Modeling Human-Agent Interaction with Active Ontologies

Didier Guzzoni and **Charles Baur**
Robotics Systems Laboratory
Swiss Federal Institute of Technology (EPFL)
1015 Lausanne, Switzerland
didier.guzzoni@epfl.ch, charles.baur@epfl.ch

Adam Cheyer
Artificial Intelligence Center
SRI International
Menlo Park, California 94025
adam.cheyer@sri.com

Abstract

As computer systems continue to grow in power and access more networked content and services, we believe there will be an increasing need to provide more user-centric systems that act as intelligent assistants, able to interact naturally with human users and with the information environment. Building such systems is a difficult task that requires expertise in many AI fields, ranging from reasoning, planning, scheduling, natural language and multimodal user interfaces. In contrast to many approaches to building agent assistants where diverse AI components are stitched together at a surface level, we propose an approach, called "Active Ontologies", and a toolset, called "Active", where a developer can model all aspects of an intelligent assistant: ontology-based knowledge structures, service-based primitive actions, composite processes and procedures, and natural language and dialog structures. We demonstrate this approach through an example prototype of an intelligent meeting scheduling assistant that communicates using instant messages and emails.

Introduction

According to conservative estimates of Moore's Law, computer hardware will surpass the processing power of the human brain sometime in the next fifteen to thirty years. Increased processing will improve today's CPU-bound components such as speech recognition and synthesis, real time vision systems, statistical machine learning, activity recognition and action planning systems. These technologies, coupled with the massive amounts of data and services accessible via the Internet, will allow the design and implementation of user-centric systems that act as "intelligent assistants", able to interact naturally with human users and with the information environment (Maes 1995).

Building software assistants is a difficult task that requires expertise in numerous Artificial Intelligence (AI) and engineering disciplines (Winikoff, Padgham, & Harland 2001). Perception of human activities is typically based on techniques such as computer vision or speech recognition. Natural language processors and dialog systems often require advanced knowledge of linguistics. Activity recognition approaches are frequently implemented using Bayesian or other statistical models. Decision making strategies and complex task execution are the responsibility of planning and scheduling systems, each potentially bringing a new programming language to learn. Finally, as planning unfolds, various actions are taken by the system to produce relevant behavior, often across a wide range of modalities and environments. Substantial integration challenges arise as these actions communicate with humans, gather and produce information content, or physically change the world through robotics. Testing and debugging an environment of such heterogeneous intricacy requires strong technical knowledge and a diverse set of tools.

To ease development and performance of intelligent assistant systems, we propose a technique where user interaction and core reasoning are deeply intertwined into a coherent and unified system, through a modeling and programming process we call Active Ontologies. Many intelligent assistant systems are built around powerful processing cores (reasoning, learning, scheduling and planning) connected with separate components in charge user interaction (language processing, dialog management, user interface rendering). It is often a significant challenge to perform the integration and mapping of a user-model of a problem domain to the background reasoning components and structures as significant information must flow back and forth during processing and dialog. Additionally, maintenance is a challenge, as improvements and new features require work on both front and backend layers, as well as to the inter-layer communication interface. By contrast, our approach is to provide a unified platform where core processing and user interaction come together in a seamless way. Our platform, Active, is a toolset and methodology to create intelligent applications featuring core reasoning, user interaction and web services integration within a single unified framework. To leverage Active Ontologies, where data structures and programming concepts are defined in ontological terms, software developers can rapidly model a domain to incorporate many of the best AI techniques and web-accessible data and services through a visual, drag-and-drop interface, easy-to-use wizards, and a familiar programming language. This is the vision we are pursuing through the development of the Active framework.

The following sections of this paper present the Active

framework in more detail, illustrating how user interaction and agent-based processing are simultaneously developed in an integrated way. We first look at related work on building intelligent software. Then, we outline the Active framework, tools, and methodologies. The next section presents how Active is used to implement a simple intelligent assistant able to organize meetings. Finally, a conclusion summarizes our results and outlines future directions of our work.

Related work

There are numerous examples of specialized applications aimed at creating intelligent user interfaces to assist humans in specific domains (Middleton 2002). Some projects focus on Internet assistance, were intelligent assistants can leverage a vast amount of information and services to help users with complex tasks (Morris, Ree, & Maes 2000). Scheduling meetings, managing an agenda and communicating also represent applications where intelligent assistants are relevant. Intelligent assistants are also used in the domain of smart spaces (Addlesee *et al.* 2001), where instrumented rooms are able to sense their environment and act upon events and conditions. Ubiquitous mobile computing is another domain for intelligent assistants. They allow users to access remote information and make use of a dynamic environment. In this field, generic rule based processing engines have been used as the core component of intelligent assistant frameworks. Projects are dealing with ubiquitous mobile applications (Biegel & Cahill 2004) provide powerful middleware platforms but lack natural language processing or multimodal fusion.

In contrast to these more narrowly focused AI applications, DARPA's recent PAL (Perceptive Assistant that Learns) program has funded two very ambitious projects that attempt to integrate many AI technologies (e.g. machine vision, speech recognition, natural dialog, machine learning, planning, reasoning) into a single system. The CALO project (Calo 2006) aims at designing and deploying a personal assistant that learns and helps users with complex tasks. CALO is an extremely heterogeneous system, involving components written in eleven different programming languages. CALO meets the requirements for which it was designed, but because of its diversity and complexity, is not a cognitive architecture suitable for use by any single developer. Similarly, the RADAR project (Modi *et al.* 2004) is a learning-based intelligent assistant designed to help users deal with crisis management. Its flexibility and sound design have allowed the system to be effectively deployed and tested by users in realistic situations. However, again, its complexity prevents a programmer to leverage its capabilities and approaches to other domains.

The design of user-centric systems often involves reasoning systems connected to language processing and user interfaces. To design intelligent applications, user interaction components have been layered on top of core AI components such as constraint solvers (Berry *et al.* 2005), BDI reactive planners (Morley & Myers 2004) or machine learning (Segal & Kephart 2000). Instead of a layered application design, we propose a more unified approach where user interaction, language processing and core reasoning are deeply integrated in a unified framework. Core processing of our applications may not be as sophisticated and advanced, and we rather focus on ease of design, implementation and deployment of intelligent assistants. In addition, deep integration of reasoning, language processing and user interaction components allows for easier dynamic multimodal fusion and more reactive user dialog.

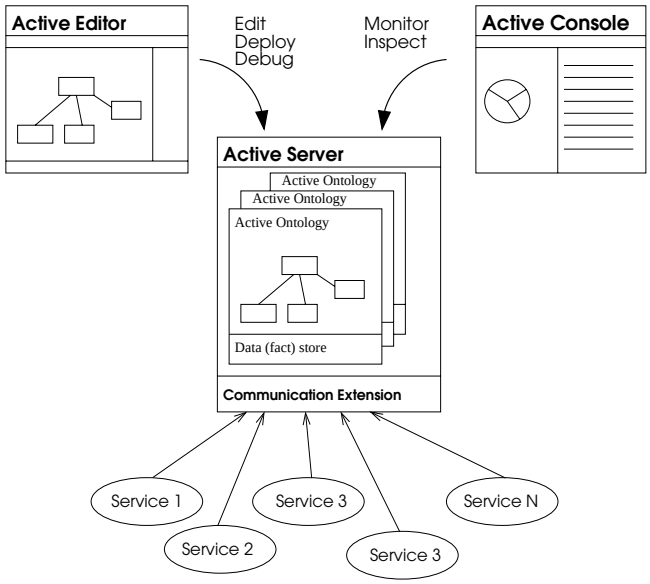

Figure 1: Active application design

Perhaps the closest work to Active is the SOAR project (Laird, Newell, & Rosenbloom 1987). In development since 1983, SOAR offers an open, unified framework for building intelligent "cognitive" systems using a foundation based on production rules. It incorporates planning and reasoning technologies, truth maintenance, machine learning, and has been integrated with natural language systems (Lehman, Lewis, & Newell 1991). The Active Ontology approach offers some advantages in rapidly modeling an application, but in many ways, Active can be thought of as a lighter-weight, developer-friendly version of SOAR that works well in an Internet and web-services environment.

The Active framework

Active introduces the original concept of *Active Ontologies* as the foundation for creating intelligent applications. Whereas a conventional ontology is a data structure, defined as a formal representation for domain knowledge, with distinct classes, attributes, and relations among classes, an Active Ontology is a processing formalism where distinct processing elements are arranged according to ontology notions; it is an execution environment. Active Ontologies are made up of a relational network of concepts, where concepts serve to define both data structures in the domain (e.g. a meeting has a time, a location, a topic and a list of attendees) as well as associated rule sets that perform actions within and among concepts. An Active-based application consists

of a set of loosely coupled services working with one or more Active Ontologies (see figure 1). Using loosely coupled services eases integration of sensors (e.g. speech recognition, vision systems, mobile or remote user interfaces), effectors (e.g. speech synthesis, user interfaces, robotics) and processing services (e.g. remote data sources, processing components).

Active Tools

Active consists of three component tools, the Active Editor, the Active Server and the Active Console.

- The Active Editor (see figures 2 and 3) is a design environment used by developers to model, deploy and test Active applications. Within Active Editor, developers can graphically create and relate concept nodes; select Wizards that automatically generate rule sets within a concept to perform actions like interpret natural language, model executable processes, or connect to third-party web services; and then developers can test or modify the rule sets as needed.
- The Active Server is a scalable runtime engine that hosts and executes one or more Active programs. It can either be run as a standalone application or deployed on a J2EE compliant application server. The Active server exposes a remote API (via SOAP or RMI) allowing external sensors component to report their results by reporting events, thus triggering actions within the deployed Active Ontologies.
- The Active Console permits observation and maintenance of a running Active Server.

The Active framework implementation is a Java-based software suite designed to be extensible and open. For both the Active Editor and Active Server, a plug-in mechanism enables researchers to package AI functionality to allow developers to apply and combine the concepts quickly and easily. A growing set of Active extensions is available for language parsing, multimodal fusion, dialog and context management, and web services integration. To ensure ease of integration and extensibility, all three components of the Active platform communicate through web service (SOAP) interfaces.

Active Fundamentals

At the core of Active is a specialized rule engine, where data and events stored in a fact base are manipulated by rules written using JavaScript augmented by a light-layer of first-order logic. JavaScript was chosen for its robustness, clean syntax, popularity in the developer community, and smooth interoperability with Java. First-order logic was chosen for its rich matching capabilities (unification) so often used in production rule systems.

An Active Ontology is made up of interconnected processing elements called concepts, graphically arranged to represent the domain objects, events, actions, and processes that make up an application. The logic of an Active application is represented by rule sets attached to concepts. Rule sets are collections of rules where each rule consists of a condition and an action. When the contents of the fact store changes, an execution cycle is triggered and conditions evaluated. When a rule condition is validated, the associated action is executed. Rule conditions are represented by a combination of fact patterns, compared with the content of the fact store using a unification mechanism. Unification is essentially a boolean operation to compare two facts. If two facts unify, they are considered as equivalents. Variables play an important role in the process of unification, where they can be seen as wild cards that always unify with their counter parts. When a variable unifies with a fact, it is instantiated with the matching fact and, for the rest of the unification process, is not considered as a variable anymore but as a known bound value. In the process of Active rule evaluation, instantiated variables of the condition can be used in the action code of the rule.

To help model time based constraints, fact stores manage the life cycle of facts. When a fact is inserted into the fact store, additional optional information can be specified to define when the fact should actually be asserted and when it should be removed. Both a controlled life cycle management of facts and modeling of events in rule conditions provide Active with a time dimension that allows Active programmers to model complex time-based behaviors. Finally, relationships among concepts define additional knowledge about the domain. Depending on the application modeled with Active, relationships carry attributes such as cardinality, weight or priorities.

AI Methodologies in Active

On top of the basic language elements and techniques described in the previous section, we have been encoding a number of AI approaches using Active, making them available as reusable Active Ontologies or Concept wizards for rapid assembly by developers.

Language processing

The goal of a language processing component is to gather input utterances, understand their meaning, and to finally generate a command to be delegated for execution. In Active, we currently offer two componentized approaches to natural language parsing (NLP):

1. Chart parsing (Earley 1970), where we use a bottom-up, unification based grammar approach to model the target language (e.g. *Sentence* $\Rightarrow$ *NounPhrase*, *VerbPhrase*) using Active concepts;
2. Pattern recognition, where we use Active concepts to model the domain in question and then add a light layer of language (words and patterns) inside each concept. This approach is often very natural for developers, produces good results and the domain model is portable across languages.

We will describe the pattern recognition approach in more detail, as it is a less-standard approach to NLP with some practical benefits. To implement the pattern recognition approach for a domain, the first step consists of using concepts and relationships to specify the model of the application (see figure 2). A tree like structure is built, defining the structure

of a valid command. For instance, in our example a meeting is made of a set of *persons*, a *topic* , a *location* and a *date*.

Once the domain has been defined using concepts and relationships, a layer of language processing is applied, by associating rule sets directly on the domain concepts. Active's unique design allows programmers to model the domain of an application and the associated language processing component in a single unified workspace.

The domain tree has two types of processing concepts: sensor concepts (leaves) and node concepts (non-leaves). Sensor concepts are specialized filters to sense and rate incoming words about their possible meaning. A rating defines the degree of confidence about the possible meaning of the corresponding sensed signal. Typically sensor concepts generate ratings by testing values of word sequences. For instance, the *topic* leaf is in charge of detecting meeting topics and is therefore instrumented with the following rule : *if two consecutive words are received and the first one is "about", then rate the second one as a topic*. Therefore, if the user provides, anywhere in a sentence, the combination *"about XXX"*, then *XXX* will be reported as the topic of the meeting. Word values can also be tested using regular expression pattern matching or a known vocabulary set. For instance, the *date* concept has a set of specialized rules able to convert utterances such as *"next Monday at 2 am"* or *"tomorrow morning"* into date objects of the form: *date(DAY, MONTH, YEAR, HOURS, MINUTES)*. When detected, date objects are created and reported to the *meeting* concept. Sensors use communication channels to report their results to their parents, the node concepts. There are two types of node concepts: gathering nodes and selection nodes. Gathering nodes, e.g. the *meeting* node in our example, create and rate a structured object made of information coming from their children. Selection nodes pick the single best rating coming from their children. Node concepts are also part of the hierarchy and report to their own parent nodes. Through this bottom up execution, input signals are incrementally assembled up the domain tree to produce a structured command and its global rating at the root node. To simplify the task of building Active based language processing, this technique has been encapsulated into a set of Active extensions and wizards. The Active Editor provides wizards allowing programmers to interactively define rules to apply to both sensors and node concepts. At the end of a sequence of simple interactive steps, wizards automatically generate processing rules and attach them to associated concepts.

Relationships play an important role in our approach. Two types of relationships can be used: *structural* and *classification*. Structural relationships (arrow-ended links on figure 2) represent structures by binding elements together, they relate to a *"has a"* ontological notion. For instance, *topic*, *eventdate*, *location* and *person* are members of a *meeting*. Structural relationships also carry cardinality information and record whether children nodes are optional, mandatory, unique or multiple. For instance, the relationship between *person* and *meeting* is *multiple* and *mandatory* . This information is used by Active to provide the user with contextual information. For instance, if a user initiates a dialog with *"schedule a meeting with john doe"*, if the *location* node is linked as *mandatory* , the user will be asked to provide a location. Through this mechanism, the parsing tree not only generates a structured command but also builds dynamic information to interactively assist the user. Relationship can also represent types of nodes to link children to a selection node. They relate to a *"is a"* ontological notion. For instance, let us imagine a more complex assistant able to not only organize meetings but also organize trips. Based on the user's input a root node *command* would have to choose between scheduling a meeting or organizing a trip.

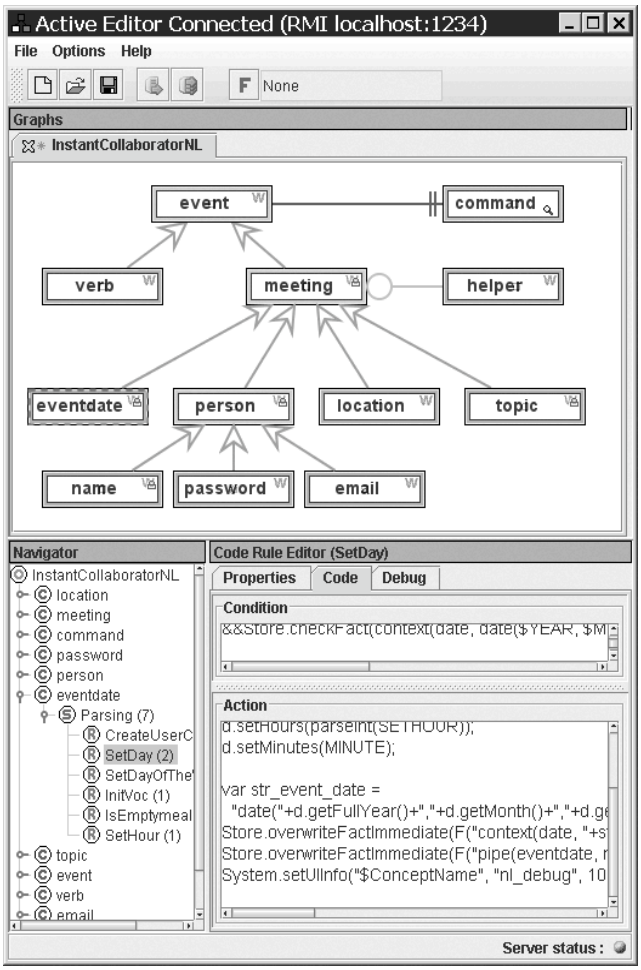

Figure 2: Language processing Active Ontology in the Active Editor

The Active network of concepts holds the context of the conversation. Nodes and leaves remember their current state, allowing the user to use partial sentences, or incomplete information when necessary. One can say *"schedule a meeting with john doe about hiring"*and provide a second utterance later as *"tomorrow morning in room 302"* to set the meeting's location or *"and john smith"*to add a participant. This type of dialogue is well suited for natural interaction where incomplete sentences are generated by the user.

Changes to language domain definition and processing are easily and graphically done through the Active Editor. There

is no code to edit, system to shutdown nor program to recompile. For instance, to add a new attribute *phone number* to a *person*, a user uses the *Add Leaf* wizard to define parameters and rules that control how leaf nodes rate incoming words. The wizard automatically generates a new concept and its processing rules. The next step consists of graphically connecting the new leaf with the *person* node with a *structural* relationship and specifying its attributes (mandatory, cardinality). The final step is to redeploy the modified Active Ontology onto the Active Server.

Dynamic service brokering

At the heart of many multi-agent systems, such as SRI's Open Agent Architecture (OAA) (Cheyer & Martin 2001) or CMU's Retsina (Sycara *et al.* 2001), is a dynamic service broker which reasons about how to deal with dynamic selection of service providers. In such systems, a brokering mechanism is used to select on the fly relevant providers on behalf of the caller. Service providers are chosen based on a service class and a set of selection attributes, which typically include properties such as service availability, user preferences, quality of service, or cost. Meta-agents can additionally provide third party recommendations about specific service providers. In such delegated model, callers do not specify which service provider to call, but rather the type of service to call.

To implement this technique, we have created a specialized Active Ontology to work as a service registry and dynamic service broker. Service providers register their capabilities and attributes by asserting a set of facts. This data set represents a simple service registry where providers can register their capabilities to later be discovered and invoked.

When a caller requests usage of a service, the broker selects which providers can be called based on the caller's attributes and current context. Once a list of suitable providers have been selected, the broker invokes them using one of two policies:

- Sequential: Providers are called in sequence, until one of them successfully responds. This would for instance be used to send a notification message to a user. If several service providers can send email, the message should be delivered only once.
- Parallel: Providers are concurrently invoked, and their responses are aggregated into a result set. This technique is used when a caller needs to retrieve information from multiple sources.

In addition to implementing the processing part of the delegation mechanism through facts and rules, Active is also used to model service categories. Concepts and relationships are used to define input and output parameters of service categories. Such a unified approach allows Active developers to graphically model an application domain and encoding its processing elements in a unified environment. The Active Editor provides a wizard allowing programmers to easily integrate and register any SOAP-enabled service as an Active service provider, mapping the SOAP inputs and outputs directly to Active domain models.

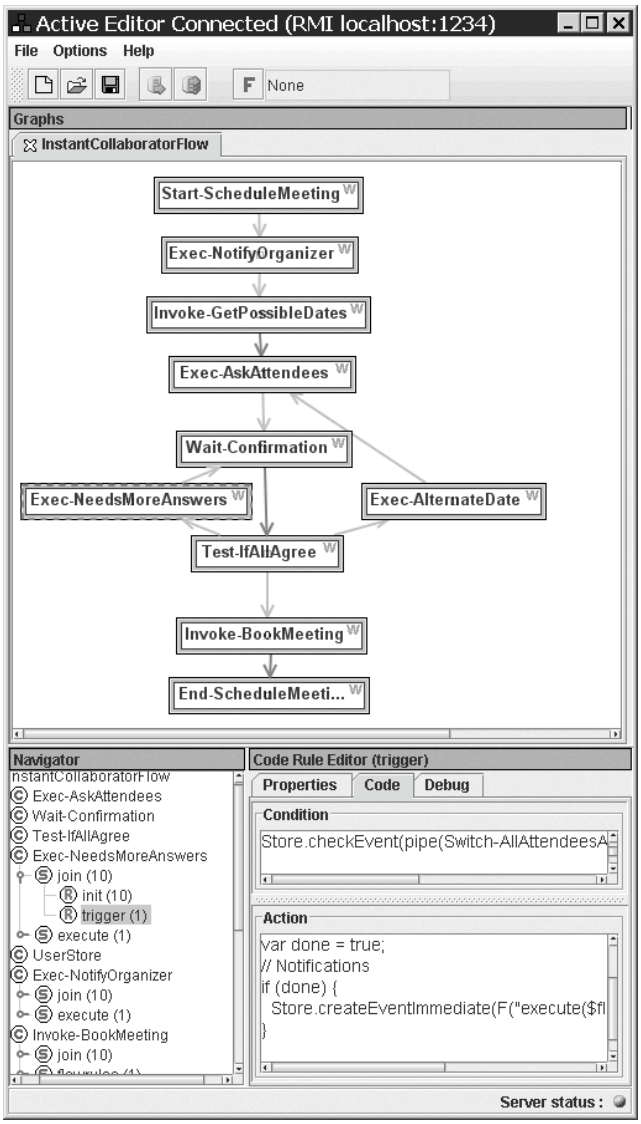

Figure 3: Process modeling in the Active Editor

Process modeling and execution

An Active methodology to model execution processes has been designed and implemented. Using concepts and rules it is possible to model generic processes, hence using the Active environment as a basic business process engine. Typically, processes model user dialog execution and sequences of actions to be undertaken by Active. As other Active methodologies, this technique has been encapsulated into a set of interactive Active Editor wizards allowing programmers to model complex processes without writing any code.

Processes consist of *nodes* and *transitions*. Nodes are represented as Active concepts and transitions as Active relationships. Processes have conditional *start* nodes in charge of triggering the execution of a process instance. Process nodes are activated upon reception of a execution token, passed among nodes through process transitions. The execu-

tion status of running processes and their instance variables are persisted as Active facts in Active data stores. Finally, when an *end* node is reached, the process instance terminates. A collection of functional building blocks are available to model various types of process nodes:

- *Start* nodes define entry points that trigger process executions. A start node has a condition based on the content of the Active fact store. Whenever the condition is valid, a unique process instance is created and its execution is started.
- *End* nodes terminate a process execution. They clean up all variables and data associated with the process to terminate.
- *Conditional Fork* nodes allow to model conditional branches (sub processes to be executed in parallel).
- *Execution* nodes contain Javascript code to be executed.
- *Wait* nodes have a condition based on the content of the Active fact store. Whenever the condition is valid, the flow will resume its activity. A timeout can be specified to undertake action when an awaited event does not occur.
- *Delegation* nodes perform delegated invocation of services (see section *Dynamic service brokering*). A timeout can be specified to resume the process activity when no response is generated by the Active delegation mechanism.

A simple Active based meeting assistant

To illustrate how Active can be used as the backbone of an intelligent assistant application, we present a simple system able to organize a meeting for a group of attendees. The assistant interacts in a natural way (plain English) with the meeting organizer and attendees through instant messages and email.

Scenario

The organizer uses an instant messenger client (*Yahoo! Instant Messenger* ^{TM}in our example) to interact with the intelligent assistant (See figure 4). The organizer can express her needs in plain English by sending sentences like: *"organize a meeting with john doe and mary smith tomorrow at 2 pm about funding"*. The assistant initially responds with a summary of what was understood. If any mandatory piece of information is missing (i.e. a location), this is communicated to the user. Specific details about the meeting can be updated or added by the organizer using partial utterances such as *"tomorrow at 7 am"* to adjust the time or *"with bob"* to add an attendee. After each user input, the assistant will also provide suggestions about what can be specified. Through this iterative process the organizer can define all details regarding the meeting to organize. When everything has been decided, the organizer triggers the task of actually organizing the meeting by sending *"do it"*or an equivalent utterance.

Then, the assistant uses web services to access public calendars (*Google Calendar* ^{TM}in our case) of all attendees to find a list of suitable slots for the meeting. Starting from the meeting time, each one-hour slot is checked against all attendees calendars. For each slot, if all attendees are available the slot is kept as a candidate for the meeting. Once the list of possible time slots for the meeting is gathered, the assistant sends an email to all participants asking if the first slot of the list is suitable for the meeting. If all attendees respond positively, the meeting is scheduled. If one attendee responds negatively, the next candidate time slot is selected and a new email message is sent to all attendees asking for approval. This process continues until a suitable date and time are found. If no date can be agreed on within the day of the attempted meeting, the process is aborted and the organizer is notified. If a date suits all attendees, the assistant will actually schedule the meeting by automatically adding an entry into all attendees calendars. Although this example proposes a fairly simplistic modeling of the scheduling process, it is sufficient to illustrate natural language interpretation and dialog, long running asynchronous processes, web-service delegation, and multimodal user interaction. We shall now look at the approach for developing this example application.

Core Application

The core of the application consists of three Active Ontologies implementing the overall behavior of the intelligent assistant: language processing, plan execution and service brokering. The rest of the application consists of SOAP enabled web services providing access to calendar information and two Active Server extensions providing email and instant messenger communications.

The first stage of the application performs language processing. Incoming utterances from the meeting organizer are gathered by Active server extensions from either the intelligent assistant's email account or its instant messenger client impersonation. Utterances are then asserted as facts to trigger an Active Ontology in charge of language processing based on the method described in section *Language processing*.

A second processing stage carries out the sequence of actions required to execute incoming user requests. Once a command has been generated by the language processing Active Ontology, it is asserted into the fact store of the Active Ontology implementing the logic of our scenario. The plan to execute has been modeled as an Active Ontology (see figure 3) using the process technique defined in section *Process modeling and execution*. Following a top down execution, the start node *Start-ScheduleMeeting* triggers the execution of the process. First, the organizer is notified through the execution of the *NotifyOrganizer* node that contains Javascript code to format user messages. Then, the *GetPossibleDate* node invokes an external SOAP service to get a list of possible meeting dates. The *AskAttendees* node then formats and sends email messages to all meeting attendees. The *Wait-Confirmation* node waits for incoming emails (deposited as facts by the Active email Extension) and passes control to the *Test-IfAllAgree* node. As a *switch node* , the *Test-IfAllAgree* node conditionally controls which node should execute next. Three possible situations are possible: If one of the attendees has rejected the

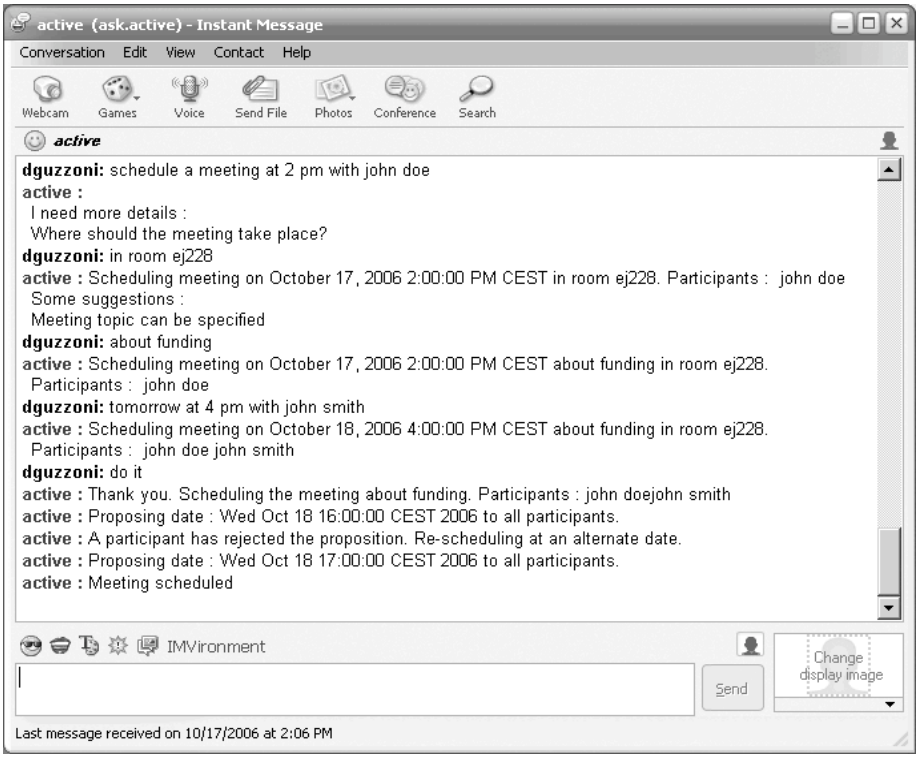

Figure 4: Example of user interaction

proposed date, control is passed to the *AlternateDate* node that picks the next possible date and resumes the notification process through the *AskAttendees* node. If more answers are expected from attendees, the systems loops back through the *NeedsMoreAnswers* node. If all attendees have positively responded, the execution control is passed to the *Invoke-BookMeeting* to actually schedule the meeting. Finally the *End-ScheduleMeeting* node terminates the process and cleans up all its instance data.

As the plan to execute unfolds, a third Active Ontology is used to dynamically pick and invoke external services. To perform its task, our meeting assistant requires external services to access the personal calendars of attendees, notify them (by email) and converse with the organizer (instant messages). All interactions with these external resources are ran through the delegation mechanism described in section *Dynamic service brokering*. The decoupling between the caller and service providers allows the caller to specify *what* is needed, not *who* nor *how* tasks should be performed. It allows for dynamic swapping of service providers without changing anything on the caller side. Service providers can be dynamically picked based on the user's preferences, its location or current availability of providers.

External services

Following the Active design pattern, our application is made out of one or more Active Ontologies and a set of loosely coupled services (see figure 1). In our case, the backend is based on the Google calendar API's to read and modify the personal calendars of meeting attendees who are supposed to have a Google calendar account. Google public API's have been exposed as SOAP web services for easy integration with the Active Ontology in charge of service brokering. The meeting organizer uses *Yahoo! Instant Messenger* ^{TM}whose public API has been integrated as an Active Server extension. Finally, an Active Server extension uses the POP protocol to regularly check its email account. Each incoming email is converted into an Active fact for processing. For instance, the *Wait-Confirmation* node (see figure 3) waits for incoming email messages asserted as Active facts.

Conclusion

In this paper we present an innovative architecture to develop intelligent assistants. The Active framework provides a unified tool and approach for rapidly developing applications incorporating language interpretation, dialog management, plan execution and brokering of web services. In contrast to related efforts, Active combines core reasoning and user interaction in a unified design to provide flexibility and dynamic user dialog. In a broader way, Active aims to unleash the potential of intelligent software by making required technologies more easily accessible. This paper shows how a simple assistant able to schedule meetings has been designed and successfully implemented based on Active.

More work remains to be done on application, implementation and methodology aspects of Active. First on the application side, in addition to the meeting assistant pre-

sented here, Active is used in different intelligent assistants systems. For instance, Active is used as the backbone of an application that helps mobile users access data and services through a natural email or short messages based dialog. In a different field, an Active-based intelligent operating room helps surgeons interact, through a multimodal approach mixing voice and hand gestures, with computer based equipments during surgery. For these applications, we plan to gather feedback and experience from users to help us improve and validate the flexibility and robustness of our approach.

On the implementation side, we are working on scalability and robustness of the Active Server. We are planning on building clusters of Active Servers, able to balance large workloads to host multiple personal assistants serving a large number of users. In parallel, the evaluation algorithm of the Active rule engine is being further optimized to improve scalability and performances.

For Active-based methodologies, we are exploring innovative AI techniques for activity representation and recognition. Our goal is to unify plan execution and activity recognition, so that an Active-powered assistant could look at the activities of a user, understand what is being attempted to proactively provide relevant assistance and even take over the execution of the task as appropriate.

Acknowledgments

This research is supported by SRI International and the NCCR Co-Me of the Swiss National Science Foundation.

References

Addlesee, M.; Curwen, R.; Hodges, S.; Newman, J.; Steggles, P.; Ward, A.; and Hopper, A. 2001. Implementing a sentient computing system. *Computer* 34(8):50–56.

Berry, P.; Myers, K.; Uribe, T.; and Yorke-Smith, N. 2005. Constraint solving experience with the calo project. In *Proceedings of CP05 Workshop on Constraint Solving under Change and Uncertainty*, 4–8.

Biegel, G., and Cahill, V. 2004. A framework for developing mobile, context-aware applications. In *PERCOM '04: Proceedings of the Second IEEE International Conference on Pervasive Computing and Communications (PerCom'04)*, 361. Washington, DC, USA: IEEE Computer Society.

Calo, C. 2006. http://www.calosystem.org/.

Cheyer, A., and Martin, D. 2001. The open agent architecture. *Journal of Autonomous Agents and Multi-Agent Systems* 4(1):143–148. OAA.

Earley, J. 1970. An efficient context-free parsing algorithm. *Commun. ACM* 13(2):94–102.

Laird, J. E.; Newell, A.; and Rosenbloom, P. S. 1987. Soar: An architecture for general intelligence. *Artif. Intell.* 33(1):1–64.

Lehman, J. F.; Lewis, R. L.; and Newell, A. 1991. Integrating knowledge sources in language comprehension. In *Proceedings of the Thirteenth Annual Conference of the Cognitive Science Society*, 461–466.

Maes, P. 1995. Agents that reduce work and information overload. In *Communications of the ACM*, volume 38.

Middleton, S. E. 2002. Interface agents: A review of the field. *CoRR* cs.MA/0203012.

Modi, P. J.; Veloso, M.; Smith, S.; and Oh, J. 2004. Cmradar: A personal assistant agent for calendar management. In *Agent Oriented Information Systems, (AOIS) 2004*.

Morley, D., and Myers, K. 2004. The spark agent framework. In *AAMAS '04: Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems*, 714–721. Washington, DC, USA: IEEE Computer Society.

Morris, J.; Ree, P.; and Maes, P. 2000. Sardine: dynamic seller strategies in an auction marketplace. In *ACM Conference on Electronic Commerce*, 128–134.

Segal, R. B., and Kephart, J. O. 2000. Incremental learning in swiftFile. In *Proc. 17th International Conf. on Machine Learning*, 863–870. Morgan Kaufmann, San Francisco, CA.

Sycara, K.; Paolucci, M.; van Velsen, M.; and Giampapa, J. 2001. The RETSINA MAS infrastructure. Technical Report CMU-RI-TR-01-05, Robotics Institute Technical Report, Carnegie Mellon.

Winikoff, M.; Padgham, L.; and Harland, J. 2001. Simplifying the development of intelligent agents. In *Australian Joint Conference on Artificial Intelligence*, 557–568.