performing automated tasks in the SPARK procedure language [14], which is similar to the hierarchical task network (HTN) representations used in many practical AI planning systems [15]. 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. At its simplest, a SPARK *procedure* has the form

```
{defprocedure name cue : trigger precondition: P body: N}
```
This indicates that if $P$ is true when $\text{trigger}$ occurs, then executing $N$ is a valid way of responding. The cue may be of the form $(\text{newfact: } P)$ to respond to the fact $P$ being added to the KB, or $(\text{do: } A)$ to expand the action $A$.

For example, the following procedure, $\text{Get\_Bid}$, describes one way of expanding the task of performing action $\text{find\_bids}$. It is applicable if the condition $(\text{Online "DM4QR"})$ holds. It performs a $\text{laptop\_query}$ action that binds variable $\$temp\_quotes$ and then depending upon whether or not this list is empty, either performs $\text{relax\_and\_redo\_query}$, binding $\$quotes$, or binds $\$quotes$ to $\$temp\_quotes$.

\begin{verbatim}
{defprocedure Get_Bid
cue:
  [do: (find_bids $item $criteria $quotes)]
precondition: (Online "DM4QR")
body:
  [seq:
    [do: (laptop_query $criteria $temp_quotes)]
    [select:
      (= $temp_quotes [])
      [do: (relax_and_redo_query $criteria $quotes)]
      (True)
      [set: $quotes $temp_quotes]]]
\end{verbatim}

We represent the uncertain knowledge extracted by the learning algorithms using weighted first-order logic rules that are processed using a Max-SAT solver [16]. The weights are assigned by a learning module, and the rules use vocabulary drawn from the CALO KB. We show an example of such a rule below that states that if a document is attached to an email message, both the document and the message tend to be associated with the same project. The Markov logic weight is shown in square brackets.

\begin{verbatim}
(forall E, P, D
  (=> (and
    (instance-of E Email)
    (instance-of P Project)
    (instance-of D Document)
    (attached D E))
    (and (projecOf E P)
    (projectOf D P))) [2.20]
\end{verbatim}

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 a
Resource Description Framework (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 [17] 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 the 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 [18] 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. Details of the translation of the KB into OWL are also available in a separate paper [19].

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.

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. Therefore, we did not use a formal requirement specification language as advocated in the Methontology approach [20], we allowed the consumers to state the requirements either in plain English, or as lists of classes and relations that needed to be represented.

In this phase, we strove for large-scale reuse of existing knowledge representations. For example, we imported the 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.

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 [21].
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.

**Knowledge Base Implementation**

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] and customized it to allow one to see the all of the transitive closure of slots that may apply to a given class and its superclasses.

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.

While it made good sense to provide transparent access for querying the knowledge through a query manager, 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-subscribe 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.

Ontology changes not only affect instance data stored in the KB, they also affect the software modules. While SOUP allowed propagation of changes in the ontology to the KB, we also needed a means to keep software modules (specifically our JAVA code base supporting IRIS) in sync with the ontology. We implemented ontology instances as POJOs (Plain Old Java Objects) that could be automatically generated from the ontology. Since ontologies need to eventually ground out in real code, and since they are evolving
frequently, automatically generating Java representations from our ontologies allowed static (compile-time) type checking. When a class is renamed or dropped (e.g. during the ontology simplification and refinement effort mentioned below), a re-compile will find all occurrence in our huge codebase that used the class. If a developer wanted to use an ontology class (from the Java side), we had a Protoge-based documentation ontology that needed to be completed first, and both OWLDOC and POJOs were generated directly from that. This guaranteed that the documentation and the code were always in synch, something that is very important on such a large project with varying levels of documentation hygiene.

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].

Knowledge Base Content
In this section we give an overview of the content in the knowledge base, and illustrate how these methodologies were applied in creating this content.

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.

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.

Representing Meetings

We represent meetings in the CAKO 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 [22].

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.

**Experience and Feedback on Initial Design**

We undertook an assessment of the experience of the initial design and implementation of the KB describe above after approximately two years into the project. By that time, there was sufficient experience in trying out different techniques, and there was a greater sense of project direction and the following conclusions were apparent.
1. Much of the ontology was not being used by anyone in the project. This was primarily because the needs of the learning researchers were for knowledge were limited to only a small number of classes and relations.
2. The detail of representation in the ontology was way too fine-grained for developers as well as for learning researchers. For example, to query the knowledge base for a person’s email address required the traversal of an unacceptably large number of links.
3. There was a lack of agreement on the meaning of key terms, for example, Task, Project, etc, that were used very frequently in the project.
4. The development process did not scale due to the use of a centralized knowledge engineer

Based on these experiences, we undertook an ontology simplification and refinement effort that we describe next.

**KB Simplification & Refinement**

The KB simplification and refinement involved the following steps

1. Pruning the ontology based on the extensive analysis of the code
2. Decentralization of the ontology development
3. Redesign key elements of the ontology
   a. Agree on the meaning of a few key terms and a core ontology of tasks
   b. Agree on a small set of relations that are central for relational learning
   c. Define an ontology of tasks
   d. Support query answering over executable procedures

Pruning the ontology based on the extensive analysis of the system code was a straightforward process. The domain-specific classes that were not being used by anyone on the project were simply dropped from the ontology. The table below shows the sizes of the CALO Specific Ontology (CSO) before and after simplification compared with the size of CLib, i.e. the core ontology which the CSO extended.

<table>
<thead>
<tr>
<th>Time</th>
<th>Classes</th>
<th>Slots</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLib only</td>
<td>571</td>
<td>626</td>
</tr>
<tr>
<td>CSO pre-simplification</td>
<td>714</td>
<td>1345</td>
</tr>
<tr>
<td>CSO post-simplification</td>
<td>156</td>
<td>349</td>
</tr>
</tbody>
</table>

An additional goal of ontology pruning was to produce representations that would reduce the number of individuals in the KB as a result of reification. Consider, for example, the following English assertion:

*Fred wrote the email message.*

Before simplification, that assertion was represented as the following:

```xml
(plays Fred _AuthorRole43)
(agent _AuthorRole43 _Writing-Event41)
(results _Writing-Event41 _EmailMessage7)
```
Here, _AuthorRole43 is a role entity that relates Fred to a writing event, _Writing-Event41, that results in an email message _EmailMessage7.

After simplification, the same knowledge was represented as:

Fred - authorOf → EmailMessageIns7

In the simplified representation, there is no straightforward way to associate the specific time duration during which Fred was an author of a specific email especially using OWL, but the simplified assertion does not require reifying _AuthorRole43 and _Writing-Event41. The reasoning performed by CALO did not leverage such additional detail in the representation.

We achieved decentralization in ontology development by carving out naturally disjoint subsets of ontology that could be managed by a specific individual with little need of coordination. We invested bulk of the design effort in agreeing on the meaning of a few key terms and designing a core ontology of tasks that we describe next.

**Agreeing on the Meaning of Key Terms**

The goal of an ontology is to arrive at the shared conceptualization of a domain. While the initial version of the CALO ontology had about 1000 classes and 500 relations, which were defined based on their usage in standard vocabularies such as iCal, there was lot of confusion on the meaning of a few basic terms. For example, a simple term such as Task was being used by different researchers to mean different things. The user feedback on the ontology also indicated that instead of aiming to get a consensus definition for all the terms in the ontology we should focus on just a small number of terms that almost everyone needs to refer to in the project: Document, Email Message, Contact Entry, Task, Project, Procedure, Action Item, and a To Do item. These terms could also be viewed as the upper ontology of the application domain of task assistance in an office setting.

To specify the definitions of these terms, we developed the following guidelines that were based on the experience of [23, 24].

1. The terms should be defined in the context of the office domain.
2. Reference already available resources for guidance before writing a definition. The resources we consulted include Wordnet, Open Cyc, CLib, Oxford English Dictionary.
3. All the definitions should take the following form: an A =def. a B which C's, where B is a superclass of A, and C is a property that distinguishes A and C. It is not necessary that the distinguishing properties must be necessarily represented in the ontology.
4. Each definition must be accompanied by positive and negative example individuals drawn from the application domain.
5. The definition must be reviewed and agreed upon by the key stake holders.
6. Add necessary clarifications, but keep such clarifications distinct from the definition of a term.
We will now present the definitions of the key terms that were agreed across the project. For each term, we present its definition, positive and negative examples, and then discuss some of the issues that arose in settling on that definition.

A prerequisite for defining the term *Task* was the notion of an *Event*. As per the principle (2) stated above, we simply reused the definition from CLIB, and added the project specific positive and negative examples.

\[
\begin{align*}
\text{**Event:**} & \quad \text{Things that happen as opposed to Things that are.} \\
\text{**Positive example instances:**} & \quad \text{a meeting between Fred and Sue; sending Email43 to Jane} \\
\text{**Negative example instances:**} & \quad \text{Vijay’s laptop, a document}
\end{align*}
\]

During the review of this definition, the users asked for clarifications based on which we needed to add the following information to this definition. Events can have temporal properties such as begin time, end time, duration, and can also have subsubevents. An event can be actual (already happened or happening) or presumptive (future or theoretical). An event may have zero length duration.

\[
\begin{align*}
\text{**Task:**} & \quad \text{An Event that one or more agents intend to do (or execute) to achieve a specific or a set of interrelated objectives.} \\
\text{**Positive example instances:**} & \quad \text{Schedule a meeting with John in EJ211, on Monday, Jan 10th, at 5PM; my communication with DARPA} \\
\text{**Negative Example instances:**} & \quad \text{A powerpoint document}
\end{align*}
\]

Thus, the key distinguishing characteristic of a *Task* as compared to an *Event* is that there is an agent associated with it, who is trying to achieve one or more objectives. While one can imagine tasks that may not have agents or objectives, the principle 1 mentioned earlier provides the context for the definition: for the purposes of this project, we are only interested in those tasks that have an agent and which have one or more objectives. As an example, a module trying to learn the task being performed by a user may learn patterns that were not intended by the user. An example of such a pattern is that “a user A immediately responds to an email from another user B.” This pattern may suggest that the user A intends to perform the *Task* of responding to the user B, but it is incorrect to label it as a task unless the user’s intent is confirmed by a suitable method, for example, by asking the user.

Since there was a lot of confusion in the team around the distinction between the referents of the English terms, *procedure, task, task Instance, activity*, we added the following clarifications to the definition of a task.

A specific instance of the class Task may have *sub-tasks*. For example, an instance of *Buy* has *sub-tasks* such as obtaining quotes, checking budget, etc. The description of a Task may specify which sub-tasks an instance of a Task may have. For example, the description of *Buy* may assert that every instance of *Buy* has sub-tasks such
as obtaining quotes, checking budget etc., but that description by itself does not imply that the class \textit{Buy} has those sub tasks. Other properties of the instances of a task are state of execution (e.g., active, suspended, etc.), priority, the person responsible for it, etc.

After significant discussions, we concluded that the sense in which the word \textit{activity} was being used in the project allowed by the definition of a \textit{Task} as defined here. Therefore, the English term “activity” should be interpreted to mean the same thing as the \textit{Task}.

| Procedure: | A procedure is a piece of information that specifies a method for achieving a designated objective, defined in terms of tasks to be performed. |
| Positive example instances: | an implementation of a task written in the process modeling language SPARK |
| Negative example instances: | Attending a meeting; the \textit{event} of executing a machine learning clustering algorithm on an email folder; the number 42. |

The definition of a \textit{Procedure} uses the term \textit{Information} that has not been defined so far. We used the definition from the CLIB, and also included terms such as \textit{Entity}.

Based on the discussion with the team, we added the following clarifications to the definition of a \textit{Procedure}. A procedure can have many other components to it (for example, gating conditions on its application). A task learning module may learn only a new task, or a procedure to execute a known task. A task recognition module observes the procedures to recognize which task might be executed, but does not recognize all the details necessary to execute that task.

| Project: | A project is an event in which an individual or an organization seeks to meet specific objectives via a set of coordinated tasks, within defined time, cost and performance parameters. |
| Positive example instances: | Furnishing my house, The CALO project |
| Negative example instances: | a procedure; the \textit{set} of emails associated with ProjectXYZ, the \textit{collection} of people working on “The Calo Project”; The SRI homepage; |

This definition of project was adapted from a definition used by the Association of Project Management\textsuperscript{1}, and is based on an IEEE standard. Detailed background and rationale for this definition is available elsewhere\textsuperscript{2}. The original definition required the projects to have definite starting and ending point, but for the purposes of its use within CALO, we dropped that requirement from the definition.

A colloquial usage of the term \textit{Project} is used to refer to things such as entities and events associated with it also. Under a strict use of the term \textit{Project} as defined here, such a usage is incorrect, as the entities and event associated with a specific projects are participants or sub-tasks in it, and not the \textit{Project} itself.

\textsuperscript{1} http://www.apm.org.uk/PtoQ.asp
\textsuperscript{2} http://egweb.mines.edu/eggn491/Information\%20and\%20Resources/pribok.pdf
An example of the confusing use of this term in the early phase of the project is as follows. In the organize information application of CALO, a machine learning algorithm was run to cluster the emails of a user. The emails in a cluster suggest that they might be affiliated to the same project. Some developers referred to those clusters of emails as *Projects*. The email cluster by itself is merely a cluster of information and is a negative example of an instance of a Project.

**Action Item**: An action item is a task that involves a joint commitment between at least two people: a Supervisor and one or more Owners.

**Positive example instances**: Tasks arising during a discussion between people.

**Negative example instances**: I decide to call my wife

The distinguishing aspect of an action item and a task is that an action item involves a joint commitment between at least two people. The Owner of an Action Item is the person responsible for taking actions to achieve the Action Item. The Supervisor is the person the Owner must report success/failure to. An Action Item is agreed upon by a group. For example, in the CALO ontology meeting, the group decided to write a summary document defining the key terms used in CALO. In this case, the task of writing a summary document is an action item for which the agreeing agents are the participants of the ontology meeting.

**To do**: A To Do is a task or an action item that is on some agents’s Task list. An optional property of a To Do is a deadline.

**Positive example instances**: The following entries on my To Do list: Go to the Gym; Write a proposal;

**Negative example instances**: The string “Dentist” which is a label of an entry in a task manager, Morning, 50

Whenever a To Do item is stated as an informal text string, it is viewed as a label of a task or action item that is represented in the knowledge base. Thus, while informal strings such as “Dentist” may denote tasks, but they do not become part of the CALO KB until they correspond to a reified To Do item.

**Vocabulary of Tasks**

Since the goal of the CALO system is to assist its users by performing tasks, it needed to have a vocabulary for defining a variety of tasks that it could potentially do for its users, for example, sending an email, or arranging a meeting, etc. The Task Ontology provides basic vocabulary for tasks and conceptual definitions for tasks and their properties. In addition, its goal is to provide knowledge to answer queries about tasks such as: Can CALO perform a specific task X? Where should a new learned task fit in an already existing taxonomy of tasks? The design of the ontology of tasks proceeded in the following steps:

1. Define extensions to the key term definitions to define procedures
2. Define relations to be associated with tasks with procedures
3. Define a taxonomy of tasks

We will describe each of these steps in greater detail now.

**Extending Key Term Definitions to define Procedures**

We extended the key terms considered earlier to include several task specific terms that were necessary to relate a Task with a Procedure used to implement it. These terms include *AbstractTask, ConcreteTask, Parameter, Input Parameter, OutputParameter, Required Parameter* and *Optional Parameter*. We used the same methodology to define these terms as we did for the key terms.

**Abstract Task**: An abstract task is a Task that has no procedure description associated with it.
**Positive example instances**: The general task of Buying or Moving
**Negative example instances**: The task of buying a computer for which there exists an implemented procedure in CALO to execute it.

**Concrete Task**: A concrete task is a Task that has a procedure description associated with it. The procedure description is sufficient for SPARK to execute the concrete task.
**Positive example instances**: The task of arranging an interview visit for which there exists a procedure in CALO.
**Negative example instances**: The general task of Buy or Move

A concrete task has a procedure interface associated with it which could be implemented by one or more procedures. For simplicity, we decided not to explicitly model a procedure interface in our ontology. Instead, we directly associate with a task the parameters that are associated with a procedure interface.

**Parameter**: A Parameter is an input or output argument/variable that defines a procedure that implements a concrete task.
**Positive example instances**: The *model number parameter* that is necessary to specify when executing a Buy task
**Negative example instances**: 24601 (the specific model number of a particular item to be purchased); the slot *emailCCIs* (which specifies the cc list for a SendEmail task) $49.95 (the actual price of a particular item that has been tasked to purchase); an actual computer file

For example, the implementation of a concrete task called *BuyAComputer* would take *model number*, and *purchase price* as input parameters. The parameters are properties of a task class. They are not shared among tasks. An abstract task need not have any parameters, but may specify them if desired. An executable task must specify a full parameter signature.

Next, we define several subclasses of the class *Parameter*. 
**Input Parameter:** A parameter that acts as an input to a procedure.
**Positive example instances:** filename parameter for a CreateFile task
**Negative example instances:** “Conference Agenda.doc” -- the actual filename given as input to a CreateFile task; computer file parameter in a CreateFile task (computer file parameter is an output parameter);

**Output Parameter:** A parameter that acts as an output to a procedure.
**Positive example instances:** quoted price for a Get Quotes task; computer file parameter in a CreateFile task.
**Negative example instances:** $49.95 the actual value produced by a Get Quotes task; filename parameter for a DeleteFile task (filename parameter is an input parameter),

**Required Parameter:** A parameter that must be bound to an actual value either before or after its Task is executed.
**Positive example instances:** The recipient parameter of a SendMail Task, The mail folder parameter in a Create Mail Folder task.
**Negative example instances:** The attachment parameter of a SendMail Task (attachments are not required); billg@microsoft.com (actual values are not formal parameters)

**Optional Parameter:** A parameter that is not required to be bound to an actual value when its Task is invoked.
**Positive example instances:** The attachment of a Send Mail Task; The bcc parameter of a Send Email Task
**Negative example instances:** The recipient of a Send Mail Task

The positive and negative examples of different kinds of parameters given above are specific to the choices made in the CALO ontology, and these examples may not apply more widely. For example, while the **Quoted Price** is a negative example for an input parameter for *Get Quotes* task, but it might be an input parameter for some other procedure. Similarly, the recipient of some *Send Mail* task in another application may be an optional parameter.

The parameters play different roles in a procedure.

**Parameter Role:** Describes the function or purpose that a parameter of a procedure implementing a task can play in that task. A role does not specify any implementation details about a parameter.
**Positive example instances:** Purchaser, Authorizer
**Negative example instances:** Display Name, Execution State
Taxonomy of Tasks

The CALO system needs to deal with the tasks that a user may perform on his desktop, for example, sending email, moving folders, or the tasks that arise in an office setting, for example, purchasing equipment. We collected an inventory of such tasks and organized them in a task taxonomy. Our approach for doing so was as follows:

1. Identify a set of task classes that are general purpose, reusable, and domain-independent, for example, Add, Delete, Create, Send, etc.
2. Create domain-specific classes by combining each Task class with an entity that is used in that task class, for example, Send Email, Create File, DeleteFile, DeleteEmail etc. Send Email is created by combining the Send with the entity Email. While many classes could be created by combining every Task with every Entity, we create only those classes that were explicitly referenced in the application.
3. We added additional organization into the taxonomy by using meta classes such as BrowserTask, ChatTask, FileTask. For example, the meta class Browser Task collects all tasks that a user may perform while using an internet browser, for example, loading a web page, navigating to a previous page, etc.

The task ontology has 128 primitive classes, 4 meta classes, and 239 defined classes.

Relations to associate tasks with procedures

We drove the design of the relations by the reasoning requirements of CALO. This methodology is consistent with advocated in the TOVE project [25]. These requirements were:

1. Given a task and a role, determine which property/parameter has the value for that role.
2. Given a task and a role, determine in which position does the parameter representing that role appear in the procedure implementing that task.
3. Given a task and a role, determine the value of the parameter representing that role.
4. The parameter information should not inherit across the task hierarchy. This is because the procedure implementing the sub task of a task may have a completely different sequence of parameters. Therefore, it is important that each Task class is able to specify parameters that make sense for the procedures that implement it.
5. The role and property information should inherit to subclasses and instances of a task.

We first define classes InputParameter, OutputParameter, RequiredParameter, and OptionalParameter. InputParameter and OutputParameter are disjoint from each other. RequiredParameter and OptionalParameter are disjoint from each other.

\[(\text{disjoint } \text{InputParameter } \text{OutputParameter})\]
\[(\text{disjoint } \text{RequiredParameter } \text{OptionalParameter})\]

To illustrate different relations, we will take the example of the task of buying computers.
We first define \textit{BuyComputer} as a subclass of \textit{Buy}.

\textit{(subclassOf BuyComputer Buy)}

Next, we say that the instances of \textit{BuyComputer} have a property \textit{agentIs} whose values are drawn from the class \textit{TangibleEntity}. (The relation \textit{someValuesFrom} is a standard relation from OWL)

\textit{(someValuesFrom BuyComputer agentIs TangibleEntity)}

This assertion ensures that all subclasses and instances of \textit{BuyComputer} have the property \textit{agentIs}. Next, we define the roles for this task.

\textit{(hasValue BuyComputer taskRoleTypeIs PurchaserRole)}

We associate a role with a task using the relation \textit{taskRoleTypeIs}. We say \textit{(taskRoleTypeIs T R)} if the task could have some parameter that plays the role \textit{R}.

This assertion ensures that all subclasses and instances of \textit{BuyComputer} have a \textit{PurchaserRole}.

We define the relation \textit{parameterIs} to represent the parameters of a Task. We say \textit{(parameterIs T P)} if the task \textit{T} has parameter \textit{P}. An example use of this relation follows:

\textit{(parameterIs BuyComputer Parameter-7)}
\textit{(instance-of Parameter-7 InputParameter)}
\textit{(instance-of Parameter-7 RequiredParameter)}

With each parameter, we can associate the position at which it appears, the role that it plays in the task, and which property of the task contains the value for that parameter as follows:

\textit{(parameterPositionIs Parameter-7 1)}
\textit{(parameterRoleTypeIs Parameter-7 PurchaserRole)}
\textit{(propertyWithValueIs Parameter-7 agentIs)}

We say \textit{(parameterPositionIs P N)} to represent that the parameter \textit{P} appears in the position \textit{N} in the signature of the procedure. In the example above, the \textit{Parameter-7} appears in the first position in the procedure that implements \textit{BuyComputer}.

We say \textit{(parameterRoleTypeIs P R)} to represent that the parameter \textit{P} plays the role \textit{R} in the procedure using it. In the example above, \textit{Parameter-7} plays the \textit{PurchaserRole} in the procedure that implements \textit{BuyComputer}.
We say \((\text{propertyWithValueIs } P \ R)\) to represent that the value of the parameter \(P\) is the same as the value of the property \(R\) of the task. In the example above, the parameter \(P\) has the same value as the value of the property \(\text{agentIs}\) for the \(\text{BuyComputer}\) task.

We associate a parameter with a role using the relation \(\text{parameterRoleTypeIs}\). We say \((\text{parameterRoleTypeIs } P \ R)\) if the parameter \(P\) plays role \(R\). A role can be played by multiple parameters, and each parameter can play many roles. Thus, the relation \(\text{parameterRoleTypeIs}\) is a many to many relation.

We associate a parameter with a task using the relation \(\text{taskRoleTypeIs}\). We say \((\text{taskRoleTypeIs } T \ R)\) if the task could have some parameter that plays the role \(R\).

For a \(\text{Concrete Task}\), each role is associated with one or more parameters. For a concrete task \(T\) with role \(R\), then there must exist one or more parameters \(P\) such that \((\text{parameterRoleTypeIs } P \ R)\) holds true.

For an \(\text{Abstract Task}\), the roles constrain the roles of the Concrete Tasks that are its subclasses. Thus, if a \(\text{Concrete Task} T\) inherits a role \(R\) from one of its super classes, then there must exist a parameter \(P\), such that \((\text{parameterRoleTypeIs } P \ R)\) holds true.

Next, consider an individual concrete task: \(\text{JohnBuysAComputer}\).

\((\text{instance-of } \text{JohnBuysAComputer} \text{ Buy})\)
\((\text{agentIs } \text{JohnBuysAComputer} \text{ John})\)

Therefore, by inheritance we must have:

\((\text{taskRoleTypeIs } \text{JohnBuysAComputer} \text{ PurchaserRole})\)

Now, we will illustrate how this representation meets the requirements outlined earlier.

Given a task \(T\) and a role \(R\), we can determine which parameter has the value for that role using the following query:

\((\text{and})\)
\((\text{(parameterIs } T \ ?p))\)
\((\text{(parameterRoleTypeIs } ?p \ T))\)

Given a task \(T\) and a role \(R\), we can determine which property has the value for that role using the following query:

\((\text{and})\)
\((\text{(parameterIs } T \ ?p))\)
\((\text{(parameterRoleTypeIs } ?p \ R))\)
\((\text{(propertyWithValueIs } ?p \ ?W))\)
Given a task T and a role R, we can determine in which position does the parameter representing that role appear in the procedure implementing that task using the following query: Given a task and a role, determine the value of the parameter representing that role?

Unlike the previous two queries, this query makes sense only for an instance of a task. Therefore, we need to know not only the task T, but also the instance of the task I to complete this query.

\[
\text{(and} \\
\text{(parameterPositionIs T ?p)} \\
\text{(parameterRoleTypeIs ?p R)} \\
\text{(propertyWithValueIs ?p ?P)} \\
\text{(holds ?P I ?v)})
\]

The following query shows that the role information inherits to the instances of a Task.

\[
\text{(taskRoleTypeIs JohnBuysAComputer ?x)}
\]

These sample queries illustrate that the chosen relations meet the reasoning requirements for reasoning with parameters of procedures that implement tasks.

**Support Query Answering Over Executable Procedures**

As explained earlier, the executable representation of a procedure is represented using a process modeling language called SPARK. SPARK includes a comprehensive API for monitoring executing procedures—called procedure instances—allowing external modules to view the "trace" of a procedure's execution. For example, by using this API an external module can find the specific task(s) currently being executed, and trace back to find specific previously executed tasks and their details. While allowing access to how a general procedure is playing out, this API does not allow access to the general "flow chart" of the procedure itself, for example, to find possible future tasks, choice points, cycles, and alternative ways the procedure may play out. Many CALO functions, for example, query answering and explanation, require that a system be able to introspect on its own general procedures. To meet these requirements, we devised an interchange format called the Portable Process Language (PPL) that can be used for exporting a declarative fragment of the procedures represented in SPARK. In this section, we will explain the design of PPL in more detail.

**Querying and Explanation Requirements for Procedures**

We studied the queries and explanations that needed to be supported using SPARK, and in this section we explain those requirements.

As a general-purpose assistant, CALO is expected to field answers to a wide variety of questions from a user, including about its own knowledge of how to do things. While SPARK's execution trace API allows CALO to answer questions about specific things it has done (e.g., "When you purchased the laptop, did you request
CAŁO also needs knowledge of the procedural flow chart itself for more general questions about procedures, for example,

1. How do you purchase a laptop?
2. Will you need to access to the Web during the purchase?
3. Is authorization required [i.e., is there an authorization step] to purchase a laptop over $2000?
4. What will happen if the authorizing manager is unavailable?
5. Email the quotes you find to my home email address.
6. Get authorization from Joe, not Steve.

The first four of these questions are independent of any specific execution of the procedure, and necessarily require access to the general procedure itself to answer the questions. The last two questions are in the context of a partially executed procedure, in which the user is making reference to a to-be-executed future step. Again with these two questions, the system needs a representation of the general procedure to identify the future step to which the user is referring (these are not in the execution trace, as they have not yet happened).

Answering "why" questions need for explanations requires introspecting not just on what happened, but the specific tests and conditions that caused those things to happen, a particular form of meta-reasoning. Again, knowledge of the structure of the process flow chart is often required to answer these questions, including details of tests directing flow of execution at branch points, and details of how earlier steps support later steps. (In its current form PPL does not capture all this knowledge, but it is a step toward this.) Example questions from the user to CAŁO include

1. Why didn't you ask for authorization?
2. Why did you send the purchase request to Dallas?
3. Why haven't you started searching yet?

**Design of PPL**

The Portable Process Language (PPL) represents the "flow chart" information using a filtered view of the SPARK process models to include tasks, subtasks, task parameters, and task ordering. It does not currently capture the semantics of conditional tests, context, and other logical evaluations.

Consider a toy example of a two-step process—namely, going to work—with a trivial two-step flow chart consisting of (i) entering a car and then (ii) driving the car to work. The PPL for a skolem procedure instance of this process is follows:

```
(instance-of goto-work1 goto-work-step)
(instance-of enter1 enter-step)
(instance-of drivel drive-step)

(possibleTask goto-work1 enter1)
(possibleTask goto-work1 drivel)
(followedBy enter1 drivel)
```
In PPL, each of the steps in the flow chart is denoted by a specific individual. These individuals are not instances of events in the world (e.g., the specific event of entering the car at a certain time); rather, they are instances of steps in a flow chart. Similarly, each parameter (e.g., the person, the car) for a flow chart step is denoted by a specific individual. These individuals are also not instances of objects in the world (e.g., a specific person or car); rather, they are instances of parameters (variables) in the flow chart. In other words, PPL is a literal logical depiction of flow chart steps, rather than of event sequences. We can relate flow chart steps to event types (classes), and flow chart parameters to object types (classes), with some simple correspondence statements:

(event-type-for goto-work-step Goto-work)
(event-type-for enter-step Enter)
(event-type-for drive-step Drive)
(object-type-for person-var Person)
(object-type-for car-var Car)

where Goto-work, Enter, Drive are types (classes) of actual events in the world, and Person, Car are types (classes) of actual objects in the world, all of which are drawn from the CALO ontology.

Given a PPL flow chart, in principle one could execute it to perform the plan it denotes. However, the AI planning community has long matured beyond these kinds of toy executions—in the real world, plan execution also involves plan monitoring, recovery in the case of failure, consideration of time and resource constraints, and so on. To adequately represent such executable processes, much richer languages are needed, and this is precisely the role of SPARK. Rather, one should view PPL as capturing a simple abstraction of a complex executable process, in a language-neutral form suitable for introspection, to support the kinds of tasks discussed earlier.

For contrast, a simple SPARK representation of the above procedure would be

{defprocedure Goto_Work_by_Car
  cue: [do: (goto_work $person $car)]
  precondition: (True)
body:
[seq: [do: (enter $person $car)]
 [do: (drive $person $car)]]}

While suitable for execution, this syntactic structure is difficult to manipulate for introspection. Note also that some information in this SPARK representation is implicit: PPL makes explicit the step ordering (implicit in the ordering of lines of SPARK-L representation) and step-substep relations (implicit in the grouping of tasks within a single procedure in SPARK-L). This allows other tools easy access to the process itself. PPL denotes conditional branches in a flow chart using a predicate shown below:

(conditionalFollowedBy <step> <test> <next-step>)

meaning that if execution is at <step>, and <test> evaluates to true, then the next step will be <next-step>. At present, <test> is the opaque (quoted) logical expression copied from the SPARK-L test itself. For example, from the SPARK procedure for purchasing a laptop, the step "relax-and-redo" follows "laptop-query" only if no quotes were found. This appears in PPL as

(conditionalFollowedBy laptop-query1
   (and "(= $temp_quotes []")
   relax-and-redol)

This allows the external systems to see that relax-and-redol is a possible next step in the plan, but not at present to understand details of conditions under which it will be executed.

While it is not our intention that PPL capture the full details of the original SPARK process models, it is clear that there are additional details that should be captured. These include preconditions, "cue" conditions, and better handling of logical assertions and tests in the original SPARK. These are all items for future work.

**Implementation of PPL**

The PPL is generated by introducing specific individual names for each variable, for the cue task, and for each atomic step, such as [do: A] in the procedure. SPARK action symbols, such as `enter`, become event types. Type and role declarations for the actions are translated into object type statements and role statements. Thus, for an action `enter` with parameter roles `agent` of type `person` and object of type `car`, we translate

[do: (enter $person $car)]

into

(instance-of enter1 enter-step)
(event-type-for enter-step enter)
(agent enter1 person1)
(instance-of person1 person-var)
Each of the specific individuals corresponding to the atomic steps is related to the individual corresponding to the cue by possibleTask. These tasks are considered only “possible” because conditional branching or unexpected failures may prevent the tasks from being attempted:

(possibleTask goto-work1 enter1)

Of more interest is the ordering relationship between the atomic steps, represented by the followedBy and conditionalFollowedBy predicates. Determining this relationship requires walking over the body task network expression, and keeping track of all the possible prior atomic steps and the sequences of conditions that must be satisfied for the current step to follow each of these. We have to consider multiple prior atomic steps when considering a step following a parallel or select construct. An atomic step following a context construct or within a select or wait will be executed only if the appropriate conditions holds, and there may be multiple conditions between the execution of one atomic step and another. To translate iterative constructs, we need to introduce “null” tasks to link the start and end of the loop.

For simplicity, we have ignored the possibility of alternative procedures for the same action. However, this can be represented by making each procedure a subtype of the event type for the action, and the making the cue an instance of a step of that subtype.

A translator from the SPARK representation language to PPL is part of the CALO system. Using this translator, we exported SPARK process models. The exported information was used in the query answering process.

**Evaluation of the Knowledge Base**

Since the role of the ontology in the CALO system was as an enabler, it is difficult to evaluate its contribution in a straightforward manner. While we could conduct studies that measure how many people agreed with the key term definitions we came up with, and whether our taxonomy satisfied some abstract properties, the ultimate success or failure of the ontology is tied to the extent to which the CALO system overall was able to achieve its goals. It is also not economically practical for us to build two versions of CALO with identical set of capabilities – one with an ontology, and one without the ontology and report a comparison. The CALO system did undergo evolution and the ontology changed with evolving requirements. There are lessons to be learned from that evolution that we document in a later section, but that by itself cannot constitute the sole evaluation. It is also not straightforward to answer the question whether the specific ontology that is currently being used in the system is better than any other ontology. Therefore, our evaluation relies on documenting two specific functions of the CALO system that heavily rely on the ontological information. The CALO functions that we consider here are: use of ontology in task learning, and use of ontology in learning
semantic desktop relations. These functions would not have been possible unless there was ontology in the system.

**Use of Knowledge Base in Task Learning**

LAPDOG (Learning Assistant Procedures from Demonstration, Observation, and Generalization) is one of the several task learning components in CALO [26]. It provides the capability of learning procedures from demonstration.

An action is defined by a set of input and output arguments. Let $A(i_1 + +i_m -o_1 ... -o_n)$ denote the action $A$ with input arguments $(i_1, ..., i_m)$ and output arguments $(o_1, ..., o_n)$. A demonstration consists of a sequence of actions with specified arguments. Given a demonstration, LAPDOG’s task is to generalize it into an executable procedure. This involves two major operations: dataflow completion and generalization. By dataflow completion, we mean that there must be a way to supply value for every input for an action, for example, either by input from the user, or from an output of a previous action. By generalization, we mean that given a specific task instance as a demonstration, LAPDOG must come up with a procedure to execute the Task class for any of its other specific task instances. As a side effect, LAPDOG also determines the task signature—specifically, the inputs required by the procedure and the outputs generated by its execution.

For a procedure to be executable there must be a way to compute every input of every action. Because demonstrations may include unobservable actions (e.g., mental operations by the user), the dataflow in the observed actions may be incomplete (i.e., there may be no way to compute certain inputs). LAPDOG attempts to find additional actions that could support these unsupported inputs and complete the dataflow. There are two ways LAPDOG can complete a dataflow. The first is through the insertion of information-producing actions to generate required inputs from known outputs. LAPDOG determines these actions by running a depth-bounded forward search over the space of information producing actions, essentially creating an information plan for generating the required inputs. Examples of information-producing actions are information extractors and string manipulation operations. The second method for completing a dataflow is through a KB query to follow a relational path from known outputs to required inputs. This involves the use of Pathfinder, a KB utility for finding connections between objects in the KB [26]. Since this process relies on the ontology in the KB, we illustrate this with a more detailed example.

Suppose LAPDOG observes that a user *Karen* is sending an email to recipient *Perrault*. For generalizing this procedure, LAPDOG must determine how to compute the value of the recipient. The recipient of an email can depend on multiple factors such as the content of the email, and the relationship between the sender and the recipient. Since the KB represents the relationships amongst various individuals of interest, it is possible to enumerate possible relationships, prioritize them on some heuristics, and present them to the user for validation. As a specific illustration, suppose the KB contains the following relationships shown in Figure 2 amongst *Karen* and *Perrault*:
The top-most path in the graph represents that Ray is a supervisor of Karen, and the path below it in the graph represents that Karen and Ray attended an Event that was related to a project for which Ray is the project leader. The Pathfinder utility incorporates a number of heuristics that help filter out interesting paths from uninteresting ones. For example, one heuristic prefers paths that uniquely identify the target. For example, in the situation considered here, there may be paths in the KB from Karen to any of her colleagues, for example, Adam, Tom, etc, but there is only one path to Ray. As per the heuristic to prefer fewer targets, a path to Ray that has only one target will be preferred over the path to Adam, Tom, etc. that has multiple targets.

The Pathfinder utility pre-computes the populated paths between objects of certain types to greatly reduce the search space and improve efficiency when finding relations between objects during runtime. After an initial ramp-up period, new objects and relations will not be added very frequently to the KB. Pathfinder takes advantage of this relative stability to periodically crawl the KB to update its record of populated paths. It stores this information in an A(k) index [27], a highly efficient indexing structure for answering path queries over graph-structured data.

The LAPDOG module has a procedure generalization capability, and a user interaction capability that support the rest of the steps in acquiring procedures. A detailed description of those steps is available elsewhere [26]. Once a procedure has been learned, it is registered in to the CALO task taxonomy using the task ontology described earlier after which it is available for execution as necessary.

The LAPDOG module made extensive use of the Pathfinder in learning procedures and their subsequent validation and understanding by the users. The overall performance of Pathfinder would not have been possible without the CALO ontology. The LAPDOG and Pathfinder provide a compelling use of the knowledge in the KB to support the reasoning necessary to learn to new tasks.

**Use of Knowledge Base in Learning Semantic Desktop Relations**

In the Figure 3 below we show the fragment of the CALO ontology that was of primary interest for relational learning research in the project [16]. The solid edges between the
ovals denote the relations that are known with certainty, the dashed edges indicate uncertain relationships, and the asterisks indicate the relations learned by classifiers.

The CALO KB contains data about individuals’ emails, email folders, persons, projects, web pages and websites. In addition to the ground facts, there were a small number of rules in the system that could be used for inferring relations that are not known with certainty. For example, one of the rules in the system states that if a document is attached to an email, then the both the email and the document tend to be associated with the same project. End-users can write their own rules as well. The focus on a handful of classes and relations explains why we were able to get away with agreeing on the meaning of only a small number of terms, and a larger scale agreement across the ontology was not necessary. In IRIS, the natural language vocabulary was tied via the query processor to the ontological structure, so almost every end-user-facing ontology class, instance or relation could be queried or used in rules.

With the data conforming to the ontology fragment shown above, an inference engine was implemented to integrate reasoning amongst hand written rules, learned rules, and learned classifiers to perform joint reasoning and learning [16]. The implementation was based on a Markov Logic inference engine, and is described in detail elsewhere [16]. The primary role played by the ontology in the joint inference was to provide a shared vocabulary that was used by different learning modules, hand authored rules, and extracted facts. Without the shared vocabulary, and the commonly introduced definitions of the terms that were introduced in the paper, the relational learning module would not have been possible.
Summary Evaluation
In summary, we address the key questions that we had raised earlier. How well do the traditional KB development methodologies and upper ontologies work in a context in which the goal is not ontology research? What ontology support is needed for an information management application that leverages statistical machine learning? How can a KB support cognitive task assistance?

How well did traditional KB Development methods apply?
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 [4]. In practice this approach worked very well. There are elements of KB development approaches advocated by others that would not have been very suitable for the current project. For example, the Methontology approach advocates a formal specification of requirements which was not practical for this project [20]. The most we could have hoped to allow the consumers to state the requirements either in plain English, or as lists of classes and relations that needed to be represented. The approach of using a set of competency questions worked well in bounding the scope and the design of the task ontology [25]. Finally, the collaborative processes were largely informal, and we did not use a formal structure of the sort advocated by the Diligent approach [28].

What ontology support is needed for machine learning?
The needs of the current machine learning methods are driven by the set of entities and their properties that can be directly observed, and form the basis for learning properties that are not directly observed. For the CALO system, such entities and relationships are captured in Figure 2. The machine learning methods did not make use of the taxonomic structure in the knowledge base [29], and thus, the structure of the rest of the ontology, and other semantic information such as the domain and range constraints were of little use to statistical machine learning methods.

The above experience should not be viewed as a definitive answer to the needs of machine learning, as the deeper integration between machine learning and declarative knowledge bases is still an open problem. Machine learning requires extensive engineering of features that serve the basis for learning which still require lot of manual effort and experimentation [30]. A possible research direction is to investigation if such features could be generated from a multi-functional declarative knowledge base capturing the knowledge of a domain.

How can a KB support cognitive task assistance?
The use of the CALO KB in the Task Learning scenario is an excellent example of how a knowledge base can support cognitive task assistance by providing reasoning support. The knowledge in the KB provided the basis for learning rules of inference, the task ontology provided a framework into which newly learned actions could be categorized, and then invoked at runtime. The combination of these capabilities illustrates how the knowledge enabled the kind of robust behavior that the CALO system was designed for.
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. We summarize the lessons in the table below and then provide additional detail.

<table>
<thead>
<tr>
<th>Key Lessons Learned</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. For integrating diverse agents, it was more useful to focus on agreeing the meaning of a small number of terms that each of the agents needed to know about as opposed to imposing an upper ontology to be shared by all of the agents.</td>
</tr>
<tr>
<td>2. The interchange language does not need to provide full expressiveness supported by each of the agents. It was sufficient to provide interchange only at the level of data that each agent needed to communicate to others.</td>
</tr>
<tr>
<td>3. The development process should provide support for using a heterogeneous set of tools and languages for ontology development.</td>
</tr>
<tr>
<td>4. 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.</td>
</tr>
<tr>
<td>5. We need to provide a way to combine the formal representation in the KB with the less formal and more direct representations.</td>
</tr>
<tr>
<td>6. Acceptability of an ontology can significantly diminish unless we provide tools for its evolution that support migrating the data across different versions of the ontology.</td>
</tr>
<tr>
<td>7. A new breed of learning systems is now possible that combine explicit and implicit knowledge and the user input can play an important role in bridging between the two forms of acquiring and representing knowledge.</td>
</tr>
</tbody>
</table>

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.

In our initial design, we used the CLib as the upper ontology. For semantic exchange of information, and for storing the user data, the upper ontology was not of much direct use, and was, therefore, de-emphasized during the simplified design. The relational model (the ontology fragment shown in Figure 2) that was the basis for most of the learning activities proved to be the most used fragment of the ontology.
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.

**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 [18] 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 partial information exchange.

**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 the 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 that was developed by multiple contributors, 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 [19]. (Recall that OWL has three sublanguages: OWL-Lite, OWL-DL, and OWL full [17]).

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.

**Scenarios and Use Cases**

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.

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, To Do list items, 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.

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.

Role of Explicit User Input

In classical knowledge acquisition and learning literature, a distinction is drawn between two classes of methods: methods in which knowledge is obtained from a human and is represented explicitly vs methods that incorporate knowledge into programs in other ways or represent it implicitly [31]. In the Task Learning use case, the knowledge already implicitly present in the knowledge base forms the basis of making suggestions to a user for possible relationships between objects that in turn help the system to learn new rules. In the use case for learning relations for a Semantic Desktop, some of the
knowledge is represented using rules explicitly authored by a human while the knowledge learned by statistical methods is represented implicitly by probabilities which can then be overridden by explicit input from the user. One general lesson one can draw from both of these use cases is that the traditional boundaries between explicit and implicit knowledge are starting to blur with new ways of combining the two where each makes sense, and the user input can play an important role in bridging between the two forms of acquiring and representing knowledge.

**Comparison to Related Work**

There have been several other projects on building cognitive assistants [32, 33] 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 [32]. In a similar vein, several systems have been created to investigate the semantic desktop concept, including the Haystack system at MIT [34] and the Gnowsis system at DFKI [35]. 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. In a closely related agent called RADAR [36], a knowledge representation system SCONE was used as a representation framework (http://www.cs.cmu.edu/~sef/scone/). Just as in CALO, their integration approach required sharing the definitions of only key knowledge structures such as messages, tasks, and annotations.

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.

Given our focus on representing tasks and procedures, it is interesting to compare the work reported here with related work. Our work built on the process representation language SPARK that was already available to us, and our claim is not about a new process language. The novelty of the work presented here is in defining how the procedures encoded in the procedure language of SPARK could be interfaced into a declarative ontology framework, and in creating a taxonomy of office tasks and using it to enable both learning by demonstration as well as statistical learning. To the best of our knowledge, there has been no prior effort on representing knowledge about procedures in an ontological framework. There have been prior efforts on the use of knowledge in learning new tasks. For example, the Tailor system is a procedure editing tool for directly authoring SPARK procedures [37], but it does not perform any reasoning over the knowledge base in learning tasks. The task representation that we consider here is closest to the one advocated for mixed initiative systems for collaborative problem solving [38]: one must be able to represent partial knowledge about tasks, ability to represent knowledge requirements of tasks, ability to represent tasks at different levels of details, and ability to represent execution of tasks by agents, and suitable for us in interpreting and generating natural language. We do not claim to have made progress on all the requirements outlined here for task representation for intelligent agents, but we believe
that ours is one of the only detailed accounts of a task representation that has been actually implemented in a functional intelligent assistant.

We also compare our work on PPL with two other well-known process languages: the process specification language (PSL), and OWL-S.

The PSL is a logic-based standard for capturing the semantics of processes [39]. PPL is intended as a complement to, rather than competitor of, PSL, as it captures process knowledge in a different way. The most significant difference between PPL and PSL is that PSL's representation of the ordering of steps is in terms of actual events ("activity occurrences"), while PPL orders the abstract flow chart steps ("activities") themselves. For example, in PSL the above toy plan for going to work would be

\%
enter is a subactivity of going to work
(forall (?person ?car)
     (subactivity (goto-work ?person ?car)
                  (enter ?person ?car)))

\%
driving is a subactivity of going to work
(forall (?person ?car)
     (subactivity (goto-work ?person ?car)
                  (drive ?person ?car)))

\%
In all occurrences of going to work, a driving occurrence
follows a entering occurrence.
(forall (?occ ?person ?car)
     (implies (occurrence_of ?occ (goto-work ?person ?car))
              (exists (?occl ?occ2)
                      (and (occurrence_of ?occl (enter ?person ?car))
                           (occurrence_of ?occ2 (drive ?person ?car))
                           (subactivity_occurrence_of ?occl ?occ)
                           (subactivity_occurrence_of ?occ2 ?occ)
                           (successor ?occl ?occ2))))))

The last axiom states that for all occurrences of going to work, there will be an occurrence of entering followed by an occurrence of driving. While this makes the semantics of the original flow chart explicit, the actual structure of the flow chart has been lost (the simple relationship “enter is followed by drive” is expressed as a complex quantified logical expression).

In principle one could perhaps recover the original flow chart by reverse-engineering it from these PSL axioms, either by parsing the axioms themselves or by theorem proving the general relationships (e.g., proving that driving always follows entering in goto-work). However, neither of these options is particularly easy. In contrast, our goal with PPL is to preserve the original flow chart structure so that it is directly accessible for other agents. One could imagine extending PSL to include some predicate "macros" that would allow the general flow chart relationships to be made explicit, and which would also expand to the traditional PSL axioms such as those shown above. Conversely, PSL makes explicit the actual semantics of the flow chart, and PSL could be
generated from PPL if one wanted to make these semantics explicit (indeed, PPL could be a “straw man” candidate for such PSL “macros”).

OWL-S is a OWL-based Web service ontology, which supplies Web service providers with a core set of markup language constructs for describing the properties and capabilities of their Web services in an unambiguous, computer-interpretable form [40]. Like PPL, OWL-S represents generic procedures themselves, and similarly uses individuals to denote process objects and parameters used by those processes (In this sense, PPL is more similar to OWL-S in approach than to PSL). Generally speaking, OWL-S process is not a program to be executed. It is a specification of the ways a client may interact with a service. A process can generate and return some new information based on information it is given and the world state, or it can produce a change in the world. This transition is described by the preconditions and effects of the process. Processes can be atomic or composite. The composite processes may have control structure such as sequence, parallel, split, join, if-then-else, etc.

Clearly, the scope and the applicability of OWL-S is much different from either SPARK-L or PPL. In terms of expressiveness, OWL-S is comparable to SPARK-L as both are expressive process description languages. OWL-S, PPL, and SPARK-L all use similar representations for inputs, outputs, and results. Over and above this shared core, PPL provides a representation for the control structure in a process by using the \textit{followedBy} relation. The current design of PPL is limited to just that. PPL is a subset of SPARK-L designed to support the requirements of reasoning applications. If one were to perform a similar reasoning over OWL-S processes, a PPL-like subset will need to be identified in order to avoid the potential overkill of using OWL-S in its entirety. In a similar vein, PPL has a simple logical syntax, intended to be easily generated and processed by sending/receiving applications, both abstracting out some details and making some implicit semantics of SPARK-L explicit (e.g., event ordering). In contrast, OWL-S is specifically designed to support Web-based services, and hence uses an RDF-based syntax, clearly appropriate for Web-based applications but possibly more cumbersome to deal with in the wider context of process communication.

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 yet 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 [4]. Even though we did not use a formal requirement specification language as advocated in the Methontology approach [20], 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 [41]. Finally, our collaborative processes were largely informal, and we did not use a formal structure of the sort advocated by the Diligent approach [28].

Summary and Conclusions
We presented our experience in engineering a knowledge base to meet the requirements of a cognitive assistant that learns to organize information and performs tasks. Our development was driven by the requirements of the project and had its foundation in agreeing the meaning of only a handful of terms. The rest of the vocabulary in the knowledge base was created to represent the tasks and procedures that were to be learned by the system. The knowledge base was used both by two different classes of learning methods to learn new tasks and procedures as well as semantic relationships amongst entities on a user’s desktop. Our experience shows the effectiveness of KB development methodologies that are driven by requirements, illustrates how the user input can bridge between apparently different learning methods, and exemplifies how a KB can enable two different forms of learning: learning in the context of a cognitive assistant.

References