Active : A unified platform for building intelligent assistant applications

Didier Guzzoni

October 25, 2007

Abstract

Computers have become affordable, small, omnipresent and are often connected to the Internet. However, despite the availability of such rich environments, user interfaces have not been adapted to fully leverage their potential. In our view, user interfaces will evolve to become more than simple tools that act using a *click-and-act* paradigm – people want software that can act as assistants to whom tasks can be delegated. This new type of software will provide us with more user-centric systems, able to interact naturally with human users and with the information environment. Although progress has been made in this direction over the last decade, current research in the field has shown that building intelligent assistants is a complex task that requires expertise in many fields, ranging from artificial intelligence to core software and hardware engineering. The difficulty of deeply integrating of all the technologies and AI methodologies required to produce robust, intelligent assistants has greatly limited the impact and widespread adoption of this form of software.

In this thesis, we propose to design, implement and evaluate a new methodology and associated tool suite whose aim is to ease and accelerate the development of intelligent assistant software. Active, our solution, introduces the original concept of Active Ontologies, and combines it with a service oriented architecture to serve as the foundation for user-centric applications. The Active software suite features a programming editor, a runtime server and associated management tools. Using this unified platform, we developed techniques for rapidly creating intelligent assistant applications that weave together language processing, process modeling and dynamic service orchestration.

To validate our approach, three prototypes have been implemented and evaluated. First, we created an assistant that helps mobile users retrieve information and access online services. The system allows users to retrieve information through natural language dialog about restaurants, hotels, points of interest, flights status and weather forecasts. As a second prototype, we have implemented an assistant to help surgeons in the context of the operating room. The system uses a combination of voice and gesture recognition to help surgeons navigate through pre-operative information, visualize and control a live video stream coming from an endoscope mounted on a robotic arm. Lastly, we have created a system that helps organize meetings through emails and instant messages with the organizer and attendees. These three prototypes are deployed in very different application domains, and yet are built with the same tools and methods; this demonstrates the flexibility and versatility of our approach.

We conducted a user study to validate our claim that, in specific domains, intelligent assistant software can perform better than conventional software approaches. First, we compared our mobile assistant against Google Mobile^{TM}, the leading commercially available search application for mobile users. We asked a population of users to accomplish ten travel-related tasks with both systems. Results show that, for the travel domain, our assistant-based system performs significantly better both in terms of effectiveness and time to completion than the more conventional, keyword-based search engine. In the medical domain, we asked a small population of surgeons to accomplish a set of tasks with our second prototype. After providing us with a positive feedback, surgeons used our

prototype to discuss and experiment with innovative forms of user interaction in the context of the operating room.

We also performed evaluations to support our claim that our unified approach accelerates the software development of intelligent assistants. First, we gave basic training on the Active platform to a population of software developers, asking them to go through a simple tutorial. Once trained, they were able to successfully program the core component of the mobile assistant prototype in under two hours. As a second validation, the meeting organizing assistant was designed, implemented and tested within three working days – comparable systems published in the literature were the product of significantly more time and resource investment. This demonstrates that a system encompassing numerous AI and software development aspects, including language processing, multiple modalities, contextual dialog, processing logic and backend services integration, can be rapidly implemented and tested using the Active framework.

Our work demonstrates that for many task-oriented domains, intelligent assistant applications can be significantly more effective than conventional systems. Through our Active Ontology approach and associated methodologies and tools, we have shown that complex intelligent assistant applications, perhaps for the first time, can be developed with low cost and high reliability. In addition to receiving a positive echo from the research world, our work is also causing ripples in the commercial world.

Résumé

Les ordinateurs sont devenus bon marché, petits, omniprésents et sont souvent connectés à Internet. Cependant, bien que cet environnement riche soit à notre disposition, les interfaces homme-machine n'ont pas été adaptées pour exploiter pleinement ce nouveau potentiel. Nous pensons que les interfaces utilisateur vont évoluer pour devenir bien plus que les outils simples qu'ils sont ajourd'hui. Leurs utilisateurs cherchent un nouveau type de logiciels, qui se comportent comme des assistants à qui des tâches peuvent être déléguées. Ces systèmes d'un nouveau genre vont nous permettre de créer des applications qui s'articulent autour de l'utilisateur et sont capables d'interagir naturellement avec les humains et leur environnement. Bien que des progrès ont été réalisés dans cette direction ces dix dernières années, l'état de l'art démontre que construire de tels systèmes est une tâche complexe, qui requiert des connaissances dans de nombreux domaines allant de l'intelligence artificielle à la maitrise approfondie de techniques d'ingénierie logicielle et matérielle. La difficulté à intégrer en profondeur toutes ces technologies pour produire des assistants intelligents efficaces et robustes, a limité l'impact et l'adoption à grande échelle de cette forme de logiciels.

La thèse présentée ici propose de concevoir, réaliser, mettre en oeuvre et évaluer un outil intégré, ainsi qu'un ensemble des techniques associées, conçus pour créer plus facilement et efficacement des assistants intelligents. A cet effet, notre solution Active, introduit le concept original d'Active Ontologies qui est combiné avec une architecture orientée service, pour modéliser et implémenter des assistants intelligents. La suite logicielle Active est composée d'un éditeur, un serveur et une console de gestion. Armés de cet outil cohérent et unifié, nous avons développé des techniques pour créer des assistants intelligents capables de traiter le langage naturel, modéliser la logique d'une application ainsi que de gérer dynamiquement une communauté de services.

Pour valider notre approche, nous avons réalisé et évalué plusieurs prototypes dans différents domaines. Nous avons tout d'abord crée un assistant dans le domaine de la recherche d'information pour des utilisateurs mobiles. Le système permet de poser des questions en langage naturel, par courriel, au travers de messages instantanés ou depuis un site web, relatives à des restaurants, hôtels, points d'intérêt ou encore prévisions météorologiques pour n'importe quelle ville des Etat-Unis. Dans le domaine aérien, le système permet également d'obtenir des informations détaillées et en temps réel au sujet la majeure parties de vols au départ ou à destination des Etat-Unis. Dans un tout autre domaine, nous avons utilisé Active pour implémenter et évaluer un assistant destiné à aider une équipe chirurgicale en salle d'opération. Pour se faire, nous avons réalisé un système qui combine la reconnaissance de voix et de gestes afin d'aider un chirurgien à accéder à des image préopératoires (2D et 3), visualiser et contrôler un flux vidéo provenant d'un endoscope monté sur un bras robotisé. Finalement, nous avons conçu un prototype d'assistant qui aide à l'organisation d'une réunion au travers d'échanges de courriels entre l'organisateur et les invités. Ces trois prototypes sont fonctionnels et, bien que très différents, ont tous été implémenté avec Active. Cela nous a permis de démontrer la flexibilité et la viabilité de notre approche.

Sur la base de ces prototypes nous avons montré que, dans des domaines spécifiques, l'utilisation d'assistants apporte plus d'efficacité que des logiciels

classiques. Nous avons tout d'abord comparé notre assistant pour utilisateurs mobiles avec le produit commercial Google Mobile^{TM}, une version spécialisée du populaire moteur de recherche. Pour se faire, nous avons demandé à une population d'utilisateurs de réaliser une dizaine de tâches avec les deux systèmes. Les résultats montrent que pour un domaine défini, un assistant intelligent est significativement plus efficace et plus rapide d'une moteur de recherche classique basé sur des mots clés. Nous avons également demandé à une population de chirurgiens d'accomplir une séquence de tâches avec notre prototype d'assistant pour salle d'opération. Après avoir été positivement évalué, le système a permis de définir des méthodes d'interaction efficaces entre chirurgiens et les ordinateurs utilisés en salle d'opération.

Nous avons aussi montré que notre approche unifiée facilite la conception et l'implémentation d'assistants intelligents. Pour se faire, nous avons donné une formation simple sur Active à une population de programmeurs. Ils ont ensuite conçu et implémenté, en une heure, une application capable de traiter le language naturel. Les programmes crées ont rempli 95% de leur cahier des charges, et traité avec succès quasiment 80% d'un ensemble de phrases tirées de requêtes formulées par des utilisateurs. Aussi, le système d'assistant organisateur de réunion a été réalisé en trois jours, démontrant qu'Active peut être utilisé pour rapidement créer des prototypes complets et fonctionnels d'assistants intelligents.

Nos travaux nous ont permis de démontrer que, dans certains domaines, des logiciels prenant la forme d'assistants sont plus efficaces que des approches classiques. Nous avons aussi montré que notre approche unifiée, basée sur le concept d'Active Ontologies, facilite la conception et l'implémentation des systèmes informatiques capables de traiter le langage naturel, entreprendre des actions complexes en orchestrant dynamiquement une communauté de services. Finalement, en plus des échos positifs reçus de monde de la recherche, nos travaux ont suscité un grand intérêt du monde industriel.

Keywords : Intelligent Assistants, Artificial Intelligence, Software Engineering

Acknowledgments

First, I would like to thank Charles Baur and Adam Cheyer for their initial suggestions, brilliant ideas, constant support, and for providing me with an environment in which I was able to lead my research very freely.

I would also like to thank all jury members, Prof. Michèle Courant, Prof. Alcherio Martinoli and Prof. Hannes Bleuler for their availability and interest in our work.

Completing this project would not have been possible without the support of my family, particularly my wife who has provided me with needed support, understanding and encouragement.

I would also like to express my fondness to my fellow co-workers of the VRAI group.

Finally, this project would not have been successful without the support of the NCCR Co-Me of the Swiss National Science Foundation and SRI International.

Contents

Document conventions

- *italic* : Used for emphasis and to signify the first use of a technical term, usually closely followed by a definition or referenced in the glossary.
- Constant Width : Used for all code snippets as well as for anything that you would type literally when programming. Also used for pseudo-code snippets expressed in a Java-like syntax.
- **Bold** : Occasionally used to emphasize relevant words.

Chapter 1

Introduction

1.1 Motivation

The easiest way to introduce the topic and motivation of our work is to consider the following scenario:

Dr. Stanley is a successful and highly skilled brain surgeon who lives in San Francisco. On an early Monday morning, he drives to the Stanford hospital where he is scheduled to perform a brain tumor removal intervention. He receives a phone call from his colleague, Dr. Livingstone from Boston. A patient, Mr. Smith, urgently needs an important operation that requires the unique skills of Dr. Stanley. Committed to helping, Dr. Stanley immediately agrees to fly out to Boston on the very same evening. After asking the social security number of the patient to operate, he calls his assistant to book an airplane ticket from San Francisco to Boston anytime after 5 pm and requests medical details providing Mr. Smith's social security number. In addition, he notifies his assistant about his new schedule so that existing appointments can be re-scheduled. Minutes later, he gets a short message about possible flight schedules to book. The message also indicates a traffic jam on highway 280 and advises switching to highway 101 instead. Successfully avoiding road congestions, Dr. Stanley arrives at his office and logs on his computer to start preparing for the morning brain surgery. He notices that all information about Mr. Smith has been delivered to his computer, organized appropriately. While working on pre-operation planning, he receives an email to confirm the flight he picked earlier. As Dr. Stanley walks to the operating room for the surgery, his pager notifies him that an email arrived for him about hotels to choose in Boston. As he gets to the operating room, he greets his staff and logs on to the operating room computer, which automatically loads all prepared information about the patient to operate. The surgery to perform is a frontal tumor extraction, the tumor being located on the surface, between the patient's skull and cortex. By interacting with his assistant Dr. Stanley gets helps to quickly and easily navigate pre-operative data reconstructed from the patient's brain. He also instructs his assistant to take photos, control light conditions and move the powered microscope enhancing his view of the work area. Effective collaboration in the operating room allows Dr. Stanley to accurately pinpoint the location of the tumor to extract. The operation is a success, and as Dr. Stanley leaves the operating room, his pager

delivers all pending messages received while he was operating. A message from Dr. Livingstone confirms the reception of the flight information and that he will personally be picking Dr. Stanley at the Boston airport.

This simple scenario introduces two types of assistants. The first one, helps with organizing daily tasks, making reservations and delivering relevant information at the right time, through the most appropriate channel. A second type of assistant is specialized to provide help in a specific environment, an operating room in our case. Both examples show how users can focus on the most important aspects of their activities by delegating background tasks to assistants. Today's technology is mature enough to create software systems implementing such intelligent assistants, able to naturally interact with users to understand requests and execute complex tasks. The goal of the work presented here is to explore, create and evaluate software tools and techniques to ease the design and implementation of such assistants.

1.2 Problem statement

Computer systems are constantly and rapidly growing in processing power, complexity and inter-connectivity. On the hardware side, even considering conservative estimates of Moores Law, in the coming decades microprocessors are likely to surpass the processing power of the human brain. Leveraging this potential, CPU intensive software applications can perform complex tasks more easily. For instance, components such as speech recognition and synthesis, real-time vision systems or rich graphic applications are now commonly used in software applications. In addition to hardware and software progress, computers are not isolated processing nodes anymore, they are part of a large worldwide network. The ability to easily create networks of computers paves the way to new software designs where applications are not large monolithic programs anymore, but made out of loosely coupled distributed independent services. Parallel advances along these three axes, hardware, software and networking, are dramatically increasing the potential of new technologies.

This triple push forward is driven by both the industry and the research world. Hardware manufacturers are supporting the development of a wired, and increasingly wireless, global network. Major players of the microprocessor industry are creating smaller, less power hungry and more powerful chips that become the computing core of not only computers, but mobile devices, cars and various appliances[70]. The software industry is also fueling the trend. Standards for reliable, secure and efficient service-oriented distributed architectures are being defined and implemented. In addition, lightweight operating systems are available to allow the creation of mobile applications. As a consequence, computer technologies are becoming more ubiquitous, more affordable, and more widely accepted by users and customers.

These dramatic improvements in networking, software engineering and processing power open the door to a new breed of software systems. Despite major breakthroughs in both hardware and software technologies, user experience with computer-based systems have not changed, and the opportunity to create software that better leverages these advances has not yet been exploited.

New technologies, coupled with the massive amounts of data and services accessible via the Internet, open the door to the design and implementation

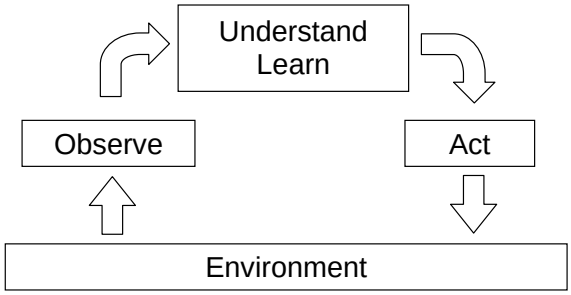


Figure 1.1: Intelligent Assistant Functional Diagram

of user-centric systems that act as intelligent assistants, able to interact naturally with human users and with the information environment[49]. In our view, user interfaces will evolve from simple tools using a *click-and-act* paradigm to become assistants to whom tasks can be delegated. Today's available and affordable technologies provide many scenarios where intelligent assistants could be given simple tasks. As an example, let us consider someone looking for a flight from San Francisco to Boston for a specific date. Instead of the user milling through multiple web sites to get quotes, one should be able to express the request in a more natural way by, for instance, simply sending an email to an intelligent assistant stating in plain English: *"find me the best flight from SFO to Boston next Thursday"*. The system would then start executing the task by automatically querying online sites or even paying services to get quotes. It would then send an email back to the user with a list of possible flights or a request for more details. Through such thread of email messages, intelligent assistants engage in natural and non-intrusive dialogs with users.

Personal assistant applications present intuitive user interfaces, for users to delegate tedious tasks while focusing on the most important aspects of their activities. As described in the previous scenario, personal assistants interact naturally through input (i.e. emails and instant messages expressed in natural language, speech and gesture recognition) and output modalities (i.e text generated emails, instant messages, sounds, speech synthesis or robotics). This mode of interaction saves users from learning about multiple program interfaces and unleashes the software component from its sometimes cumbersome keyboard and mouse. Voice-based natural interaction makes it possible to delegate tasks to personal assistants from anywhere. The ability to easily express and delegate tasks keeps the mind of users clear, allowing them to focus on their most important activities. Finally, personal assistants are constantly on alert to provide critical information through the most appropriate channel.

Intelligent personal assistants, also called interface agents, are software systems designed to process high level tasks delegated by human users[53]. The main idea is to move from the direct manipulation of a tool-like software program to delegated and dialog-based natural interaction. By nature, such software system consists of a mix of AI and HCI to provide three main functional features: sensing, reasoning and acting. First, to sense and observe its environment, including human communications, an intelligent assistant provides a

user interface component. Intelligent assistants should be able to interact naturally with human users using a wide variety of modalities, both synchronous (direct mouse, keyboard inputs or speech recognition) and asynchronous (email messages). Secondly, to analyze and understand a situation there is a need for natural language processing and activity recognition. Finally, to undertake a sequence of actions that will produce relevant and useful behavior planning and execution abilities have to be included.

Building software assistants is a difficult task that requires expertise in numerous AI and engineering disciplines [84]. 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.

1.3 Approach and original contribution

As previously described, intelligent assistant software is becoming a necessity but is difficult to design, implement and deploy. To the best of our knowledge, there is currently no integrated tool and methodology to easily and effectively build intelligent assistant systems. What if there were a toolset and associated methodologies that lowered the bar for creating intelligent applications, such that a single software developer could rapidly model a domain and then apply many of the best AI techniques to 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 and goal of the work presented here.

1.3.1 Unified approach

To ease the development of intelligent assistants and solve some of the problems described above, we propose a unified tool and associated methods for developing AI-based software.

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 or a problem domain into 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 provides a unified platform where core processing and user interaction come together in a seamless way. Our platform, Active, is a toolset and associated methods to create intelligent applications featuring core reasoning, user interaction and web services integration within a single unified framework. Using *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.

1.3.2 Active Ontologies

Our development suite, called Active, is based on the original concept of Active Ontologies, used to model and implement applications. A conventional ontology is defined as a formal representation for domain knowledge, with distinct concepts, attributes, and relations among classes; it is a data structure. An Active ontology is an enhanced ontology where processing elements are arranged according to ontology notions; the ontology becomes an execution environment.

An Active Ontology consists of interconnected processing elements called concepts, graphically arranged to represent the domain objects, events, actions, and processes that make up an application. In Active Ontologies, relationships among concepts not only represent information about how concepts are related, but also provide communication channels for the processing elements attached to concepts. The logic of an Active application is represented by rulesets attached to concepts. Rulesets are collections of rules where each rule consists of a condition and an action. Each Active ontology is associated to a data store, used to persist facts that represent the state and variables of the current processing. When the contents of the fact store changes, an evaluation cycle is triggered and concept conditions are evaluated. This innovative approach where an ontology and a production rules engine techniques are combined is the basis of the Active framework. When designing an Active-based system, both the domain of the application and the associated processing rules can be expressed in a single unified view.

1.3.3 Design of Active-based methods

Based on the implementation of the Active framework, a set of methodologies needs to be implemented to create *mixed-initiative* applications. Mixed-initiative systems combine human and computer-based reasoning to accomplish complex tasks. To create such intelligent assistant applications, we identified a set of AI-based techniques to be implemented with Active :

- *Language Processing.* Natural language processing is in charge of gathering information coming from sensors, match it with the application domain to produce a command or plan to be executed by the system. Our goal is to provide an Active-based technique that allows programmers to model an application domain and associated processing constrains. This stage also features fusion of modalities, where for instance, users can express commands by a combination of voice and gestures. Language processing

also includes reference resolution and disambiguation. For instance, reference resolutions would automatically resolve a city name if a zipcode (postal code) is sensed. Disambiguation should provide a set of heuristics to resolve cases where multiple parsing options have the same weight.

- *Process Management.* Process modeling consists of defining a sequence of actions to perform a complex command, typically produced at the language processing level. Process definitions represent finite state machines associated with a set of variables. Multiple instances of a same process can be running at the same type, for instance to support multiple users running simultaneous dialogs with multiple instances of an assistant.

- *Service Management.* Dynamic service brokering uses agent-based technologies to provide on-the-fly selection of resources. For instance, notification messages are sent to users through the most appropriate modality (email, instant messenger or short message) given the current user location, preferences or any other parameters such as the priority of the message to deliver.

1.3.4 Application design

An Active-based application (see figure 1.2) consists of a set of loosely coupled services working with one or more Active Ontologies in charge of core reasoning tasks. A service oriented approach provides multiple advantages. 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), data source (i.e. database) and processing services (e.g. processing intensive data processing such as translation). In addition, services are generic and can therefore be reused as building blocks for multiple different Active powered applications. Finally, using a non-monolithic approach allows us to use agent-based techniques to dynamically orchestrate services.

1.4 Objectives

1.4.1 Claims

The main objective of our work is to validate two claims.

- Using Active, AI methodologies can be encapsulated to be easily and effectively applied to build assistant-like software.

- Assistant-like software can, in some domains, be more efficient and provide better experience than traditional software.

The following sections describe our plan to validate both claims.

1.4.2 Evaluate the concept of Active Ontologies

Our first objective consists of evaluating and validating the concept of Active Ontologies. The following aspects need to be evaluated:

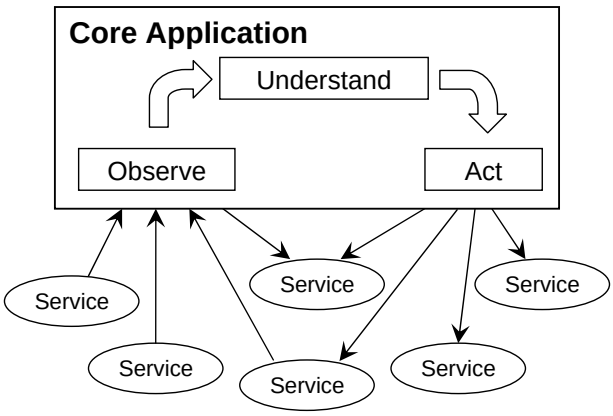


Figure 1.2: Active Application Design

- Flexibility and expressiveness. Active Ontologies need to provide a rich enough language to adequately encode a diverse set of AI methodologies, ranging from sensing and fusion, language interpretation and dialog, activity recognition, planning and scheduling, and agent-based service coordination and delegation.
- Robustness and performance. The approach and its implementation needs to provide an acceptable response time for end-to-end management of sensor inputs and user requests.

1.4.3 Design and implementation of the Active framework

To apply Active concepts to real-world situations, a software implementation needs to be designed and developed. With this objective, four tools have been developed.

- A server to host and run Active Ontologies. The server engine is in charge of hosting multiple Active Ontologies, maintaining their state and periodically evaluating their rules to execute relevant actions. To encapsulate AI techniques and provide ease of integration, the server has a plugin mechanism to easily extend its capabilities.
- An integrated development environment (IDE) to provide Active developers with a tool to create, deploy, debug and test Active Ontologies. Similarly to the runtime server, the IDE provides a plugin-based mechanism to extend its feature set.
- A management console that can introspect and control a runtime server for administrative and monitoring tasks.

- A natural language test tool accepts both interactive and batch-mode inputs to support development and regression testing of natural-language applications.

1.4.4 Design of Active-based methods

Using the tool described above, a set of programming techniques have been developed. These techniques implement the three component types defined in section 1.3.3 : language processing, process management and dynamic service selection and invocation.

1.4.5 Implementation of prototypes

To apply our approach in real-world situations, we have developed assistant-oriented applications in three domains.

- The operating room. Computers are part of the standard equipment used in modern surgery rooms to help surgeons perform complex operations that would not have been feasible before. However, limited user interfaces, combined with the specific constraints of the operating room, prevent surgeons and their staff from fully leveraging this technology. Surgeons and their staff need to interact with computers as easily and naturally as if they were just another member of the team. Our goal is to create a prototype of an intelligent assistant and have it evaluated by surgeons to assess if this approach is beneficial to their work.

- Mobile communication. Computer-phones and PDAs have become affordable, small, omnipresent and are often connected to the Internet. However, despite the availability of such rich environment, user interfaces have not been adapted to fully leverage its potential. To help with complex tasks and retrieve information, a new type of software is needed to provide more user-centric systems that act as intelligent assistants, able to interact naturally with human users and with the information environment.We intend to design and implement a system able to retrieve online information through dialog-based natural language interaction.

- Office assistant. The field of office-related tasks is popular for intelligent assistant applications; examples include an intelligent meeting planner, a smart email sorting agent or an intelligent meeting room. This is a field where an Active-based system can be compared with similar approaches.

1.4.6 Evaluation of claims

To validate our approach and verify our claims, the following evaluations have been undertaken.

- To validate the Active approach, both in terms of theory and practice, we have implemented the three prototype applications described in section 1.4.5

- To quantify how assistant-oriented software can be more effective and enjoyable than traditional software such as search engines, we placed our *Active Mobile* prototype head-to-head against the leading commercial mobile search engine, Google Mobile^{TM}.

- To demonstrate that we have been successful in lowering the cost, time, and specialized knowledge requirements for developing integrated AI assistants, we asked *"normal"* programmers to learn the Active methodology, and measured the training and development time for implementing the language understanding module of the Active Mobile prototype described in the previous evaluation.

1.5 Summary of chapters

The thesis is divided into the following chapters:

- Chapter 2 describes the state-of-the-art of research projects related to intelligent assistants. Relevant work is organized in three categories : AI frameworks, agent architectures, and intelligent assistant software.

- Chapter 3 provides the motivation and reflections that led us to design intelligent assistant applications with Active Ontologies. It starts with a definition of intelligent assistant, followed by a brief description of the research that inspired our approach. It concludes by presenting a high level introduction to the Active framework.

- Chapter 4 provides a detailed description of Active Ontologies and the basic elements of their execution. It also presents the current implementation of the Active software suite.

- Chapter 5 presents Active-based AI methodologies. It start with basic techniques, then moves towards more complex methods implementing language processing, process modeling and dynamic service management.

- Chapter 6 depicts Active-based prototypes deployed in three different application domains. The description of each prototype starts with the goal that motivated their implementation. This is followed by a constraint definition, implementation description and evaluation. Evaluations are two-fold. First, functional tests are used to verify that the implemented prototype complies with its requirements. Secondly, prototypes are used as tools to evaluate, through a population of end-users, the advantages of user-centric intelligent applications over conventional user interfaces.

- Chapter 7 provides three evaluations of the Active approach. First, we evaluate our claim that intelligent assistant software can be more effective than traditional software – this is tested by user studies comparing an Active-based mobile assistant to the leading commercial mobile search engine. Next, we evaluate our claim of making AI technologies more accessible to "regular" programmers. To do this, a number of programmers, unfamiliar with AI-technologies, are asked to create a language processing application using our tool. We measure time to completion and the

quality of the achieved systems. Finally, to validate that our methods are computationally tractable and scalable, we measure the performance of the current implementation and identify its strength and weaknesses. It provides suggestions about how one might implement a commercially viable system inspired by our work.

- Finally, chapter 8 concludes the thesis by summarizing achievements and discussing future research.

Chapter 2

Literature Review

This chapter presents relevant work related to our research. Our proposition, the creation of a platform aimed at easing the development of intelligent assistants, involves many technologies. The domain of related work to study being rather wide, we have organized this review in three sections:

- AI Frameworks : generic tools designed to be the foundation of AI-based systems.
- Agent Architectures : agent-based systems used to create user-centric applications.
- Intelligent Assistant Software : a review of systems that assist users in specific domains, and relevant work that focuses on specialized components such as language processing, dialog management or user interfaces.

2.1 AI frameworks

This first section presents frameworks designed to be the core foundation of intelligent systems. These include production rule systems, Belief-Desire-Intention (BDI) frameworks, Event-Condition-Action (ECA) platforms, and cognitive architectures of various kinds.

The first class of generic AI applications we reviewed are *production rules* systems. Before introducing the most relevant work of this area, we briefly present the main characteristics of production rule systems. The goal of such systems is to model and implement human-like reasoning. Production rule engines are typically made of a set of *rules*, a *memory* and an *inference engine*. Each rule is made out of a condition and an action; conditions are boolean expressions based on the content of the memory. The memory is a data store used to represent the current state of the system. An inference engine is in charge of evaluating the conditions of the rules. For each rule whose condition is verified, the corresponding action is executed. When fired, actions can modify the memory of the system, thus triggering further rules execution. This chain of events, called *forward chaining*, is a method used by production rule systems to implement reasoning.

In both the academic world and the industry, many projects have led to the implementation of production rule systems. In the academic world, the most

mature and utilized system is CLIPS[22]. Under development since 1985, CLIPS combines production rules, object oriented and procedural programming. The language to express conditions and actions has a Lisp-like syntax. A template definition mechanism allows the specification of data structures, where members have a name, a type and a default value. On the implementation side, CLIPS is written in C and implements the Rete[19] algorithm for fast rule evaluation. In the same family, Jess[20] is a Java based production rule system, originally inspired by CLIPS. After its initial implementation, Jess diverged from CLIPS and has now its own unique feature set. CLIPS and Jess are designed to provide general purpose production rule systems, whereas Active is designed as a programmer friendly tool to build intelligent assistant software. As very efficient and mature production rule engines, CLIPS and Jess could conceivably be used to be the processing core of the Active system. To rapidly add all necessary features, easily be modified and optimized, Active currently uses its own specific inference engine. This decision, made at the early stage of the development of Active, allowed us to quickly design, implement, experiment and validate our approach. Since Jess is implemented in Java, it is an excellent candidate to be integrated into the Active platform to provide an alternate, very efficient processing core.

In the area of AI framework, the closest work to Active may be the SOAR project[45]. Since 1983, SOAR offers an open, unified framework for building intelligent cognitive systems using a foundation based on production rules. SOAR is inspired by the principle of the *unified theory of cognition* [61], stating that a single unified set of mechanisms can be used to model all *cognition*. In this context, the term cognition (from the Latin *cognoscere*, "to know") refers to the process of reasoning, which includes knowledge, perception and awareness. The SOAR implementation features a programming language and an engine to run models (SOAR programs). The SOAR production rule engine has built in mechanisms for conflict resolution and learning. Conflict resolution is based on the principle of *sub-goaling*. When two conflicting rules are applicable, current activities are paused until the conflict is resolved using production rules. Learning is performed through *chunks,* where new production rules are created from the deliberations of the SOAR engine. Similarly to SOAR, Active can be cast as a cognitive architecture, but it is not as advanced when it comes to built-in features for learning and conflict resolution. In addition, Active is designed to explore and implement applications in a more pragmatic way. It was primarily designed to encapsulate AI techniques required to build intelligent assistants. On the other hand, SOAR is a tool to help understand how intelligence and cognition, in a broader way, can be modeled and analyzed. 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.

Another cognitive architecture aiming at implementing a unified theory of cognition is the ACT-R framework[2]. It consists of a production rule system, with two fundamental elements. First, *chunks* represent knowledge in a declarative style: *Paris in the capital of France*, *1+2 = 3*. The second type of elements are *productions*, or sets of production rules. At each evaluation cycle, the *buffer*, or working memory, is tested against the productions rules to pick candidates for execution. Only one rule can fire at each cycle and a specific *conflict resolution* technique is used to pick the winner. ACT-R is a dynamic project,

enjoying a large community of users and is used in numerous academic projects. Unfortunately, its richness, complexity and limited extensibility has prevented ACT-R from being used as the core of an open system including user interfaces and easy integration with external components. Even if *modules* can be written to connect ACT-R with the external world, allowing integration with user interfaces and data sources, the inherent complexity and LISP-like programming language confines its usage to a small domain of specialists. There are two main differences between ACT-R and Active. First, on the core reasoning side, Active is based on a more simplistic approach where multiple rules can be executed at each cycles, leaving conflict resolution under the responsibility of programmers. Secondly, Active was designed to be open and extensible. Open to easily reuse existing components through standard API, and extensible so that programmers can encapsulate and easily share their Active-based programming techniques.

Similarly, the Icarus[46] architecture aims at constructing an integrated cognitive architecture to model and control an agent in a complex physical environment. Unlike cognitive architectures presented earlier, Icarus is not based on a production rules but uses a hierarchy of probabilistic concepts and a repertoire of plans, or skills, dynamically selected to produce the most appropriate behavior. Icarus is a powerful architecture, but is primarily designed for modeling interactions of physical agents (robots) and their environment. Unlike Active, it does not provide language processing nor dynamic services invocation modules.

A second group of engines designed to implement multi-purpose reasoning systems consists of BDI-based[64] applications. BDI software architectures are agents whose behaviors are based on *beliefs*, *desires* and *intentions*. Beliefs represent the information agents assume about their environment. Beliefs can be sensed from the environment, communicated by other agents or locally inferred. The term *belief*, used instead of *knowledge*, denotes that the information is unreliable and can be erroneous. Desires represent the overall tasks the agent has to fulfill. Intentions are goals the agent has committed to and started to implement. To illustrate these concepts, let us consider simple outdoors mobile robot application. The task of the robot is to navigate from its current location to a target position. Our robot uses a GPS system to keep track of its current position and is equipped with vision sensors to detect obstacles. Its goals are to reach its target position (*reach-the-target* goal) while avoiding obstacles (*avoid-obstacle* goal). At first, the *reach-the-target* goal becomes an intention and the robot starts moving towards its target location. The current beliefs of the robot consists of its current location, constantly updated from the GPS sensor, and the goal position. Suddenly an obstacle is detected by short range vision sensors, immediately reported as a new belief. This new state upgrades the *avoid-obstacle* goal into an intention. The robot is not only trying to reach its final goal, but is also undertaking actions to avoid an obstacle by, for instance, triggering a sharp left turn. Once the sensors report no more obstacles, the *avoid-obstacle* intention becomes a goal and to robot resumes its journey towards its final goal. This examples illustrates the main characteristic of BDI agents. They are situated (are part of an environment), goal directed and reactive. Numerous software systems implement the BDI approach. Reactive planning systems have been used in the field of mobile robotics[44] and multi agent planning[82]. More recently, light-weight and programmer friendly BDI systems have become available. The Jack[8] toolkit is a commercially available tool designed to build multi-agent systems whose behavior is driven according

to BDI notions. The open-source Jam[39] system and research engine Spark[58] are both BDI engines entirely written in Java. BDI-based engines would be well suited to be the core of our research, where dynamic decisions need to be made to respond to an event. Their design is nevertheless constrain us to dynamic planning and would not be suited to implement tasks such as natural language processing or modality fusion. Our goal is to cover a broader set of AI techniques to provide, in a unified tool, all components required for building assistant-like applications.

Event if they were not originally as AI tools, *ECA* (Event-Condition-Action) systems bear many similarities with pure AI frameworks and have therefore been used as the processing core of reasoning systems. ECA rules provide conditional actions based on both event detection and data constraints. A rule is expressed as follows : on *event* if *condition* then *action*. An action is executed when a specific event occurs and a condition, historically based on a database query, is verified. ECA based engines have been used for database management[51], semantic web reasoning[63] and business process modeling [43][12].

The Numenta[32] platform attempts to create cognitive applications based on a technique inspired by the human neocortex. The system aims at solving problems such as object recognition, speech understanding, navigation or predictions. The foundation of Numenta is the Hierarchical Temporal Memory (HTM), a paradigm inspired from Baysian networks. An HTM is an upside-down tree-like structure made out of connected nodes. Terminal nodes (leaves at the bottom of the structure) are fed with raw sensory data. They analyze incoming information to detect patterns and sequences that repeat over time, to infer beliefs and report them to their parent nodes. Non-terminal nodes receive information from their children, analyze them and, in turn, report beliefs to their own parents. All nodes (leaves and non-leaves) run the exact same algorithm. Numenta is a challenging project aiming at modeling the low-level operation of the human brain to solve cognitive tasks. In contrast, the goal of Active is to provide a platform working at a higher level, providing coarser-grained AI-based components used in user-centric intelligent assistant applications.

The CALO project[5] is a large, 25-institution effort aimed at designing and deploying a personal assistant that learns and helps user with complex tasks in the office domain. CALO is an extremely heterogeneous system, involving components written in eleven different programming languages. CALO meets the requirements for which it was designed but is not a cognitive architecture tool to be used non expert programmers.

Similarly, the RADAR project[57] is focused on developing intelligent assistants 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. However, its complexity prevents programmers from rapidly get up to speed without learning about implementation details and AI concepts. Both RADAR and CALO are specialized, learning-based assistants that would be very difficult to adapt for other domain applications.

The MULTIPLATFORM testbed[34] is a generic service-oriented software framework for building dialog systems. It has been used in numerous applications ranging from interactive kiosks to mobile assistants. Although is has shown robustness and effectiveness, the system lacks some of the flexibility required to support dynamic planning and runtime reconfiguration. All data structures and messages exchanged among components are defined as XML documents at de-

sign time, and cannot be easily changed on the fly. Adding new types of services requires the application to be taken offline and redesigned, whereas we are trying to provide a more dynamic environment where services and service types can easily be added to the system.

2.2 Agent architectures

Agent architectures are software systems made out of independent intelligent entities called agents. Each agent has its own behavior, goals and representation of the world. A communication system allows agent to exchange information.

In this field, the open agent architecture [10] (OAA) introduces the powerful concept of delegated computing. Similarly to our approach, OAA systems consist of communities of services whose actions are combined to execute complex plans. Requests and plans are delegated to a facilitator in charge of orchestrating actions based on declared capabilities of agents. Thanks to its ease of deployment and clean design, OAA is used in a large number of projects. The design unifies in a single formalism the application domain knowledge, the messages exchanged among agents, the capabilities of agents and data driven events. Though very powerful, OAA does not provide a unified methodology to create intelligent systems. It rather provides a robust and flexible framework where heterogeneous elements, written in many programming languages, are turned into OAA compatible agents to form intelligent communities. The goal of Active is to not only provide an OAA inspired delegated-like computing, but also in a unified fashion, provide the language processing and plan execution components required to build intelligent assistants.

Similarly, the Retsina [74] framework is an advanced multi agent architecture to build distributed intelligent systems. It is based on four classes of agents. Interface agents that interact with users, task agents that carry out plans, information retrieval agents and middle agents to help match agents that request services with agents that provide services. Though very efficient in producing independent reactive behavior, Retsina would not be suited as a unified methodology to implement basic AI components such as natural language processors or multi-modal fusion engines. In addition the design of Retsina uses different formalisms for communication, domain representation and reasoning technique. In contrast, our aim is to use the same formalism for all intelligent assistant aspects.

Another advanced agent architecture is the CoABS[41] grid, a DARPA funded project whose goal consists of creating a grid of heterogeneous agents. The middleware offers registration and lookup for dynamic, on the fly discovery of services. It also provides high level layers to encapsulate the complexity of data marshaling and security aspects of communications. In addition, mobile applications are supported, where entire agents can migrate from one host to another for optimized execution. CoABS is a powerful Java-based middleware but, unlike OAA or Retsina, does not provide any means to orchestrate agents on the grid. For basic inter-agent communications, our work will rather rely on open standards such as *SOAP* or *REST* and implement the reasoning of discovery and selection of services at a separated layer based on the delegation mechanism introduced by OAA.

2.3 Intelligent assistants

This section presents relevant work directly applied to the field of intelligent assistant systems. Its first part presented toolkits and components designed to be used as building blocks for assistant-like applications. The second section presents relevant examples of actual intelligent application prototypes.

2.3.1 Toolkits

This section presents relevant architectures designed specifically to implement intelligent assistants. We also mention projects that provide important components to be integrated into the design of interactive assistant systems, such as specialized language processing systems or dialog managers.

It would not be fair to start this section without mentioning Eliza [81], the 1966 intelligent assistant-like system aimed at emulating a psychotherapist-style dialog. The system uses a combination of keywords and rules to create an intelligent behavior mostly by paraphrasing user inputs. As the first attempt to create intelligent human-computer dialog, Eliza is an important milestone of the field of intelligent assistants. Since the Eliza times, computers and their environments have dramatically changed leading to many projects aiming at providing intelligent and natural computer-human dialogs.

The SmartKom[79] program is a large research project whose goal is to build systems that interact with humans through natural modalities such as speech and gestures. Research focus is on multimodality (disambiguation and fusion) and dialog management. SmartKom applications are based on a three-tier architecture : multimodal sensors and effectors integration, dialog management and backend connections. The core of the system consists of several independent programs, written a different programming languages, running on multiple computers that communicate through XML-based messages. The approach has been validated through three types of applications (mobile, kiosk-like systems and home-office assistant) that have been positively evaluated. Due to its highly heterogeneous and distributed nature, though very powerful, Smartkom could not easily be used as a lightweight and unified framework used by a small team of engineers to quickly build simple intelligent-assistant applications.

The Collagen[65] toolkit proposes to build autonomous agents that interact with human users by collaborating through shared tasks. The approach is generic, easy to learn, natural to use and does not require formal language parsing. Collagen interacts with the user through a dynamic user interface, offering options limited to the current state of the dialog. Since the user cannot freely express commands, there is no need for language processing. Constrained applications are by nature more robust than more open systems where users can express any command at any point. They are nevertheless less intuitive and sometimes tedious to use because complex operations are restricted to a predefined logical path. A common example of such interface is a phone-based system, where users are prompted about a limited set of options at each step of the dialog. Collagen also differs from our approach in its specialization to only solve the dialog interaction where we try to provide a broader tool to construct intelligent assistant applications.

TRIPS[17] is an AI integrated system providing an end-to-end framework to build mixed-initiative problem solving assistants. The system is organized

around three main components. *Modality processing:* provides user input gathering (including speech recognition, spelling correction) and user interface output (including graphics and speech synthesis). *Dialog management:* the core of the system, manages ongoing conversations, contexts and coordinates specialized reasoners through its plan solving manager. *Specialized reasoners:* specialized application specific reasoning components. Designed to help users in a crisis context and planning, prototypes of assistants based on TRIPS have been designed in the field of rail-based freight delivery and crisis management (evacuation of people and airlifting). TRIPS is a robust and powerful platform that has been used and validated in different applications. However, it is specialized and optimized for crisis management, whereas Active aims at being more generic and used in a broader range of application domains.

The domain *pervasive computing*[71] is driving relevant work in the domain of user-centric applications. The field envisions smart spaces, where our habitat (i.e. homes, cars or work places) becomes aware of our presence and activities to help and communicate with us as we go about our tasks. Computing moves away from the console-keyboard-mouse paradigm to become invisible, seamlessly integrated into our environment. In this context, the Oxygen project[68] from MIT articulates this vision around multiple axis. Devices, mostly handheld, allow users to be connected with the Oxygen network. An intelligent network allows Oxygen compliant devices and sensors to discover each other to exchange relevant information. Perceptual technologies (speech and vision) and software components complete the framework. Oxygen is a thriving think-tank, where technologies and ideas are brewed together to create exciting user-centered applications. There is nevertheless no attempts to find a unified approach to solve the numerous challenges involved in creating intelligent assistants. Our goal is far less ambitious in the scale of applications to create. It consists of finding the common denominator of some aspects required for user-centric applications and to implement them in an easy-to-use, unified and coherent toolkit.

Similarly, the XCM[75] approach proposes to model coordination of pervasive computing applications around four concepts: entities, environment, rules and apis. As its representation, the model uses the OWL language, developed for the Semantic Web to represent ontologies. Alike the Active framework, XCM proposes a blend between rules (called *social laws*) and ontologies to drive the behavior of a community of components working for a user. The focus and strength of XCM are on coordination of services, whereas Active proposes to cover a shallower, but broader set of intelligent assistant applications components. Active is not as powerful and accomplished in terms of coordination, but aims to be a tool usable for the entire design and implementation of pervasive applications. It would be an fruitful exercise to implement the powerful XCM philosophy with the Active framework, as another AI-based technique supported by our framework.

Finally, the OneWorld[24] project aims at providing developers with a framework to build applications in highly dynamic environments. The service oriented framework is based on service discovery, software migration and universal access to information. Three main principles drive the effort. First, services need to expose change, mainly failure to find a specific service, so that programmers can easily provide effective exception policies. Secondly, service actions should be composed dynamically, hard-coding or tight coupling of resources should be avoided. Finally, data representation should be separated from functionality,

so that generic pieces of data can be routed to different services for processing. One.World has inspired our work and provides a powerful framework for distributed systems designed to be deployed on highly dynamic environments. Our approach brings similar ideas in the senses of delegation, where services are not only dynamically composed, but also selected on the fly. Selection is not only based on service availability and advertised functionality, but also on the current application context such as user location and preferences.

2.3.2 Applications for a specific domain

In this section we describe relevant prototypes of mixed-initiative assistants designed for specific application domains.

Smart Sight[86] is an assistant that provides help to a tourist through a multimodal interaction based on speech, vision, voice and gestures. At the user interface level, a multimodality manager gathers user inputs through a set of recognizers (speech, handwriting and gestures). Then, at a second processing stage, a parser combined with a semantic analyzer converts inputs into structures to be further processed by a dialog manager. The dialog manager deals with *dialog acts*, modeled as production rules used as webform-like interaction model. When all attributes of the form are specified, the rule fires and its action is delegated to a specialized procedure in charge of undertaking the associated actions. This system is interesting because it shows some similarities with Active-based systems. First, it makes use of an ontological representation of the application domain to construct semantic information out of the raw results from a parser. It also uses production rules as the processing mechanism for the dialog manager. There are nevertheless two major differences with our approach. First, since Active unifies all stages of processing through the same unique formalism, there is no need to convert raw parsing information into the application domain. In our approach, the application domain is the parser itself. Secondly, Smart Sight does not have the notion of a dynamic service selection, therefore interactions among components are not flexible and cannot easily be reconfigured.

The Personal Satellite Assistant (PSA)[15] project explores human-robot interaction by creating an autonomous robot designed to assist astronauts in mission related tasks. In addition to the robotic challenges, user interaction with PSA has to be hands-free (voice based) and provide dialog management. A functioning dialog manager has been implemented specifically for the PSA project, which could not be reused as a generic tool for build intelligent assistant software.

The Smart Personal Assistant[85] application is a system that provides assistance for office-like applications over a pocket computer. It features a speech and point-click interface to access both calendar and email systems. This work is relevant to our approach in at least two ways. First, it is built around a service-based architecture, where components are dynamically orchestrated by a central manager. Secondly, the manager consists of a BDI engine where user dialog is modeled as a collection of plans. The application nevertheless differs from our vision as being a collection components designed in different programming languages, therefore difficult to integrate and debug.

2.4 Conclusion

This chapter has presented relevant work in the domain of intelligent assistant software design. Most research efforts presented have inspired our vision to create a unified framework for building intelligent assistants. Each project is focusing on a specific aspect of the problem: modeling intelligent behaviors, provide high-level natural user interfaces or dynamically select and invoke resources. The review also shows that complete end-to-end intelligent assistant applications have been designed, implemented and validated, demonstrating the tremendous potential of this most needed type of software systems. The design and implementation of these prototypes and products require significant software integration and specialist collaboration. Our goal is to lower the bar to build intelligent assistants by encapsulating and fusing the best of AI techniques into a unified toolset to create innovative techniques and ideas to implement user-centric software.

Chapter 3

Theory of Operation

This chapter presents the high-level design and motivations behind our approach to provide a unified framework for building intelligent assistant applications. First, it introduces the concept of intelligent assistant as defined by the literature. Then, it presents relevant AI theories and how they influenced our work. Based on this review, it describes our view on the definition of intelligent assistants and how it is cast into a unified framework solution. Finally, it introduces the Active framework, our software instantiation of this vision.

3.1 How intelligent assistants work

This section presents our definition of the concept of intelligent assistant. First, we present the environment that justifies the use of intelligent assistants and describe a simple concrete example. We then provide our description of what an intelligent assistant should be and compare it with other definitions found in the literature.

3.1.1 Motivation and problem space

Computer systems keep growing in complexity, processing power and interconnectivity. To leverage this rich environment and better assist users, a new type of intelligent assistant software is necessary. Intelligent assistants are software systems that can communicate with humans, understand the situation by mapping input senses into a model, act to produce useful behavior (i.e retrieving relevant information), and then interact through an appropriate rendering of communicative content.

As an example, let us imagine a situation where a visitor is looking for information about a city. Instead of navigating through multiple web sites, or struggling with a cellphone embedded browser, it would be easier to simply send a message (email or short message) to a personal assistant asking, in plain English, *"find me a Chinese restaurant tonight in San Francisco"*. The answer would then come back, as a text message, with a list of restaurants, or as a question such as *"which area of San Francisco?"*. Through a dialog-based thread of asynchronous messages, the assistant helps our visitor to focus on the purpose of the visit by delegating tedious tasks to an intelligent automated system.

The goal is to delegate tasks to computers instead of using them as tools in a direct manipulation fashion, allowing users to deal with a large amount of information and focus on important activities. There is still a fierce debate about the relevance of this approach as the future of all computer human interactions[72]. Critics argue that intelligent assistants can be bring confusion, be too intrusive, or prevent users from controlling all aspects of their computer-based activities. In many cases direct manipulation is still the most effective way to control a software system, therefore adding a caution that intelligent assistance needs to be carefully planned. Our goal is to focus on application domains where intelligent assistants are at the core of user-centric applications and not deployed as add-ons to an existing direct manipulation-based system (e.g., the highly controversial Microsoft's word paper clip). We have identified specific application domains where assistants are the most relevant, if not necessary.

- *Intelligent homes.* Intelligent homes are a natural application field for pervasive computing applications. Instrumented spaces are aware of local user activities and intentions to provide relevant contextual help when requested. Literature shows that this field goes beyond pure entertainment for luxurious home-of-tomorrow projects, but also provides realistic and most needed assistance to senior citizens, physically challenged people and patients staying at home[73, 33].

- *Information and task workload.* As described in her seminal paper, Maes[49] describes the *raison d'être* of intelligent assistant applications as their ability to help us manage the information overload brought by the complexity of our daily tasks. It includes simple tasks such as automatic email filtering and classification to more complex plans such as scheduling meetings or organizing a trip. An emerging aspect of information retrieval where intelligent assistants can provide support is search engine querying. Broder[6] shows that least 23.8% of search queries are *transactional*. A transactional query carries the user intention to perform complex web-mediated activity, such as organizing an event, purchasing an item or booking a trip. Providing the ability to express a sequence of queries in plain English in a dialog-based fashion to extract complex information from the web, users can delegate low-level tedious browsing tasks to focus on the overall activity at a higher level.

- *Challenging environments.* In some domains, the complexity of tasks could not be achieved by humans without assistance. For instance, pilots could not fly modern airplanes without active automatic electronic and mechanical assistance from the aircraft. Taken at the extreme, the domain of air combat led to highly sophisticated pieces of software providing contextual and relevant support to pilots[3]. The medical field is another example, where computers are now part of the standard equipment available in modern surgery rooms. When performing highly complex interventions, robotic and computer-based systems extend the normal capabilities of surgeons. They offer higher accuracy, better view of the operating field and the ability to perform complex moves such as cutting and stitching in areas difficult to reach. In the context of this work, we will explore if assistant-like applications could mediate interactions between surgeons and to the operating room back-end systems.

3.1.2 Intelligent assistants characteristics

This section reviews relevant work in the field and extracts the main characteristics of intelligent assistants.

First, let us clarify some terms used in the field. The literature uses different expressions such as : *personal assistant*, *interface agent*, *intelligent interface agent* or *intelligent assistant* almost interchangeably. In this document, we will use the term *intelligent assistant* or simply *assistant*.

At the highest level, relevant literature [49, 53, 48] agrees that intelligent assistants are software systems to whom some tasks can be delegated so that users can reduce their workload and focus on the most relevant aspects of their activities to gain in productivity.

More specifically, in his review of the field, Middleton [53] defines interface agents as the field where AI meets with *HCI*. He defines the domain along three axis:

- *Knowing the user.* Knowing the user starts by extracting goals and intentions from any perceived input modalities. The assistant also adapts its behavior and plans to changing goals. Finally, it learns about user preferences to provide the most appropriate response.
- *Interacting with the user.* Interaction with the user unfolds on three aspects. First, the granularity of delegation, where the assistant provides help at the right level. Secondly, the type for interaction has to be carefully designed. This is where modalities (i.e. speech, mouse/keyboard, vision, text messages) and the metaphor (i.e dialog box, desktop, animated character) are defined. Finally, the usability of the system needs to be adapted to the level of expertise of targeted users.
- *Competence in helping the user.* The assistant uses strategies to, sometimes autonomously, select and performs tasks on behalf of the user.

The definition provided by Lieberman and Selker[48] is slightly different. They start from the three-word term *intelligent interface agent* to define each word separately. *Intelligent* refers to any systems that exhibit some level of human-like intelligence, *interface* describes any medium used to communicate with a person and *agent* refers to a system that provides assistance. In addition, intelligent interface agents exhibit the following characteristics:

- *Initiative*: Unlike direct-manipulation interfaces, assistants take initiatives based on their observations of users and their environment.
- *Roles*: An *assistant* is a system that actually undertakes tasks on behalf of the user. An *adviser* on the other hand, would not perform any action but advise the user on what should be done.
- *Personalization*: User interfaces provided by intelligent interface agents should be dynamic enough to adapt on different users.
- *User Modeling*: Intelligent assistants should have a model the tasks and the application domain in which they provide help.
- *Trust*: The agent should provide a consistent, error-free behavior for users to build trust and actually use it.

- *Feedback*: The agent should always clearly explain its actions.
- *Inscrutability*: The agent should learn over time from its interactions to provide better, more appropriate and more effective support.
- *Anthropomorphization*: A metaphor often used as the user interface of assistant agents is based on human-like animated characters. Studies have not clearly shown[9] whether anthropomorphization provides better communication with users.
- *Cognitive Style*: Difficult notion to measure, it defines how appropriate is the help provided by the assistant. It is similar to the granularity of delegation introduces earlier, where agents have to adapt the amount of help and suggestions they produce to avoid being too intrusive and defeat their very purpose.

3.1.3 Relevant theories and approaches

In addition to the field of intelligent assistants, numerous ideas and theories have inspired our work and shaped the design of the Active framework. This section summarizes them and outlines how they contributed to our solution.

Cognitive architectures

The field of *cognitive architectures* is a fertile ground where many of the AI techniques used in intelligent assistant design have originated. Cognitive architectures aim at modeling human-like reasoning based on knowledge representation. Most cognitive architectures [45, 63, 2, 20], tools, and applications [5] rely on production rules to encode reasoning. A non-procedural event-based programming model is suitable the field intelligent assistants, where inputs to trigger actions not only come from users but also as asynchronous events sensed within the environment where the agent is deployed.

Knowledge representation

In the field of computer science, and AI in particular, *knowledge representation* consists of modeling, storing and exposing knowledge about a domain. The knowledge store can then be use by processing components to learn (by adding information to the knowledge store) and reason. *Ontologies* are a popular and powerful tool to represent knowledge about a domain. An ontology is made out of *concepts* augmented by *attributes* connected by *relationships* to describe data models.

Agent technologies

Marvin Minsky's *Society of Minds*[55] has influenced directions in AI research for more than twenty-five years. The theory is built on the idea that communities of simple very specialized *agents* work together to create a collective high-level intelligent behavior. Agents are simple processing components that can communicate and be connected to create larger, more complex systems. Based on this architecture, Minsky built a theory, explaining how to model systems for most AI fields. Active has been influenced by this theory at three

levels, ranging from the very core of the Active system, to its high level design model.

Neuroscience inspired approaches

Neuroscience discoveries about our brain are used to model intelligence. Base on neocortex research, Hawkins[32] developed the compelling *hierarchical temporal memory* (HTM) theory. He claims that our brain has a single unique algorithm which consists of recognizing patterns that repeat over time. Pattern recognition modules are independent and connected in an side down tree-like network. Bottom leaves produce signals about patterns to higher level nodes, also trying to detect patterns. As the information flows up the tree, it becomes more abstract to represent high level invariant concepts. Using memory, a feedback mechanism flowing down from tree top, informs and tunes pattern matchers about what signal should be expected next. This provides an integrated model of sensing, interpretation, anticipation, and action, attributes we feel are essential for an intelligent assistant architecture.

Active Ontologies

The fundamental tool of our research, *Active Ontologies*, have been inspired and inherited from all the projects mentioned so far.

First, the fusion of production rule engines and ontologies to represent an application domain is the foundation of Active Ontologies. An Active Ontology is an ontology enhanced with processing elements. Whereas a conventional ontology is a data structure, defined as a formal representation for domain knowledge, an Active Ontology is a processing formalism where distinct processing elements are arranged according to ontology notions; it is an execution environment. Concepts of an Active Ontology are augmented with production rules, and its relationships become communication channels. This approach allows AI developers to model an application domain and directly apply any required processing in a seamless and unified environment. The following chapters of this document explain how Active Ontologies are used as the unique programming technique to model and implement major components required to build intelligent assistants.

Active-based applications also leverage agent techniques at three levels (see figure 3.1). Active-based systems consist of a core (running a collection of Active Ontologies) and a community of heterogeneous loosely coupled services. Each component of this service oriented architecture can be considered as an agent, part of a larger community. At a second level, the Active server runs one or more specialized Active Ontologies, dedicated to specific tasks. For instance, one may be in charge of language processing, one executing complex processes while a third one dynamically orchestrates services. Similarly, at this level of design, one can consider each Active Ontology as an agent, working within a community of siblings to provide a collective behavior. Finally, at the very core of the Active system, Active Ontologies are enhanced ontologies made out of concepts and relationships. Concepts are extended with processing and relationships used as communication channels. In this context, each concept can be thought as a very specialized agent, that communicates with its counterparts to form a community able to collaborate on complex complex tasks.

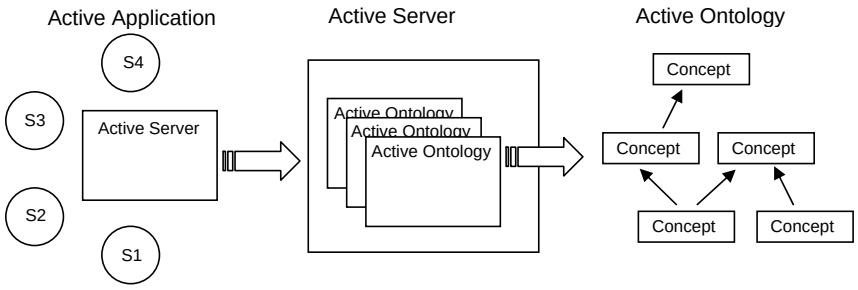


Figure 3.1: Agent-based influence

Finally, programming using Active Ontologies bears similarities with neuroscience inspired techniques. For instance, the technique developed to perform language processing with Active Ontologies (see section 5.2.3 on page 71) uses an tree-like ontological structure to represent the domain of an application. Each leaf is instrumented with specialized rules to analyze incoming words to give them a potential semantic value. For instance, a node is charge of detecting cities, would react to the word *Paris*. As nodes react, they use ontology relationships to communicate their findings to the parents for further processing. At the ontology definition level, a *city* node belongs to an *address*. As processing takes place, the *city* node reports its findings and semantic claims to the *address* node for further processing. This approach is aligned with research related to brain studies.

This combination of flavors gives Active Ontologies a unique feature set and large potential to implement AI-based systems.

3.1.4 Our intelligent assistant definition

Based on our background, observations, and the literature reviewed in this chapter, we present and contrast our attempt to define an intelligent assistant. Although similar in its overall characteristics, our view on intelligent agents is less abstract and more practical than the work reviewed in previous paragraphs. As shown by McTear[52] and Höök[37], a practical approach when building intelligent interfaces is crucial. Our more pragmatic approach takes implementation and software design into consideration from its root by articulating its definition around an *intelligent agent control loop* (see figure *3.2*) . A control loop design allows an intelligent assistant to constantly check its actions against its environment to rapidly adapt its behavior to new sensed signals. As a consequence, when signals are captured from the environment, the intelligent assistant interprets them, changes the course of its actions and, in turn, can modify its environment.

The main characteristics the intelligent agent control loop can be decomposed into four entities. The intelligent assistant, is able to *observe*, *understand*, *learn* and *act* within the *environment* in which it is deployed.

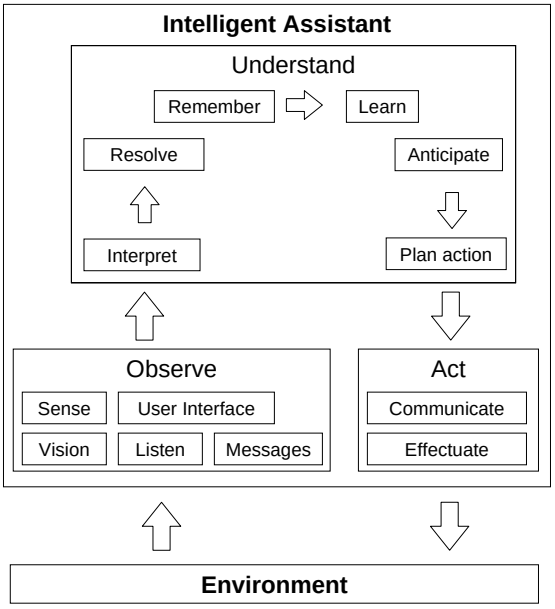


Figure 3.2: Intelligent Assistant Control Loop

Sense and observe

To provide contextual relevant support, the intelligent assistant needs to *observe* to sense its environment. We consider a global observation that not only involves listening to users, but also capture any signal (e.g. ambient temperature) that brings useful information. Intelligent assistants should support *multimodal fusion* of input signals. For instance, a user can issue a vocal command "*Show me a photo of this area*" while pointing somewhere on a map. Intelligent assistant systems should easily support heterogeneous inputs signal ranging from typed text in a conventional user interface, *utterances* produced by speech or gesture recognizers to asynchronous messages such as emails or instant messages.

Understand

Once signals have been sensed, the system has to *understand* their meaning. Understanding is at the core of intelligent assistants, this stage can be modeled into several sub tasks:

- *Interpretation.* The interpretation step processes independent input signals to produce a valid structure used for processing. This is typically the where *language processing* and *multimodal fusion* take place.
- *Resolution.* Once raw commands have been generated at the interpretation level, they need to be resolved. Resolution involves *disambiguation* and *reference resolution.* Users can issue ambiguous commands such as "*find Chinese food in Davis*", where there are multiple cities named Davis in the USA, and multiple provider types (e.g. restaurants, markets) that

sell Chinese food. The first step is to have mechanisms in place to detect ambiguities. Once an ambiguity has been found, it can either be solved automatically using user preferences (e.g., which Davis is closest to where the user lives), heuristics (e.g. choose the Davis with the largest population), or the context history (e.g. which one makes the most sense given what the user has just been talking about); or manually, by asking the user to explicitly specify which Davis to consider. Reference resolution may involve multimodal or context-based commands – if a user issues a multimodal command such as *"show me a photo of this"* while pointing on a map, the value *"this"* should be resolved with the object indicated by the gesture.

- *Validation.* At the validation stage, the assistant ensures that it has enough coherent information to undertake tasks and actions on behalf of the user. If not, a similar approach the resolution step is deployed. Using contextual information, the assistant can try to validate the information automatically. For instance, if a zip code is required for further processing, a database look up may be used to find the zip code of Davis, California. If no automatic solution is found, and the missing information is required to make forward progress on the task, the assistant will loop back to the user asking for more detailed information.

- *Learning.* Intelligent assistants need to provide a notion a learning. Learning can be used to infer user preferences out of their activities and answers. For instance, if a mobile user is constantly asking about the closest Italian restaurant, the Italian attribute may become a default option when he or she requests for places to eat.

- *Anticipation.* Based on their knowledge, either learned or a priori known, intelligent assistants need to anticipate actions. Anticipation is used to help users in their activities by anticipating options. For instance, let us consider a mobile user requesting the status of a flight. If the flight is delayed, the assistant can automatically anticipate a stay over and start looking for hotels. Anticipation can not only be a comfort and guide for the user, but is also helpful for performance, using speculative retrieval and caching of results that the user may want next.

- *Action planning.* Once user requests have been sensed, interpreted, resolved and validated, the assistant has enough information to actually launch actions. As assistant should have the processing capabilities to reason about and execute both simple action responses in reaction to the current situation, as well as manage complex automation flows based on long lasting plans and goals.

Actions

As a result of task execution, the intelligent assistant will act on its environment. Actions may involve communication with users to deliver results, to ask for more information or to communicate warning in anticipation to potential issues. Actions may also involve changing the world through a variety of effectors, ranging from manipulating physical devices, controlling software applications,

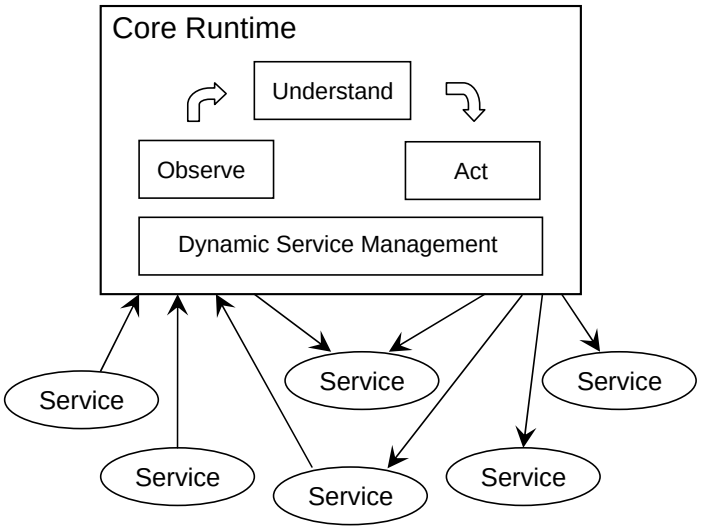


Figure 3.3: Intelligent Assistant Software Architecture

or tasking remote services. For instance, an assistant providing help at home could dim lights, draw curtains or adjust ambient temperature. An assistant providing support in the work area, could be the intelligent interface between human users and their information sources such as email, web sites, and files.

To provide the flexibility required by dynamic environments, actions undertaken by intelligent assistants are preformed through *service delegation*. Delegation consists of selecting, at runtime, the type of service and specific service provider to use. Selection of service is performed by a *broker*, that bases its decision on information coming from the current context, user preferences or advisers. For instance, a user could issue the following command *"Send this message to Bob"*. Assuming the message is known, the broker will select among notification services which is the most appropriate to deliver the message. Based on service availability, priority and type of the message, or even the location of the Bob, the broker will pick the best modality (i.e. email, instant message, voice mail) to inform Bob.

Having practical intelligent assistant software design in mind from the beginning, effective delegation calls for a Service Oriented Architecture *(SOA)*[7] software approach. This is especially true in the domain of pervasive computing where service-oriented frameworks at the core of many research efforts [71, 25, 24, 68]. We therefore envision intelligent assistant applications as a core running one or more Active Ontologies, processing information coming from their environment through **sensor** services and acting by orchestrating a set of **actor** services.

The environment

The last component of the intelligent control loop is *the Environment.* This is the domain in which the intelligent application is deployed. The main actors of the environment are human users to whom the assistant provides help. In addition, the environment is thought as a larger space, where non user-created information is collected to provide the assistant with a full picture its surroundings. Intelligent spaces, at the core of pervasive computing, are instrumented areas with sensors and computing power. Computing moves away from the console-keyboard-mouse to become invisible, seamlessly integrated into our environment.

3.2 A unified integrated framework

Based on the work presented so far in this chapter, this section presents the motivation, decisions and advantages of our vision, which consists of providing a unified toolkit and associated techniques to build intelligent assistant applications. It exposes the main characteristics of a unified framework to build such software systems and shows how the Active framework fits into this model as is actual software implementation.

3.2.1 Motivation

There are multiple reasons and advantages to use a unified integrated framework to build intelligent assistants.

- Experiment with new approaches. A unified toolkit can be used as a testbed to experiment with innovative approaches to solve AI problems. For instance, this document presents a new approach to language processing that unifies, in one single application, the definition of the application and the processing of incoming utterances (see section 5.2.3 on page 71).

- Horizontal design. 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 data structures. By implementing most of these components in a seamless and unified framework, information and domain definitions can be shared across all tiers of the application. For instance, we demonstrate later in this document how data structures defined at the language processing level can be directly be used for service definition, registration and discovery.

In addition to design improvements, practical advantages arise from this approach:

- Smaller teams. Existing complex AI-based systems aimed at assisting users[5] are very advanced and powerful, but require large teams of specialists. Using a unified toolkit along with a set of methodologies, a small

team (ultimately a single software developer) could rapidly design an intelligent assistant application by applying many of the best AI techniques into their project. Programmers could pick among existing Active-based techniques to provide the required components for their application. For instance, we present in section 5.2 on page 64 two language processing methods implemented with Active.

- Share, create and compare AI techniques. A unified toolkit allows programmers to encapsulate and share existing and original techniques more easily. Researchers can use off-the-shelf components to mix them with their original research. Sharing of compatible components fosters innovation. In addition, components based on well known techniques can be directly swapped in and out to be compared in a unique test harness with original approaches.

- Easier to debug, test and deploy. A unified framework can be used as a testbed to design, implement, evaluate and compare new intelligent assistant systems. A unified architecture leads to controlled and easier to debug applications and prototypes. Tracking events and data flow across components running within a seamless framework allows developers to quickly analyze problems and isolate functional problems. Finally, easier and more robust applications can potentially lead to productized and supported AI-based systems.

3.2.2 The Active framework

This section provides a high level introduction to the Active framework, our implementation of a unified toolkit to ease the development of intelligent assistant applications. Pointers are given to chapters and sections where more in-depth descriptions of Active components are provided. The Active framework consists of a software toolkit and a set of methods describing how to implement AI-based components.

Software toolkit

The software toolkit provides development, test and runtime tools. Development tools consist of an IDE, the Active Editor, where programmers can easily design, implement and test their components. The IDE is articulated around a graphical drawing area, where developers can model Active Ontologies by drag-and-drop creation of concepts and relationships. Specialized integrated editors are used to write and associate processing elements over the ontology definition. A runtime server, the Active Server, hosts and runs Active Ontologies. Active Ontologies are deployed on the server that manages the execution, maintains their state and handles all communications with the environment. Finally, a management console allows Active developers to test values and debug running deployed Active Ontologies.

The software suite is open and extensible. An extensibility mechanism allows researchers and programmers to collaborate by encapsulating AI-techniques into standardized plugins. Both the Active Editor and the Active server provide extension mechanisms for developers to create plugins.

The implementation of the Active software described in section 4.3 and the *Active User's guide* provided in annex.

Techniques and methods

In addition to the software toolkit, the Active framework provides a set of techniques to provide AI-based components. All of these methods are implemented as one or more Active Ontologies and server extensions. We have identified and implemented the following modules to be used as components of intelligent assistant applications.

- Natural language processing. Language processing consists of gathering and analyzing user signals over time to create a command, or a plan to be executed by the system. Gathering of user inputs needs to support different modalities, that can be fused to create user command. Language processing also provides disambiguation and validation techniques. Two Active-based approaches implementing language processing are presented in section 5.2 on page 64.

- Process modeling. Once the language processing stage has generated a command to be executed, the assistant will launch a sequence of actions to complete the task expressed by the user. Process models can be represented as finite state machines consisting of nodes and transitions. Events trigger the transition from one node to another. Events can be generated by the process itself (timeout, conditional branches or event generation) or externally posted from remote applications. Each node contains an action, executed when the state of the node becomes active. Section 5.4 on page 109 presents how process modeling and execution can be implemented with Active.

- Services management. As previously explained, intelligent assistant applications benefit from being built around communities of loosely coupled services. This design allows for ease of integration, pervasive computing and dynamic service selection. Sensor components, such as speech recognizers or vision systems, are likely to be integrated into intelligent assistants. Exposing sensors as loosely coupled services provides flexibility, reconfiguration and standardized integration. In addition, mobile users may temporarily loose the ability to connect with some services, replace them with their equivalents depending on their location. In addition to these practical advantages, service-based architectures allow for delegation, where service providers can be picked on the fly, based on the current context. Section 5.3 on page 96 presents how Active is used to register, discover, select and invoke loosely coupled services.

- Context management. Context management is horizontal, all previously mentioned items require context management. A language-driven dialog consists of multiple utterances, building a conversational context over time. For instance, one can say : *"find movies in San Francisco tonight"* and then, in a subsequent utterance express : "*get comedies only*". The context keeps contextual information, such as location and time, across utterances. Process execution also requires context, where branches and

action will be triggered based on the context. Finally, dynamic service selection is also based on the current application context.

Software design

As previously described, intelligent assistant applications should be designed around service oriented architectures. Figure 3.4 shows the software application design to build Active-powered applications. An Active Server runs one or more Active Ontologies, each implementing a critical component of the system (language processing, service management or process modeling). A community of loosely coupled services provide sensors and actors to connect the intelligent assistant to its environment. Active developers use the Active Editor to write and deploy Active Ontologies onto the Active Server. Finally, the Active Console provides monitoring and runtime management of the system.

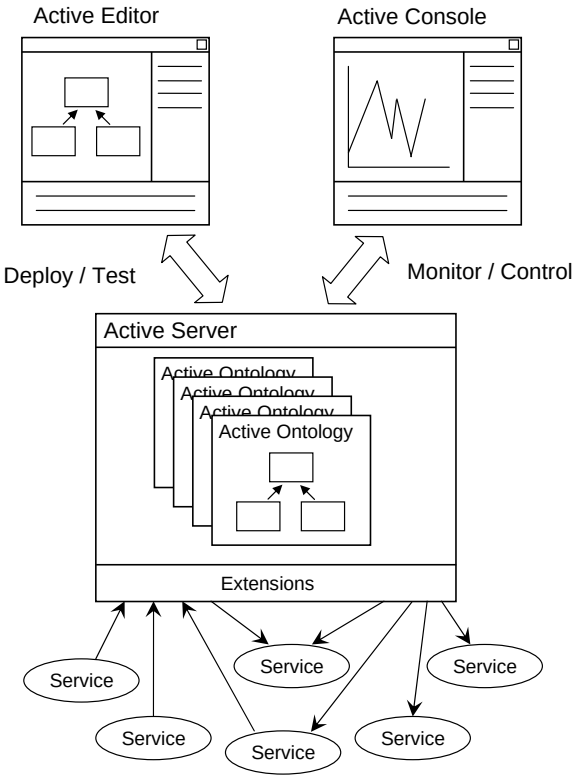


Figure 3.4: Active-based Application Software Architecture

3.3 Conclusion

In this chapter we presented the background and the motivations behind our vision. First, based on our experience, current research and technical constraints, we have provided a definition of intelligent assistant systems by enumerating a list of characteristics and discussing our software architecture. We have then shown that our work has been inspired by research ranging from fundamental work such as Marvin Minsky's *Society of Minds*[55] to more recent approaches, such as the *hierarchical temporal memory*[32] theory proposed by Jeff Hawkins. Having implementation and software architecture in mind, we have also reviewed more pragmatic approaches such as service oriented architectures and pervasive computing frameworks.

We have then described our proposition to create a unified framework to build intelligent assistants by exposing its main characteristics and advantages. Advantages includes lowering the bar to build AI-based systems, allowing smaller teams to work on intelligent assistants, providing a testbed to researcher for innovation, comparison and collaboration. Additional practical advantages are easier to implement, test and debug applications, with better potential for commercialization.

Finally, as an implementation of this vision, the Active framework has been introduced. The Active framework consists of a software suite including an IDE, a runtime engine and a management console. Both the IDE and the runtime engine provide SDK's for extensibility, encapsulation and sharing of reusable components. Active also consists of a repertoire of methods to implement AI-based modules required for intelligent assistant applications. The chapter enumerates a list of such modules providing dialog-based language processing, dynamic services registration and selection, process modeling and execution and context management.

The following chapters present the Active framework in more detail, starting from its basic characteristics and then describing existing methodologies and prototypes.

Chapter 4

Active Kernel

This chapter introduces the Active framework. It starts by presenting Active Ontologies, the foundation of our platform. Then, a detailed description of the Active processing rule engine is given, supported by a series of examples. Finally, a description of the current Active software suite is included.

4.1 Active Ontologies

Active Ontologies unify application domain modeling and processing. Active Ontologies are based on conventional ontologies and can therefore be used to model knowledge about an application domain. Active Ontologies also features processing elements so that a layer of processing can be added over the definition. Active Ontologies are both a data structure and an execution environment.

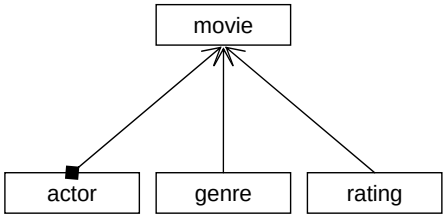


Figure 4.1: Simple Ontology

As previously introduced, an Active Ontology is an extension of a conventional ontology. Therefore, Active Ontologies inherits the basic features of classic ontologies based on *concepts* and *relationships* to represent some knowledge about a domain. In an Active Ontology, concepts have a unique name and can be connected with relationships. Relationships are oriented connectors that relate two concepts, a source and a destination. Each relationship has a *type* and a set of *attributes*.

In the example shown in figure 4.1, a simple ontology represents a movie as consisting of a set of actors, a genre and a rating. To model it, four concepts are used: **movie**, **actor**, **genre** and **rating**. To express that a movie consists

of actors, has a genre and a rating, three relationships of type *is member of* connect the **movie** concept with its constituents.

Numerous types of relationships will be introduced in this document, and each type comes with its own set of attributes. Relationship attributes are typed name/value pairs used to specify characteristics of a relationship. For instance, the relationship type *is member of* has a Boolean attribute named *is single* to specify the cardinality of the relationship. In our example, a movie has multiple actors. Therefore, the *is single* attribute of the relationship connecting the **movie** and **actor** concepts is set to *false*.

4.2 Active processing

This section presents how processing is implemented in the context of Active Ontologies. In addition to concepts and relationships, Active Ontologies contain processing elements distributed over their concepts. Processing is modeled with *rules*, whose conditions are evaluated against a set of *facts*, kept in a *fact store*.

At the core of Active is a specialized *production rule* engine, where data and events stored in a fact store are manipulated by rules written in a target language (currently Java or Javascript) augmented by a light-layer of first-order logic to perform *unification* operations. When the contents of the fact store changes, an *evaluation cycle* (see section 4.2.4 on page 44) is triggered and conditions evaluated. When a rule condition is validated, the associated action is executed. Actions can create events, insert new data (facts) or perform any processing expressed in the target action language.

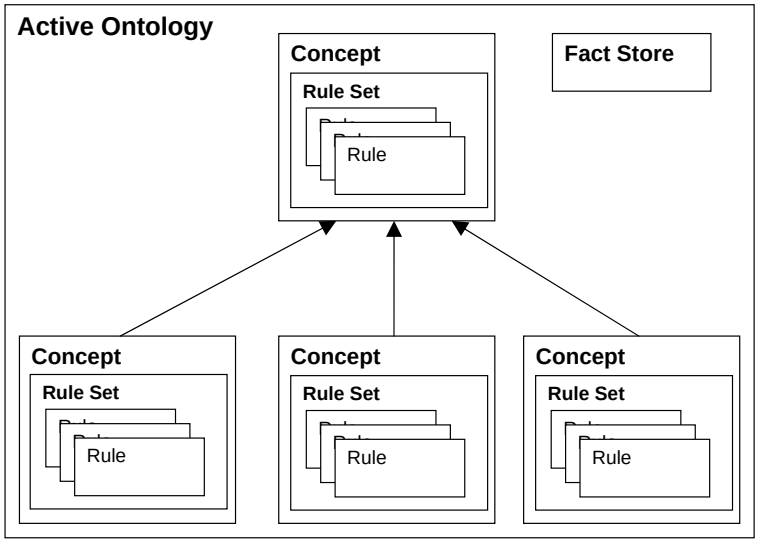


Figure 4.2: Active Ontology Components

4.2.1 Facts

The fundamental data structure in Active is the *fact*. When building an Active based application, facts are used to represent the domain definition, to store the current execution status and to communicate data and events among processing elements. Facts have a life cycle; they are created, asserted and deleted. The life cycle of each individual fact is independent and is defined by Active programmers to create time constrained behaviors.

Inspired by first-order logic predicates, there are four types of facts.

- *Simple facts*: Simple facts are atomic constant values.
  Examples : **1**, **a**, **'john doe'**, **ABxx**
- *Complex facts*: Complex facts are named predicate with one or more arguments of the form *name(t1, ..., tn)*. Arguments may in turn be any fact.
  Examples: **person(john, doe)**, **person($name, doe)**, **position(x(10), y(10))**
- *List facts*: List facts are bracket delimited and comma separated non-ordered collections (sets) of facts.
  Example: **[1, a, position(10,10)]**, **[a,b,c]**, **[1, $D, c]**
- *Variables*: Variable identifiers starts with a dollar sign ($). Anonymous variables, or wildcards, are represented as a single dollar sign ($).
  Examples: **$person**, **$priority**, **$X**, **$**

4.2.2 Unification

This section introduces the concept of unification, the fundamental mechanism through which Active rule conditions are evaluated. Unification is essentially a Boolean operation to compare two facts. If two facts unify, they are considered as equivalents. To explain the principles of unification, we start from the simple cases and gradually increase the complexity of facts we consider.

Simple facts Unification of simple facts is straightforward, the two facts have to be similar. (see table 4.1)

Fact 1	Fact 2	Unify?
a	a	yes
a	b	no
'a car'	'a car'	yes
'a car'	'a train'	no

Table 4.1: Simple Fact Unification

Complex facts Unification of complex facts (structures made out of a functor and a collection of children facts) is more complicated. For the unification to succeed, both facts have to be complex facts, their functors (names) have to be equivalent, they must have the same number of children, and finally all children have to unify in the correct order. The process of complex fact unification can

be seen as a parallel and recursive tree traversal, performed simultaneously on both inputs. At any given point, the Active fact of the traversal from the first fact has to unify with its counter part from the second fact. If the traversal completes and all visited facts have unified, the unification succeeds. If any of the traversed facts fail to unify, the process of structure unification fails. Table 4.2 provides a list of examples.

Fact 1	Fact 2	Unify?
person(bob)	person(bob)	yes
person(bob)	person(john)	no
person(bob)	employee(bob)	no
person(john, doe)	person(john, doe)	yes
person(john, doe)	person(john, wayne)	no
person(name(john, doe), ssn(1234))	person(name(john, doe), ssn(1234))	yes
person(name(john, doe), ssn(1234))	person(name(john, doe), ssn(4567))	no

Table 4.2: Complex Fact Unification

List facts Unification of list facts can be performed in two modes: *strict* or *partially strict*. The *strict* list unification imposes both lists to have the same number of elements and enforces that all of their elements must unify in order. The *strict commutative* list unification does not impose the order of elements. It nevertheless imposes that both lists must have the same number of elements and each element of the first list must unify with one, and only one, element of the second list. Active uses the least strict unification mechanism is the *partially commutative* list unification. In this case, lists do not have to contain the same number of elements. The unification succeeds if all elements of the shortest list unify with at least one element of the longest list. The order does not matter and each element of the long list can be used only once as a "match" for an element of the short list. Table 4.3 show list fact unification examples.

Fact 1	Fact 2	Strict	Partially Commutative
[a]	[a]	yes	yes
[a]	[b]	no	no
[a,b]	[a,b]	yes	yes
[a]	[a,b]	no	yes
[a(1),b(2)]	[a(1),b(2)]	yes	yes

Table 4.3: List Fact Unification

Unification with variables As mentioned above, Active facts can contain variables. Variables play an important role in the process of unification, where they can be seen as wild cards that always unify with their counterparts. 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 fact. Note that anonymous variables are never instantiated and work simply as wild cards.

Fact 1	Fact 2	Unify	Instantiations
a	$v1	yes	v1=a
a	$	yes	-
a(b)	a($v1)	yes	v1=b
a(b,c)	a($v1, c)	yes	v1=b
a(b,c)	a($v1,d)	no	-
[name(john), age(10)]	[name($v1), age($v2)]	yes	v1=john, v2=10
[name(john), age(10)]	[name($v1)]	yes	v1=john
[name(john), age(10)]	[name($v1), age($v1)]	no	-

Table 4.4: Unification with Variables

Using variables, the unification process is more than a simple Boolean operation to compare two facts, it is a rich pattern matching mechanism through which variables can be instantiated and complex conditions expressed. This mechanism is fundamental to the Active framework. For instance, let us consider an example where facts are used to store employees of a company using the following structure:

employee($firstname, $lastname, $title, $employee_id)

A unification based query for retrieve all managers would be :

employee($firstname, $lastname, manager, $employee_id)

The unification mechanism presented in this section is one of the basic concepts of Active. It is used by the fact store of Active to expose a query (read) interface and Active rules, whose conditions are based on unification.

4.2.3 Fact store

This section introduces the Active fact store (or store), and presents a subset of the Active programming *API* (or *Active API*) through a set of simple examples. Fact stores are the dynamic memory of Active, they host and manage facts. Hosting facts means offering a set of functions to persist (write) facts. It also means exposing query (read) functions so that stored facts can easily be retrieved. This section introduces a subset of the fact store API. As we cover Active in more details in this chapter, we will introduce more functions of the Active API.

Read, Write and Modify API

In this section, we introduce four basic methods of the Active API[1]:

- **void Store.writeFact(Fact factToWrite)** : This method takes a simple fact and asserts it into the fact store.

[1]The Active API is described in a Java-like pseudo code. Detailed API definitions depend on the target language selected. (Java or Javascript)

- **Fact[] Store.readFacts(Fact factPattern)**: Uses the unification pattern to retrieve all facts that match a given pattern.
- **void Store.removeFacts(Fact factPattern)** : Uses unification to remove all facts present in the fact store that unify with the pattern provided.
- **void Store.overwriteFact(Fact factToWrite, Fact factPattern)** : This method is a combination of `removeFacts` and `writeFact`. It will first remove all facts that match the given pattern, then write the provided fact. It is the equivalent to the sequential execution of : **Store.removeFacts(Fact factPattern)** and **Store.writeFact(Fact factToWrite)**
- **void Store.scheduleFact(Fact factToWrite, Date date)** : This methods takes a simple fact and schedules it for assertion. The fact store persists the incoming fact but keeps it invisible until the specified date. This basic feature allows Active programmers to model timeouts or any other time-based behavior.

Examples

Lets us consider the example introduced on section 4.2.2. To create a small repository of employees, our store fact can be populated as follows :

```
Store.writeFact(employee(john, doe, accountant, e123))
Store.writeFact(employee(jane, doe, manager, e122))
Store.writeFact(employee(alan, smith, accountant, e100))
Store.writeFact(employee(robert, martin, ceo, e145))
Store.writeFact(employee(mary, jones, receptionist, e235))
Store.writeFact(employee(john, smith, manager, e153)
```

To retrieve all managers, one would use:

```
Fact[] results = Store.readFacts(employee($, $, manager, $))
```

To delete all managers from the data base:

```
Store.removeFacts(employee($, $, manager, $))
```

4.2.4 Rule evaluation cycle

As previously described, an Active Ontology consists 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 *rulesets* attached to concepts. Rulesets are collections of *rules* where each rule consists of a *condition* and an *action*. During an evaluation cycle, condition rules are evaluated and for each condition that evaluates positively, the associated action is triggered : the rule *fires*.

The fact store of an Active Ontology is regularly checked for changes since the last cycle. As long as the content of the store is unchanged, Active is not performing any action. If any modification of the fact store attached to an Active Ontology is detected, an evaluation cycle is triggered. In an evaluation cycle, rule conditions are evaluated against the content of the store and, if the condition is verified the rule fires. Figure 4.3 summarizes the process where each evaluation cycle is a three-step process. A *pre-processing* step checks if the fact

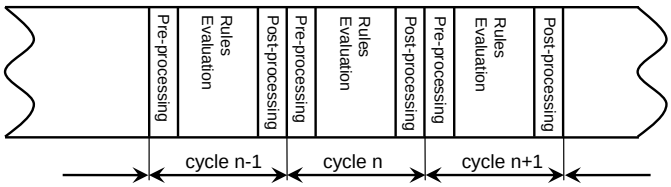


Figure 4.3: Evaluation Cycles

Rule condition EBNF grammar
<rule_condition> ⇒ <Boolean_expression>
<Boolean_expression> ⇒ <term> | <term> <add_op> <Boolean_expression>
<term> ⇒ <factor> | <factor> <mult_op> <term> | <not_op> <factor>
<factor> ⇒ <constant> | (<expression>) | <store_check>
<add_op> ⇒ OR | '||'
<mult_op> ⇒ AND | '&&'
<not_op> ⇒ NOT | '!'
<store-check> ⇒ Store.checkFact(PATTERN) | Store.checkEvent(PATTERN)
<constant> ⇒ true | false

Figure 4.4: EBNF grammar for rule conditions

store has changed, is so a *rule evaluation* processing is executed to be followed by a *post-processing* step in charge of cleanup and timers management.

The current implementation of the Active Server evaluates Active Ontologies up to ten times per second. However, each Active Ontology can be individually configured to be evaluated less frequently, depending on its purpose. The maximum evaluation cycle of 100 milliseconds (10 Hz) has been chosen for practical reasons. First, the prototype applications we intend to build are user-driven systems, their are not real-time control system that need to react within a few microseconds. We are going to build and experimental system, designed to work within the constraints of simple user centric applications. A performance evaluation and potential optimizations of our software suite are detailed in section 7.3 on page 175.

A rule condition can be represented as conditional trigger used to detect specific events on the fact store. Rules conditions are *Boolean expressions*, whose value after evaluation is true or false. Conditions are Boolean expression consisting of *operators*, *values* and *store-checks*. Supported operators include *disjunctions* (OR), *conjunctions* (AND) and *negation* (NOT). Values are Boolean constants (true or false). Elements of type *store-check* define a fact pattern, based on unification to test the presence of specific facts in the store.

There are two flavors of store-checks. The first type, *check-event*, tests only against the group of new facts that triggered the evaluation cycle. The second type, a *check-fact,* operates over the whole fact store for any fact that matches its pattern. The Active API provides two primitives to define store-checks in rule conditions:

Store.checkFact(Fact factPattern)
Store.checkEvent(Fact factPattern)

For a given evaluation cycle, *check-events* only check against the small set of facts that where modified since the last evaluation, whereas *check-facts* check against the entire fact store. Following sections show how the difference between *check-events* and *check-facts* is used to significantly improve the performance behavior of Active applications.

Actions are *snippets* of code expressed in a target programming language (currently Javascript and Java are supported) executed when the associated condition is evaluated as true. The Active API offers a collection of pre-compiled utility functions to print messages to the console or access the fact store. In addition, extensions can be designed and packaged to extend the set of functions available to Active programmers. Such extensions are pre-packaged libraries to support AI methods and techniques (i.e. language processing, process modeling) or various utility modules (i.e. web services/SOAP communications, database connectivity). The Active programming SDK allows programmers to create new extensions for the Active system.

4.2.5 Simple conditions

To describe the evaluation process of rule conditions, let us consider the following notation:

- *FS (Fact Store)* is the set of all known facts (entire fact store).
- *MF (Modified Facts)* is the set of new facts asserted since the previous evaluation cycle, that triggered the current evaluation cycle. *MF* is a subset of *FS*.
- *R* is the rule to evaluate, whose condition is a valid expression according to the grammar show in figure 4.4. The expression contains *check-event* and *check-store* elements.

For the rule *R* to fire, at least one of its *check-event* provides a pattern that unifies with at least one fact of *MF*.

Example

As an example, figure 4.8 shows a rule to detect the addition of new employees and managers.

Rule name	PrintNewEmployee
Condition	Store.checkEvent(employee($, $, $, $))
Action	System.printConsole('A new employee was added');

Rule name	PrintNewManager
Condition	Store.checkEvent(employee($, $, $, manager))
Action	System.printConsole('A new manager was added');

Figure 4.5: Simple Rules

4.2.6 Conditions with variables

Variables play an important role in the process of condition rule evaluation. Unnamed variables are wildcards that always unify with their counterparts. Named variables are more useful. When a named variable unifies with a fact, it becomes instantiated with the matching fact and, for the rest of the unification process, it is not considered as a variable anymore but as a known *bound* value. The instantiated, or *bound*, variable whose value was set from the conditions part of the rule, can be used in the action part of the rule. For instance, the rule shown in table 4.6 uses the retrieved lastname of managers from the rule condition (named variable $lastname) to print it in the action part.

When variables are used, the process of rule evaluation is slightly more complex. It starts similarly as the description given in section 4.2.5, but if the rule condition has variables, more processing is required to gather actual values for the variables. For each *check-event* of the rule condition, a result set is fetched by collecting all facts from *MF* that match the *check-event* pattern. Similarly, for each *check-fact*, a result set if fetched from *FS*. At this point, each *store-check* (whether it is a *check-fact* or a *check-event*) has a corresponding set of facts that match its pattern. Since bound variables are used in the action part of the rule, the last step of processing consist of executing the rule action with all valid and unique combination of facts that were gathered.

Examples

Rule name	PrintNewManager
Condition	Store.checkEvent(employee($, $name, $, manager))
Action	System.printConsole('A new manager '+$name+'was added');

Figure 4.6: Simple Rule with a Variable

4.2.7 Compound rule conditions

As shown before, rule conditions contain one or more *store-checks*. Complex rule conditions may involve multiple dependent queries to the fact store, they are *compound rule conditions*. Dependencies among queries are expressed with named variables, where the same variable appears in multiple *check-stores*.

Example

As an example, let us consider the rule shown in figure 4.7 designed to print out employees based on their title. To trigger the rule, a fact of the form has to be asserted into the fact store: print($type_of_employee)
For instance, to print all managers, on would assert: print(manager)

Rule name	PrintAllEmployeesByCategories
Condition	Store.checkEvent(print($category)) && Store.checkFact(employee($,$name, $, $category))
Action	System.printConsole('Employee ' + $name + ' is a ' + $category);

Figure 4.7: Compound Rule

The rule shown in figure 4.7 is a compound condition involving two dependent conditions. During the evaluation cycle, the check-event :

```
Store.checkEvent(print($category))
```

triggers the evaluation of the rule. It is acting as a pre-condition of the rule, triggered to detect the print command in the *MF* and bind the variable $category. In our example, the $category is bound to the value manager, and the second store-check (a check-fact):

```
Store.checkFact(employee($,$name, $, $category))
```

searches the whole store with the pattern : employee($, $lastname, $, manager). For each match, the second named variable $lastname gets bound with the actual manager name and the rule fires. The action code snippet is therefore executed with two bound variables and can print out the correct information. When writing Active compound conditions, it is a good practice to use check-events as *guards* to be evaluated before evaluating more costly check-fact over the entire fact store. Following this approach, Active has been inspired by ECA (Event Condition Action) techniques mentioned in section 2.1. Section 7.3 on page 175shows how this best practice can dramatically improve the performances of an Active-based application.

4.2.8 Cascade Processing

An important characteristic of production rule based systems consists of cascading rule executions, where specialized rules would detect a situation and communicate results by asserting new facts to trigger further processing. Primitives of the Active API can be used in rules actions to assert new facts, and therefore trigger more evaluation cycles and rules firing.

For instance, let us consider two rules, one to detect any assertion of a new manager and another one specialized to send notifications as emails.

Rule name	DetectNewManager
Condition	Store.checkEvent(employee($, $name, $, manager))
Action	System.printConsole('A new manager was added'); Fact.writeFact(send_email('bob@company.com', $name + ' was added as a new manager');

Rule name	SendEmail
Condition	Store.checkEvent(send_email($to, $message))
Action	Mail.send($to, $message);

Figure 4.8: Simple Rule Cascading Execution

This principle of cascading rule execution is crucial in Active-based methodologies. By nature, Active Ontologies organize rules according to a network of interconnected concepts, where each concept is instrumented with specialized rules. When a rule attached to a concept fires, it uses ontological relationships to spread information along the structure and propagate more processing towards connected concepts. Section 5.1 shows how this principle is used for named-pipe communications and invocation-like techniques.

4.2.9 Fact creation

Section 4.2.8 introduced how cascading execution of rules is implemented by creating new facts. Active supports multiple techniques to assert new data into the fact store.

Using the diagram of figure 4.3 on page 45 as a reference, we can chronologically describe what happens during fact assertions. At evaluation cycle n a fact f is to be asserted into the fact store, either internally by a the action code of a rule that fired or externally through the external communication channels of the Active Server (i.e. a sensor reports an event). This asynchronous event can occur at any time during the n^{th} cycle, after some rules have already been checked for evaluation while others are still to be evaluated. To ensure that all rules take the new fact into account within the same cycle, f is not immediately written but scheduled to be actually asserted at the end of the cycle during the *post processing* stage. Therefore, at the cycle *n+1,* all rules will have a chance to take f into consideration.

In many cases, Active rules assert facts to inform other processing components about a specific event. They do not intend to assert new permanent pieces of information but rather send an short lived signal. To facilitate this technique, Active supports the notion of specialized facts called *event-facts*. An event is a fact that lasts only for one evaluation cycle, it is ephemeral. As an example, let us assume a system with two rules, *R1* and *R2* (see figure 4.9). At cycle *n*, *R1* fires and creates the event-fact event(r2). At the end of cycle *n* the *post-processing* stage actually asserts event(r2) and tags it as being an event-fact. At the cycle *n+1*, *R2* fires and performs its action. At the end of cycle *n+1* the *post-processing* stage looks for event-facts to delete and therefore automatically removes event(r2).

Rule name	R1
Condition	Store.checkEvent(test($value))
Action	if (factToInt($value)>2) Store.createEvent(event(r2));

Rule name	R2
Condition	Store.checkEvent(event(r2))
Action	System.printConsole("Rule R2 fired");

Figure 4.9: Event Generation

The Active API offers two primitives to assert facts and event-facts:

Store.writeFact(Fact fact)
Store.createEvent(Fact fact)

Active includes the time dimension into its programming model. For instance, rules can be programmed to fire if a specific event does not occur within a given amount of time. To model this type of behavior, Active programmers use the fact scheduling technique. The Active API function Store.scheduleFact(Fact f, Date d) instructs the fact store to assert a fact at a specified moment in time and can be used to install alarms that will trigger condition rule execution at a specific time. Figure 4.10 illustrates this technique. When rule *R1* fires it installs an alarm to fire after a specific delay by scheduling the assertion of a fact of the form alarm($value). A specialized thread of the Active Server is in charge of keeping track of facts and event-facts scheduled for assertion. When the time delay is reached, it asserts the facts into the store, thus triggering an evaluation cycle. In our example, rule *R2* would fire and report the alarm.

Rule name	R1
Condition	Store.checkEvent(install_alarm($value, $time_delay))
Action	Store.scheduleEvent(alarm($value), factToInt($time_delay))

Rule name	R2
Condition	Store.checkEvent(alarm($value))
Action	System.printConsole("Alarm " + $value + " fired");

Figure 4.10: Time-based fact assertion

Usage of this technique is further demonstrated in section 5.1 on page 57.

Active applications consist of one or more specialized Active Ontologies, each performing an aspect of the intelligent assistant system to build. Therefore, Active Ontologies need to communicate with each other to synchronize their actions and share information. In the context of Active, all communications are based on fact assertion and rule firing. Two sets of primitives are offered by the Active API to provide cross-ontology processing.

On the listener side, two specialized check-stores can be used to include events happening in different Active Ontologies in rule conditions:

Store.checkFactFrom(Fact factPattern, String ontologyName)
Store.checkEventFrom(Fact factPattern, String ontologyName)

On the sender side, the Active API offers primitives to assert and schedule facts and event-facts into foreign Active Ontologies:

Store.writeFactTo(Fact fact, String ontologyName)
Store.createEventTo(Fact fact, String ontologyName)
Store.scheduleFactTo(Fact fact, Date date, String ontologyName)
Store.scheduleEventTo(Fact fact, Date date, String ontologyName)

Figure 4.11 shows an example where the Active Ontology *A1* installs an alarm into the Active Ontology *A2*.

Rule name	R1 (In Active Ontology A1)
Condition	Store.checkEvent(install_alarm($value, $time_delay))
Action	Store.scheduleEventTo(alarm($value), factToInt($time_delay), A2)

Rule name	R2 (In Active Ontology A2)
Condition	Store.checkEvent(alarm($value))
Action	System.printConsole("Alarm " + $value + " fired");

Figure 4.11: Cross-Ontology Communication

4.2.10 Evaluation cycle control

As shown in figure 4.2 on page 40, rules are grouped into rulesets attached to concepts. Both rules and rulesets have priorities, expressed as an integer value. The higher the value, the higher the priority. During an evaluation cycle, rule-sets are evaluated in their order of priority (high priority first), and within rulesets rules are also evaluated in their order of priority.

Two evaluation policies control the processing behavior of a rule set. In the case of the *evaluate-all* policy, all rules of the rule-set are evaluated in their order of priority. In the case of *first-success* policy rules of the rule-set are evaluated in their order of priority until one of them succeeds. After that, the evaluation of the rule-set stops and remaining rules (of lower priority) are skipped.

4.3 The Active software suite

This section describes the current implementation of the Active software suite. The suite is a Java-implemented set of applications designed to be extensible and open. It consists of four components. The *Active Server*, the *Active Editor*, *NL Test*, and the *Active Console*. To ensure ease of integration and extensibility, components of the Active platform communicate through web service (SOAP) interfaces. For both the Active Editor and Active Server, an open (SDK) plug-in mechanism enables researchers to package AI functionality to allow developers to apply and combine the concepts quickly and easily. A set of Active extensions is available for language parsing, multimodal fusion, dialog management and web services integration.

4.3.1 Active Editor

The Active Editor (see figure 4.12) is an *IDE* (Integrated Development Environment) used by programmers to model, program, deploy and test Active applications.

Working areas

The editor consists of four main working areas.

- *The Graph pane* (Top left). As a multi-document container, this area allows navigation among opened ontology files. Each tab hosts a visual

view of the Active Ontologies being worked on, where concepts can be added, removed, and visually arranged. Each tab is labeled with the name of the Active Ontology it contains and provides a dirty flag (orange star) to indicate that an Active Ontology has been modified but not yet saved.

- *The Tree Navigation pane* (Bottom left). The second area is the ontology navigation panel, showing a tree view of the selected ontology. It offers a tree-like navigation among concepts and rules.

- *The Code pane* (Bottom right). Based on the selected ontology element (concept, rule set, or rule), the code pane allows attributes and code edition. If the selected element is a concept, the code panel allows edition of the concept's name, its evaluation priority and text description. If the selected element is a rule, the code panel features two sub-tabs. The first for edition of the rule name, attributes, and text description, the second features two text areas to express the condition and action of the rule.

- *The Debug pane* (Top right). This section is designed for testing and debugging Active Ontologies. Programmers can assert facts into an Active Server where their Active Ontologies have been deployed, and get direct feedback through system level and user defined events.

Working with the Active Editor

Active programmers typically work with the Active Editor (the *editor*) connected with a live Active Server (the *server*). Active Ontologies being worked on are regularly *deployed* into the server for testing and execution. The action of deploying an Active Ontology actually sends its definition (concepts, rulesets, rules and relationships) from the editor into the server for execution. Once deployed, Active Ontologies can be executed in debug mode or release mode. In debug mode, the server sends runtime events back to the editor to inform programmers about execution steps. The Debug pane of the editor reports a list of runtime events received from the server. Each event is time-stamped and carries information about four types of events:

- Evaluation cycle. An event is generated for each evaluation cycle.

- Rule execution. For each rule that fires, an event reports the name of the rule and the values of all bound variables.

- Execution error. If the snippet of code executed when a rule fires fails to properly execute, a full error reports is reported.

- Logging. For debugging purposes, action code snippets can log information to be reported directly to the editor.

The editor support simultaneous editing of Active Ontologies, allowing developers to work on all aspects of their intelligent assistant application. It allows them to design, test and debug end-to-end applications within the same unified and coherent environment.

Plug-ins and Wizards

In addition to its rich visual feature set, the Active Editor comes with an *SDK* allowing Active programmers to create *Active Editor Plug-ins*, or simply *plug-ins*. A plug-in is an add-on to the Active Editor providing an interactive dialog box (or *wizard*) to ease and automate the creation of concepts. Plug-ins are written in Java and implement the required classes and interfaces of the Active Editor SDK to be seamlessly integrated to the IDE. When prototyping and test new approaches, Active rules are manually written and tested. Once the approach is functional, a plug-in can be written to gather high level information from a user through a wizard and automatically generate an Active concept with all its rules and rulesets. For instance, a plug-in for language processing applications launches a specialized wizard that will ask programmers to provide a vocabulary set. Once the list is provided, the plug-in automatically creates and inserts a concept containing all necessary rules and rulesets.

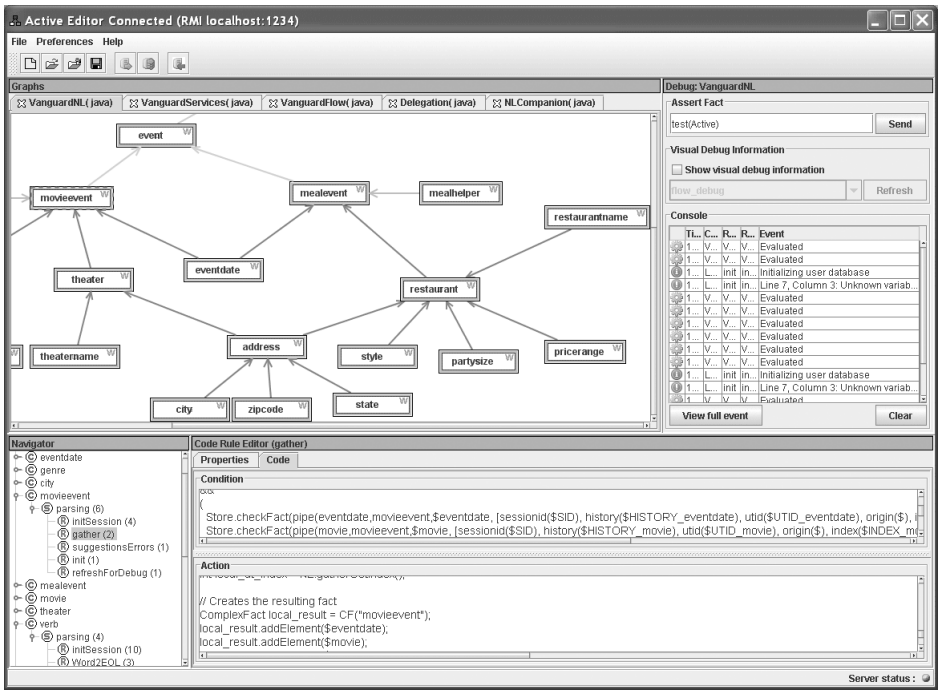


Figure 4.12: Active Editor

4.3.2 Active Server

The Active Server (see figure 4.13) is a scalable runtime engine that hosts and executes one or more Active programs. The Active Server is written in Java and can either be running as a standalone application or deployed on a J2EE compliant application server. The server consists of the following functional elements:

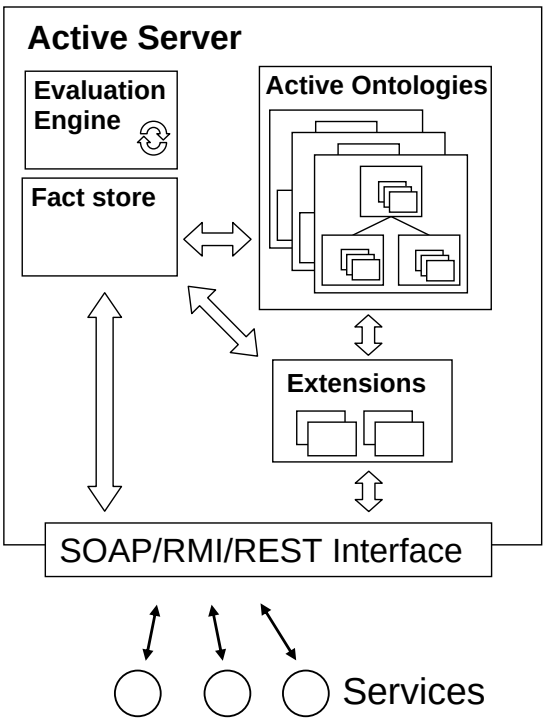


Figure 4.13: Active Server

- Fact Store. An implementation of the fact store described earlier in this chapter (see section 4.2.3 on page 43). The current implementation of the fact store is an in memory store. The store is nevertheless abstracted out and can therefore optionally support implementations that persist facts into an *RDBMS*, providing robustness to catastrophic failures.

- Active Ontology repository. Internal store where the Active server persists the definitions of Active Ontologies to run. The current implementation of the store is file-based, so that restarting the Active Server does not imply re-deploying Active Ontologies. Similarly to the fact store, the Active Ontology repository is abstracted out and can optionally connect to an RDBMS for scalability and robustness. Each deployed Active Ontology has its own private space in the fact store to persist its execution context.

- Evaluation engine. At the core of the Active Server, the evaluation engine regularly checks each Active Ontology space in the fact store for changes. If any fact has changed since the last check, an evaluation cycle (see section 4.2.4 on page 44) is triggered.

- Extensions. Similarly to the Active Editor, the Active Server is extensible using a public SDK allowing programmers to create *Active Server Extensions* or simply *extensions*. Extensions are pre-compiled libraries written

in Java that encapsulate processing into reusable components. Active Server comes with a built-in extension that offers functions to print statements in the console or generate debugging events. Extensions can also be used to extend the Active Server capabilities to sense its environment. For instance, extensions have been designed and implemented so that Active Server can read and send emails (from any *POP* compliant mail server) or have an *Yahoo Instant Messenger*© front-end as a user interaction modality.

- Communication Interface. As the core of Active-powered applications, the Active Server needs to be open and easily connected to sensors and actors. It therefore exposes access to its internal components through multiple standard communication channels (SOAP, REST and RMI).
  - Fact store. The fact store is accessible for read and write. Sensors and user interfaces report their events by asserting (writing) facts into the fact store. The next evaluation cycle will take incoming events into account and trigger relevant rule processing. In addition, the fact store can be read by external applications for debugging and monitoring of activities.
  - The Active Ontology repository is also exposed so that external applications, such as the Active Editor, can deploy new Active Ontologies for executions. Management tools, such as the Active Console, can manage the status of deployed Active Ontologies.

4.3.3 Active Console

The Active Console permits observation and maintenance of a running Active Server. The console offers multiple operation panels for fact store introspection and for Active Server administration.

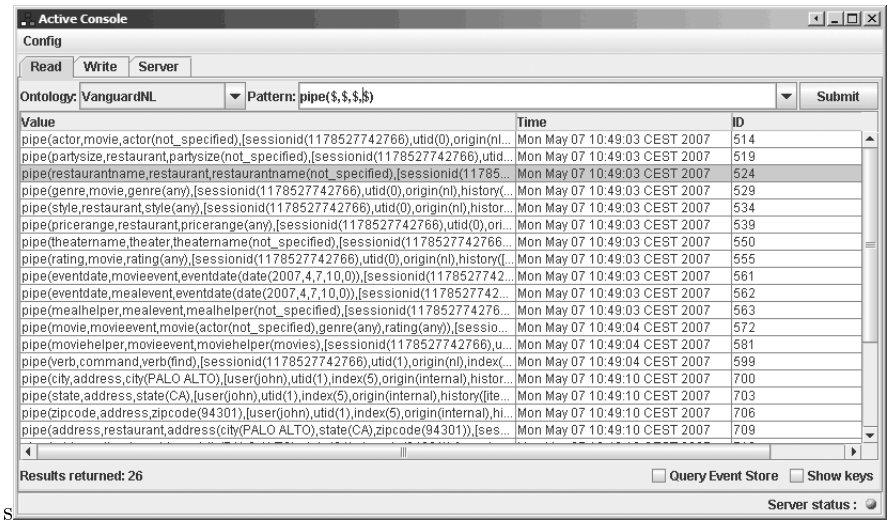


Figure 4.14: Active Console

The fact store introspection panel (see figure 4.14) allows Active programmers to inspect the content of the fact store using pattern-based queries. For each query, the fact store returns all facts that unify with the provided pattern. The Active Server administration panel is used to manage deployed Active Ontologies and available Active Extensions.

4.3.4 Language processing test console

The language processing console provides an interface for developers to test their natural language applications, entering in queries and analyzing the results returned by natural language ontologies. The tool supports both an interactive mode, as well as a batch mode, essential for regression testing – this way as the NL domain evolves, application developers can ensure that the overall system performance in terms of understanding queries has improved, not degraded. This tool is described in more details on page 92.

4.4 Conclusion

Following the *Theory of Operations* chapter that motivated our definition and vision for a unified platform for intelligent assistant applications, this chapter presented in more details the core our tool, the Active framework.

First, we defined in more details the concept of Active Ontologies. We presented how Active Ontologies are used as a conventional ontology to model knowledge about a domain. We then presented how Active Ontologies are also execution environments based on facts and rules. Basic concepts such as fact unification and evaluation cycles have been presented to further explain Active based processing. A series of simple examples follow, to gradually introduce the most important principles and concepts used in Active programming.

The last section of the chapter presented the current implementation of the Active framework. The Java-based software suite consists of an IDE (the Active Editor), a runtime engine (the Active Server), a testing tool for natural language applications, and an administration tool (the Active Console).

Based on the *Theory of Operations* and *Active Kernel*, the next chapters of this document present in details the Active-based methodologies to implement, in a unified and coherent framework, major components required to built intelligent assistant applications.

Chapter 5

Active Methodologies

This chapter presents how Active Ontologies are used as the foundation of intelligent applications. Each section presents a methodology, and has a brief introduction and a conclusion to summarize what has been achieved. Section 5.1 introduces low level constructs used by Active Ontologies to communicate and delegate invocation of sub tasks. Section 5.2 presents two language processing techniques implemented with Active. Then, section 5.4 describe how Active Ontologies are used to model complex actions and execute them in a business process engine style. Finally, section 5.3 describes how dynamic selection of services is implemented in Active.

5.1 Basic methods

This section presents two basic mechanisms implemented in Active to achieve communication over pipes and asynchronous *RPC*-like invocations.

5.1.1 Communication channels

General approach

This section describes a convention to be followed by Active programmers to implement inter-concept communication using the notion of virtual *communication channels* or *pipes*. A pipe has a *source* (the sender), a *destination* (the receiver) and some information to communicate. As the source writes a new piece of information into the pipe, the destination is notified and can read the content of the pipe.

In the context of Active, this technique is implemented using cascading rule execution (introduced in section 4.2.8 on page 48). Typically, both the source and the destination are rules, the source asserts a fact as part of its action code and the destination reacts through its condition being verified. Our convention to produce pipe-based communication uses facts of the following form:

Definition 1 Standardized pipe fact

```
pipe($source_name, $destination_name, $data, $attribute_list)
```

Where the main elements are:

- source_name: To identify of the source of the pipe
- destination_name: To identify the destination of the pipe
- data: The information to convey across the pipe
- attribute_list: A list of optional attributes of the form : [$attribute1, attribute2, ...], used to qualify the information passed in the pipe. For instance, confidence or reliability values can be used to assess the quality of the information communicated. The convention described here does not impose any specific attribute of the list, it simply states that the fourth member of the pipe fact is a list.

Usage

Figure 5.1 shows an example of two rules exchanging information through a communication channel. To trigger the exchange, the rule source S1 asserts a fact that complies with the pipe-based communication convention. It specifies its name as the source, D1 as the destination and communicates the data value(test). The rule D1 uses a condition pattern pipe($source, D1, $value, $attribute_list) designed to react to any communication with S1 as destination, regardless or the source, the data or the attributes.

Using the flexibility of fact unification (see section 4.2.2 on page 41) the destination of the channel can define more strict conditions on the pipe. To listen to facts coming from S1 only, it would use Store.checkEvent(pipe(S1, D1, $value, $attribute_list)) where the source is not a variable anymore but the constant S1.

Similarly, the source can broadcast a message to all listeners by asserting a fact of the form pipe(S1, $, value(test),[]) where the destination is not specified but represented with a wild unnamed variable. Similar unification-based constraints also can be applied on the type of data to transmit and the list of channel attributes.

Rule name	S1
Condition	Store.checkEvent(test(simple_communication))
Action	Fact.overwriteFact(pipe(S1, D1, value(test),[]), pipe(S1, D1, $, $));

Rule name	D1
Condition	Store.checkEvent(pipe($source, D1, $value, $attribute_list))
Action	System.printConsole("Got message from " + $source + " value is" + $value);

Figure 5.1: Simple channel-based communication

By convention, pipes contain a single value and every pipe assertion overwrites its current content. This is why S1 uses the Fact.overwriteFact function (introduced in section 4.2.3 on page 43) using the deletion pattern pipe(s1, d1, $, $) to ensure that there is only one value in the pipe at any given time.

Finally, using cross-ontology processing (see section 4.2.9 on page 49) channels can connect rules and concepts across multiple Active Ontologies.

5.1.2 Invocation mechanism

General approach

Using facts and rules, an *invocation mechanism* has been designed to implement asynchronous RPC-style calls with Active. An invocation is a two-way operation where a *caller* (typically the action of an Active rule) invokes a functionality provided by a *callee* (typically another Active rule) by providing an operation name and a list of parameters. In return, when processing is done the callee returns some results to the caller.

The caller triggers an invocation by asserting a standardized *invocation fact* (see definition 2) that contains the name of the function to call and a list of invocation attributes. The callee is implemented as a rule whose condition pattern reacts to standardized invocation facts and whose action provides the processing to perform. Once done, the callee asserts a standardized *invocation result fact* (see definition 3) to notify the caller about the results. In turn, the caller uses a rule whose condition reacts to the invocation result fact to retrieve the invocation results. Since Active uses a production rule model where all interactions are based on events, all invocations are asynchronous. The caller does not passively wait for the answer, but provides a specialized callback rule to process the invocation results when produced by the callee.

In the context of Active, invocations are similar to calling loosely coupled services. At runtime, there is no guarantee that a callee is available to respond to a caller. Therefore, a timeout mechanism notifies the caller if the callee does not respond within a specified amount of time. Finally, a caller can have multiple pending invocations, therefore each invocation is tagged with a unique *transaction id*, allowing the caller to correlate asynchronous results with the correct call.

Due to their asynchronous nature, invocations are considered as sessions with a unique identifier and associated context. The context is the set of data the caller had gathered at invocation time that needs to be retrieved when the result comes. Since results are processed by a stateless callback rule, the invocation identifier is used to retrieve the context at the time of the invocation. This is the responsibility of the caller to store at invocation any piece data to be retrieved when invocation results are available.

Implementation

The implementation of this technique is based on the assertion of standardized facts that convey the invocation information. These facts are defined as follows:

Definition 2 Invocation fact

```
invoke($operation_name,$input_params, [tx_id($tx_id), timeout($timeout)]))
```

Definition 3 Invocation result fact

```
invoked($tx_id, $result, [operation($operation_name), status($status)])
```

The fact definition 2 presents the standardized invocation facts. First, the caller asserts an *invocation fact* to trigger invocations. The asserted fact provides the following values:

- operation_name: The name of the operation to invoke
- input_params: A list of input parameters
- tx_id: Unique transaction id for the invocation
- timeout: How long should the caller wait until the invocation is considered to have failed

Similarly, the response of the callee is formatted as an *invocation result fact* (fact definition 3), consisting of:

- tx_id: Transaction id used by the caller to correlate the result with the response
- result: The results of the processing
- status: Invocation status, which could be success, failed or timeout

Finally, *context facts* are used to store the invocation context to be retrieved when results are available.

Definition 4 Invocation context fact

```
context($tx_id, $name, $value)
```

An invocation context fact consists of three elements:

- tx_id: Unique transaction id for the invocation
- name : The name of the variable to store
- value : The value of the stored variable

Now that we have introduced standardized facts used for invocation techniques, let us examine how they are used. Figure 5.2 illustrates the sequence of events unfolding when an invocation is performed.

To initiate the transaction, the caller performs a four-step sequence of actions:

1. Create a unique invocation identifier used to correlate the invocation with its result asynchronously produced by the callee.
2. Assert an *invocation fact* to describe all the invocation parameters.
3. Optionally assert all *invocation context facts* used to retrieve the invocation context when results are produced.
4. Schedule an *invocation result fact* to report a potential timeout. The fact to schedule is of the form:

   invoked($tx_id, $, [status(timeout)])

   The fact is asserted using the method

   void Store.scheduleFact(Fact factToWrite, Date date)

   introduced in section 4.2.3 on page 43. Therefore, if no callee reacts, an invocation result facts is automatically asserted after the specified amount of time to notify the caller that no callee has reacted.

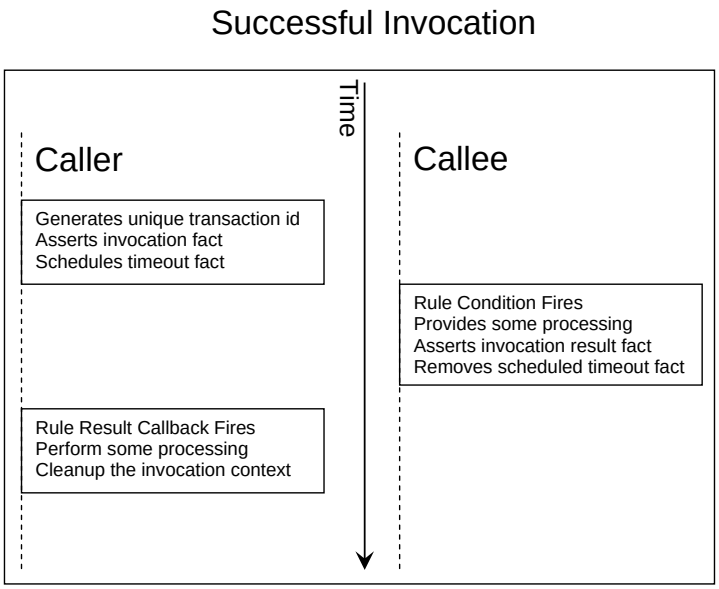


Figure 5.2: Invocation timeline

At this point, a callee reacts through a rule whose condition matches the *invocation fact* asserted by the caller.

1. Extracts input information. The callee needs to get the unique id of the transaction and all relevant input attributes required for its processing.

2. Perform its specific processing. The callee implements some business logic to be processed.

3. Communicate its results. Once the processing is done, the callee asserts an *invocation result fact* to communicate results and the status of the invocation.

4. Remove the scheduled timeout fact. Finally, since it has responded to the invocation, it is the responsibility of the callee to remove the *invocation result fact* previously asserted by the caller to report a potential timeout.

If there is no callee to respond to the invocation request, the scheduled invocation result fact to report a timeout gets automatically asserted when the timeout value is reached. It then triggers the invocation result callback rule installed by the caller to process the result of the invocation. (See figure 5.3)

Finally, the caller gets the invocation result through an invocation callback rule that listens to *invocation result facts*. This rule performs three main tasks.

1. Retrieve the results and invocation context previously saved.

2. Perform all relevant processing related to the invocation completion.

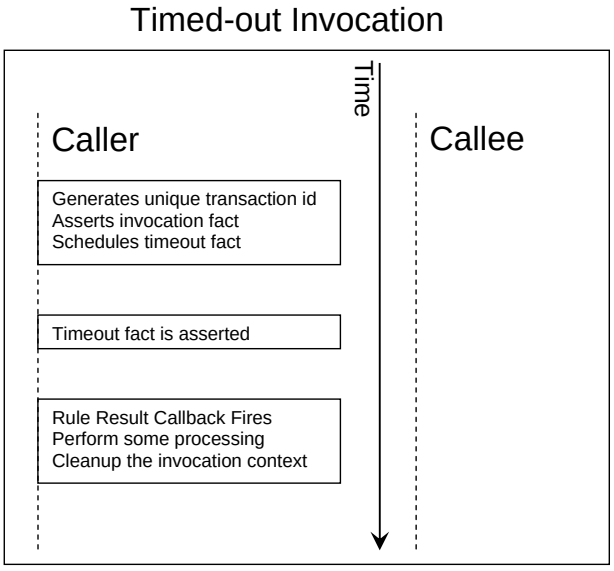


Figure 5.3: Invocation timeout timeline

3. Cleanup. All context related information of the invocation needs to be cleaned up.

To facilitate the use of this technique, an Active Sever extension provides high level functions. To help the reader understand this technique, here is a brief introduction of the most important functions provided by the extension. First, to initiate the invocation the caller uses:

String Invocation.invoke(String operationName, List inputParams, List invocationAttrs)

Where the operationName is the method to call, inputParams the input parameters to pass and invocationAttrs optional attributes about the invocation process. Such attributes specify timeout information (how long is caller ready to wait) and the name of the Active Ontology where the callee is deployed.

The method generates a unique transaction id for the invocation, creates and asserts the standardized invocation fact. It also uses timeout information to schedule an invocation result fact reporting a timeout.

On the callee side, to report results and cleanup invocation data the following method is used:

void Invocation.invoked(String invocationId, Fact result, boolean status)

This method first generates and asserts the invocation result fact providing the caller with the invocation results. In addition, since no timeout needs to be generated, the method removes the previously scheduled timeout fact.

Finally, when the results come back to the caller through its callback invocation rule, the method:

void Invocation.finalizeInvocation(String invocationId)

automatically removes all context facts that were created for the invocation that is now complete.

To illustrate this technique and the usage of the Active Invocation extension, let us consider the Active rules shown in figure 5.4. The caller uses two rules, CallerInvoke to trigger the invocation and CallerResultCallback as the invocation callback to process the result. The callee, or service provider, is implemented a single rule CalleeEcho offering an echo function.

Rule name	CallerInvoke
Condition	Store.checkEvent(test(invocation))
Action	String tx_id = Invocation.invoke('echo', [message('hello')], [timeout(1000)]);

Rule name	CalleeEcho
Condition	Store.checkEvent(invoke('echo',[message($MESSAGE)], [tx_id($INVOCATION_ID)]))
Action	Invocation.invocationSuccess(INVOCATION_ID, 'Echo: '+MESSAGE)

Rule name	CallerResultCallback
Condition	Store.checkEvent(invoked($TX_ID, $RESULT, [operation(echo), status(success)]))
Action	System.printConsole("result=" + $RESULT)

Figure 5.4: Rules for simple invocation

Discussion

The invocation technique presented in this section may seem straightforward and simplistic. It is nevertheless an important first step towards more interesting techniques and powerful approaches to implementing core components of intelligent systems. First, it demonstrates that a combination of rule-based design, unification and time-based fact management allows Active to not only model pure rule-based mechanisms, but also more classic programming constructs such as pipe communication and asynchronous invocation. More importantly, the flexibility and ease of use of the systems allows us to very quickly tackle interesting problems. For instance, what happens if there are multiple callees able to answer a request from a caller? Do we want to call all of them? Pick the best? This is the problem at the base of delegated computing and, in our case, the foundation of our Active-based approach to dynamic selection of services. (See section 5.3 on page 96 for details).

5.1.3 Conclusion

This section introduced two simple Active-based programming techniques: pipe based communication and asynchronous invocation. These two examples illustrate how techniques are implemented by defining standardized facts and encapsulating processing into Active extensions. Now that these basic techniques

have been introduced, we will present how they have been used as the basis of more complex and challenging components.

5.2 Language Understanding

This section focuses on language understanding techniques developed within the Active framework. We start with an introduction to define the role of language processing in intelligent assistant applications. Then, we present how a classic grammar-based approach has been implemented as an Active Ontology, followed by a more innovative, powerful and integrated language processing method based on domain definitions. Finally, a conclusion summarizes our results and compares both techniques.

5.2.1 Introduction

By their very nature, user-centric applications need to communicate as naturally as possible with users. Therefore, providing a *natural language understanding* (NLU) component is important. Natural language understanding consists of mapping sequences of utterances into formal structures designed for processing. Ultimately, intelligent assistant applications should provide natural dialog-based interaction, where users can literally say or express anything at any time. The main characteristics of the NLU component we need can be summarized as follows:

- *Robust parsing.* Our work focuses on dialog-like communications, providing robust parsing of short utterances. We are attempting to process, understand and classify large text documents. Robust parsing needs to deal with natural human interactions. People communicate with partial utterances, relying on context to ensure understanding. We also use *disfluencies* such as interjections, words repetitions or corrections. To be effective, the parsing technique therefore needs to be robust to short, unconstrained and partial utterances by ignoring (skipping) unnecessary words.

- *Dialog and Context.* Interactions with assistants consist of multiple utterances, sharing and contributing to a context. For instance, one could express *"find movies in San Francisco tonight"*, followed by *"oh, and also book me a table at a French restaurant"*. Both utterances share the same context, therefore the second part of the dialog should not have to specify the location.

- *Data validation and resolution.* The NLU component needs to produce valid and complete information to ensure efficient processing. For instance, if a city name is specified by an utterance, the NLU component not only detects the name of the city to consider, but could also provide additional data such as a zipcode or a state name. References to "on Tuesday" or "in three days" can be validated and resolved to a canonical form (e.g. date(08, 10, 2007)). Anaphoric references such as "him" should resolve to the appropriate values given context.

- *Suggestions/Errors.* The NLU module needs to inform the user about its reasoning by providing *suggestions* and *errors*. Suggestions inform users about what could be expressed in their queries. For instance, users trying to book a restaurants may be told by the assistant that they can specify a price range. Errors explain why the assistant cannot process a user query. It might be because of missing information or conflicting data preventing the assistant from proceeding with user requests.

- *Ambiguity management.* Dealing with ambiguities is a two-step process: first alternate interpretations are detected, then resolved. For instance, if a user expresses a query of the form : *"give me all american comedies in sunnyvale for tonight and book a table for two at an italian restaurant"*, followed by a second statement saying: *"no, I would like a French one"*. This is ambiguous, is the user asking for French comedies or a French restaurant? Once the ambiguity is detected, the assistant should use strategies to resolve the problem. Resolution strategies may leverage context (assume the user is referring to his or her latest utterance, restaurants in our example), use incremental learning (usually this person is not decisive about movies but always eats Italian food) or simply go back to the user asking for a clarification.

- *Multiple modalities.* Intelligent assistants perceive language over multiple sources. For instance, utterances can be typed in a user interface, the result of a speech, handwriting or gesture recognizer or received as an asynchronous message (email, short message or even an instant message). The NLU component should then be open enough to deal with information coming from multiple, heterogeneous sources.

The remaining parts of this section present how NLU has been implemented in the Active context. First, we demonstrate how a traditional grammar-based chart parser can be modeled using Active Ontologies. Then, after presenting the strengths and weaknesses of grammar-based parsers, we illustrate a language processing methodology based on Active Semantic Networks whose properties are more practical for building assistant applications.

5.2.2 Grammar-based parsing with Active

This section presents how traditional grammar-based chart parsers can be implemented using Active Ontologies.

Grammar-based parsing

The goal of a grammar parser is to process an expression using a well-defined set of grammar rules, resulting in a parse tree that represents a proof that the expression is a member of the acceptable utterances defined by the grammar. To illustrate our description, figure 4.4 shows a simplified English grammar in the *EBNF* form, inspired from the famous example from Russell and Norvig [69]:

Production Rule			Example
(1) S	⇒	NP VP | CP | S Conjunction S	I + feel a breeze stop here I feel a breeze + and + I smell a wumpus
(2) NP	⇒	Pronoun | Noun | Article Noun | Digit Digit | NP PP | NP RelClause	I pits the + wumpus 2 3 the wumpus + to the east the wumpus + that is smelly
(3) VP	⇒	Verb | VP NP | VP Adjective | VP PP | VP Adverb	stinks feel + a breeze is smelly turn + to the east go + ahead
(4) CP	⇒	CommandVerb | CP Adverb | CP NP	go turn + left grab + the wumpus
(5) PP	⇒	Preposition NP	to + the east
(6) RelClause	⇒	that VP	that + is smelly
(7) Noun	⇒	stench | breeze | glitter | nothing | wumpus | pit | pits | gold | east	
(8) Verb	⇒	is | see | smell | feel | stinks | go	
(9) CommandVerb	⇒	shoot | go | grab | carry | kill | turn | stop	
(10) Adjective	⇒	right | left | east | south | back | smelly	
(11) Adverb	⇒	here | there | nearby | ahead | right | left | east | south | back	
(12) Pronoun	⇒	me | you I | it	
(13) Name	⇒	John | Mary | Boston	
(14) Article	⇒	the | a | an	
(15) Preposition	⇒	to | in | on | near	
(16) Conjunction	⇒	and | or | but	
(17) digit	⇒	1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0	

Figure 5.5: Simplified English EBNF grammar

The first step of the parsing process is the *tokenization*, where the input sentence is decomposed into a sequence of tokens used to feed the parser. In our context, inputs utterances are sequences of word tokens. Given the grammar shown in figure 5.5, submitting the utterance *"I feel a breeze"* to the parser, should result in the construction of a tree-like structure shown in figure 5.6. With some grammars, ambiguous interpretations are possible, and in this case, the parser should return a list of all valid parses of an utterance. If the input utterance does not comply with the grammar, a parser should fail, indicating the utterance is not valid.

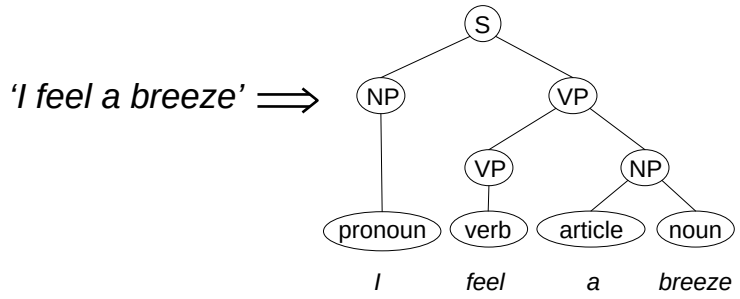


Figure 5.6: Parsing Tree

Chart parsing

Many techniques have been invented to implement grammar-based parsers. Some parsers are based on a recursive technique, mechanism that produces concise code in most procedural languages (C, C++ or Java). Other parsers take advantage of backtracking to be very elegantly implemented in declarative languages such as Prolog. In the case of Active, the most elegant and natural way to implement a grammar-based parser is to use a *chart parsing* technique [16].

Chart parsers use *dynamic programming* techniques where intermediate results are stored to be re-used when necessary. The technique consists of building small bits of partial knowledge, stored as *charts*, used as the foundation for further processing that will create more complex *charts*. This technique tends to be faster than more naive methods and naturally leverages the production-rule nature of the Active framework. A chart can be seen as a typed link that groups consecutive tokens. The simplest representation of a chart would have three values: a type and two indexes (the first and the last tokens of the chart). For instance, the sentence *"I feel a breeze"* would produce the charts sequence shown in figure 5.7.

Charts are incrementally generated using a bottom-up approach. At the first stage, low-level charts are created from the terminal rules of the grammar (rules 7 to 17 in figure 5.5). Once these basic charts are created, further processing steps combine the current knowledge of the situation to create more complex charts using higher rules of the grammar. The process continues until a chart that links the first token to the last token is generated, indicating that the utterance was successfully parsed. This incremental process where rules of the grammar incrementally fire in a sequence as new information is inferred fits very well into the Active model. The following section describes how this algorithm has been implemented for the Active framework.

Implementation

This section provides a high level definition of the Active-based chart parsing implementation by presenting the set of standardized facts and important rule

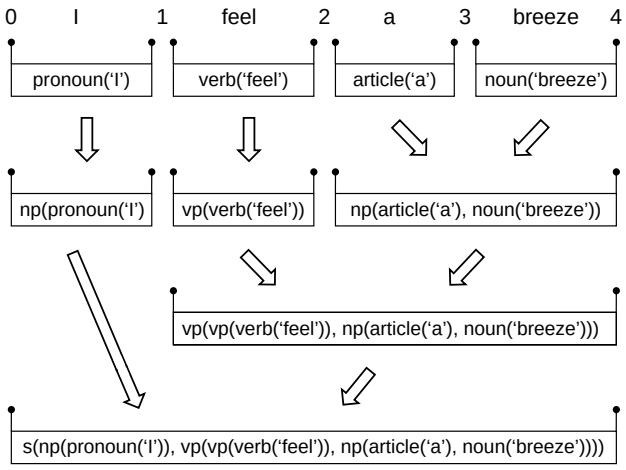


Figure 5.7: Charts creation sequence

definitions[1]. The Active implementation of a chart parser consists of converting the EBNF grammar into a set of concept and rules that communicate through standardized facts assertions.

Definition 5 Standardized facts for chart parsing

(a) recognized($source,sentence($sentence),$confidence)
(b) token($source, $value, $start_index, $end_index)
(c) nl_chart($chart_type,$value,$start_index, $end_index))
(d) voc($chart_type,$value))

The first step of the parsing process consists of asserting the utterance to process. To do so, the component that captured the inputs (a user interface, an incoming email or a speech recognizer) asserts a fact compatible with the definition shown in definition 5 (item a), where source indicates the name of the recognizer, sentence the utterance to parse and confidence, a number that ranges from 0 to 100, indicates the confidence in the recognition of the values to process. Note that in this example, only sentence is used, other values will be more relevant in more complex parsing techniques.

Next, a rule in charge of tokenizing takes over to chop-off incoming utterances into tokens. Token facts (see definition 5, item b) are created and asserted. The tokenizing rule, shown in figure 5.8, illustrates how Active takes advantage for the target language to manipulate the incoming sentence as Strings to create and assert tokens.

[1]For full implementation details please refer to the *Active Programmer's Guide* or the latest Active distribution.

Rule name : Tokenizer
Condition
Store.checkEvent(recognized($source,sentence($sentence),$confidence))
Action

var token_as_string=FactUtil.removeQuotes(TOKEN);
var local_words_array = token_as_string.split(' ');
for (i = 0; i<local_words_array.length; i++) {
  Store.createEvent(token(source, local_words_array[i], i, i+1));
}

Figure 5.8: Tokenizer rule

At the next step, rules in charge of implementing chart parsing take over. Charts to be created and manipulated are also represented as fact, standardized in definition 5, item c. A chart consists of four elements:

- chart_type. This is the name of the production rule that created the chart
- value: A list containing the token or the set of tokens that make up the chart
- start_index, end_index. The boundaries of the chart in the sentence

The Active-based chart parsing technique is based on two types of rules: rules for terminal nodes (low level rules) and rules for intermediate nodes (high level rules). First, rules for terminal nodes act as the first line of processing, where tokens are detected to create the first seminal batch of charts. The vocabulary of the application domain is stored as a set of vocabulary facts (definition 5, item d) that binds a word to a type. Low level rules use this knowledge and the value of incoming tokens to create the first group of charts. As an example of low level processing, figure 5.9 shows the rule in charge of detecting pronouns among asserted tokens. It uses a compound condition to bind tokens with known vocabulary using a variable ($word). If the rule is verified, is asserts a chart that combines the incoming word, its type and its indexes in the incoming utterance.

Rule name : Pronoun
Condition
Store.checkEvent(word($source,$word,$start_index,$next_index)) && Store.checkFact(voc(pronoun,$word))
Action
Store.writeFact(nl_char(pronoun, [word], start_index, next_index));

Figure 5.9: Rule to detect a leaf of type pronoun

Once the seminal set of charts has been created by low level rules, higher level processing takes over to incrementally build more complex charts. Higher level rules are encoded based on the grammar definition. Each non-terminal production rule (production rules 1 to 6 in figure 5.5) is converted into an Active concept with a unique rule set. Each alternative (vertical bar separated

clauses on the right side of grammar the rule) is converted into an Active rule. For instance, figure 5.10 shows the rule in charge of creating a PP chart out of the sequence of an NP chart and a Preposition chart (see production rule number 5 in figure 5.5).

Rule name : PrepositionalPhrases
Condition
Store.checkEvent(nl_chart(np, $npValues, $npStart, $npEnd)) && Store.checkFact(nl_chart(preposition, $prepValues, $prepStart, $npStart))
Action
Store.writeFact(nl_chart(pp, npValues + prepValues, prepStart, npEnd);

Figure 5.10: Rule to create a chart fact that represents a PrepositionalPhrase

Incrementally, rules fire to create more high level charts, until a chart covering all the tokens of the input utterance is generated, thus validating the utterance with the specified grammar.

Results and discussion

As the first simple application built with the Active framework, a chart-parser helped us validate our approach, tools and implementation. In addition, chart parsers are often used in natural language understanding applications, so this was a important AI methodology to include in the library of Active-supported techniques.

Since chart-parsing techniques do not require a recursive process and rely instead on an incremental production rule-based approach, it has been easily and elegantly implemented within the Active framework. As a complete basic system, this work consists of our first demonstration that validates our approach to implementing AI-based components with a single unified and user friendly toolkit. This prototype is the basis of the first Active methodology, where an EBNF grammar can be converted into a set of Active rules to implement a parser based on the chart-parsing algorithm. On the practical side, this application has been used as a test-bed to use and debug the main components of the Active software suite. Both the Active Server and the Active Editor (See figure 5.11) are working together as a programming environment in which a system has been designed, implemented and tested.

The technique was implemented to parse utterances compatible with the grammar shown on figure 5.5. The parsing of a sentence takes about 200 milliseconds. The system either produces the parsing tree for valid utterances, or an error message if the sentence cannot be validated against the grammar. To further test and validate our method, we also successfully implemented a chart-parser for simple mathematical expressions that support four basic operators (plus, minus, multiply and divide) and grouping with parenthesis.

After completion and initial tests, the chart-parser approach works as expected. Chart parsers are well suited for applications where strict parsing of well-formed expressions are the inputs, such as a mathematical expression to evaluate, or a piece of programming language code to compile. We are nevertheless far from the set of features to provide described in section 5.2.1. In

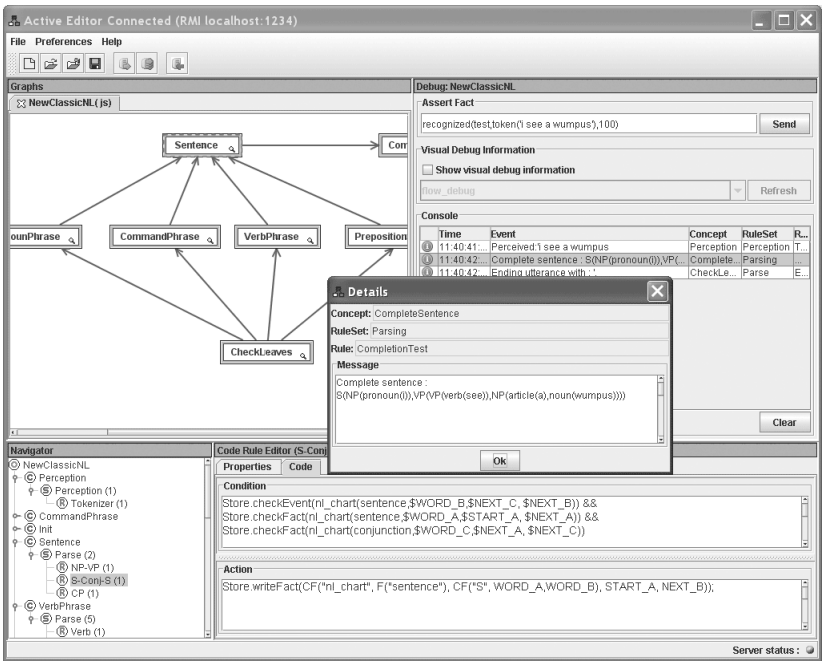


Figure 5.11: Charts parser in Active Editor

particular, human speech is often disfluent and agrammatical. Furthermore, while it is easy to create a simple subset of the grammar of English (or another natural language), it is an extremely difficult endeavor to capture all nuances of human language using grammar formalisms – researchers have dedicated decades to the effort without declaring absolute success. Finally, if the intent is for the intelligent assistant to be able to process utterances expressed in more than one human language, a developer must start from scratch implementing their application grammars for language – no or little reusability occurs in this large task. In an attempt to fill these gaps, the following section introduces a more cost-effective and robust language processing technique based on Active Semantic Networks.

5.2.3 Language processing with Active Semantic Networks

This section presents an innovative language processing technique that attempts to cover more features of the list presented in section 5.2.1. This approach fully leverages the philosophy of Active Ontologies, where semantic definition and processing are fused to implement AI techniques, language processing in this case. The result is a powerful methodology for rapidly constructing robust language understanding models suitable for intelligent assistant processing.

General approach

The technique is based on a two-step process. First, using concepts and relationships, an ontology-like model of the application domain is built. The second

step consists of injecting a thin layer of processing into specific concepts of the ontology to turn it into a language processing environment. The resulting Active Ontology is designed to react to incoming utterances and tokens, create and manage an information flow over its network of concepts to produce complete structures representing user intentions. To illustrate our description, let us consider the Active Ontology shown in figure 5.12. It is designed to parse utterances that express commands to retrieve information about movies and restaurants. Examples would be: *"find movies in san francisco"*, "*get italian restaurants"* or "*are any comedies playing in palo alto?"*.

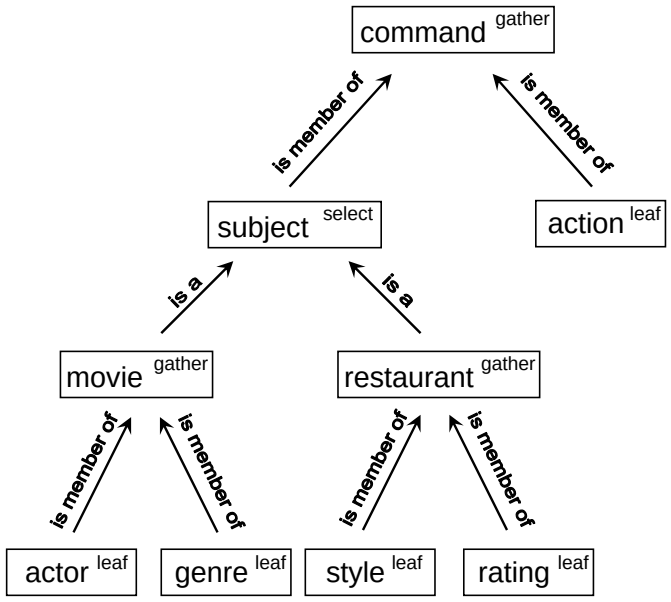


Figure 5.12: Simple semantic parsing graph

The application domain is modeled as an upside-down tree-like structure made out of connected nodes. When user utterances are submitted to the system for processing, each token (word) is injected into the tree from its bottom terminal leaves. Leaf concepts are specialized filters designed to sense and rate incoming events about their possible meaning. As tokens arrive, leaf nodes use some logic (i.e. the value of the token, its position in the sentence, its neighbors, regular expression-based pattern matching) to produce a *semantic rating* and communicate it to their parent nodes along the semantic network of relationships binding concepts. Connected to terminal leaf nodes, there are two types of non-terminal nodes that make up the rest of the semantic parsing tree: *gather* nodes and *select* nodes. First, gather nodes create structured objects made out of information coming from their children. For instance, in our example, a movie is made out of actors and a genre. The second type, select nodes, picks the single best rating coming from their children. For instance, the node subject of our example is in charge of choosing whether the user is talking about restaurants or movies. As members of the hierarchy, non-terminal nodes

report results to their own parent nodes.

Through this bottom up execution, input signals are incrementally assembled up the domain graph to produce a structured command at the root node. The following sections describe the behavior of these components in more details.

Semantic ratings Before further describing the elements of our language processing technique, let us define the basic piece of information flowing up the semantic graph: *semantic ratings*. A semantic rating is produced by any node of the semantic graph about the meaning of a group of tokens. The semantic rating is to be placed in a communication pipe (see section 5.1.1 on page 57) connecting a child node to its parents. Communication pipes follow the semantic relationships among the nodes of the application domain.

Definition 6 Pipe for communicating semantic ratings

```
pipe($child_name, $parent_name, $value, [conf($confidence)])
```

Definition 6 shows the structure of a communication pipe containing a semantic rating. It consists of the following components:

- child_name and parent name : The source and destination of the pipe
- value : the semantic value of the incoming token expressed as : semantic_value(token)
- A list of attributes to characterize the rating. Among possible attributes, the confidence of the rating is expressed as an integer number ranging from 0 to 100. Leaf nodes generate semantic ratings based on their confidence about the word being rated. In most cases, leaf nodes express either a full confidence (100) for words they successfully rate. On the other end, as explained later in this section, non-terminal nodes (*gather* and *select* nodes) compute an overall rating based on their number of children, individual ratings weighted by each individual child properties. Therefore, non-terminal nodes use a large spectrum of values, ranging from 0 to 100 when creating their semantic ratings.

For instance, a leaf node representing a movie genre reacts to words such as *comedies, thrillers* or *action movies* to notify its parent that the user may have expressed a sentence involving a type of movie. For the incoming utterance: *"find thrillers in san francisco"*, the **genre** node would create a claim of the form:

```
pipe(genre, movie, genre(thriller), [conf(100)])
```

The following sections describe in more details all the components and features of the Active semantic tree approach to language processing.

Basic components

Semantic networks consist of three types of components: terminal leaf node, non-terminal nodes and relationships. The following paragraphs describe these components how they are combined to implement a simple but robust language processing engine.

Terminal leaf nodes As described in the introduction, leaf nodes are the front line of processing, transforming incoming tokens into semantic ratings. To create their semantic ratings leaf nodes use multiple techniques.

- Vocabulary list. The easiest way for leaf nodes to make claims about incoming tokens is to compare them with a known vocabulary set. For instance, a list of movie types can be specified so that the *genre* leaf of our example can make decisions.
  Each entry of the vocabulary list consists of a semantic value and a list of possible synonyms. For instance, the semantic value *william* would have *william, bill, willy* as possible synonyms to check against.
  Finally, to support disfluencies our approach not only uses strict string comparison but also a less strict mechanism, based on the Levenhstein distance[47]. The Levenhstein distance between two strings is the minimum number of operations (character permutation, addition and removal) to perform on one string to become equivalent to the other.

- Prefix and Postfix. The vocabulary list technique described above may not be flexible enough to represent large partially known sets of data. For instance, the **city** node cannot specify an exhaustive list of cities, nor could the **actor** leaf node specify all actors in the world. Therefore, a technique using multiple tokens and their position in the incoming utterance has been designed for leaf nodes to create semantic ratings. A rating can be generated for any word that directly follows a list of known prefixes. For instance, the **city** leaf can fire for any token that follows the prefix *in*. Therefore, for any utterance of containing *"in paris"* or *"in miami"* the **city** leaf would claim the token *"paris"* or *"miami"* as a city without having to know about all cities in the world. For instance, if you wanted to parse *three dollars*, where three could be any number word, you might look for the suffix word *dollars*.

- Regular expression. Regular expressions can be used by leaf nodes to create their ratings about incoming tokens. For instance, one can imagine an application domain where users specify zipcodes. American zipcodes (or postal codes) are expressed as either a five-digit number or a combination of a five-digit number, followed by a dash (-) and a four-digit number. A leaf node in charge of detecting them would use a regular expression pattern of the form `\d{5}` to detect short forms of five digits or `\d{5}-\d{4}` for full zipcodes. The richness and flexibility of pattern matching techniques could also be used to detect numbers, e-mail addresses or any formatted type structures.

- Specialized leaf node. Finally, some leaf nodes encapsulate a specific hard coded logic to generate specialized ratings. For instance, many applications require time and date management (e.g. *"book me a french restaurant tomorrow night in Palo Alto")*. We chose to implement date/time logic as a plugin that generates ratings information in the form:

  ```
  date($year, $month, $day, $hour, $minute)
  ```

  The date node is capable of dealing with utterances involving information about hours and days, such as *"next Monday at 3 am"*, *"tomorrow morning"*.

All four techniques described above may apply to multiple consecutive tokens or words. Users can for instance specify an utterance that contains *"find movies in san francisco"* or *"get indian restaurants in new york city"*, where city names are longer than one token. Allowing multiple token processing brings interesting situations. For instance, let us consider the following utterances:

1) "find movies in san francisco with john wayne"
2) "get comedies in miami with charlie chaplin"
3) "find thrillers in new york"

Let us assume that the **city** node is configured to process up to two consecutive tokens. When analyzing the first utterance, the **city** leaf node looks for a single token or two tokens that follow the prefix *in*. It has two possibilities : *san* or in *san francisco*. To make the correct decision, leaf nodes are aware of the overall length of the incoming utterance and all prefixes used in the application domain. Tokens after the prefix are concatenated as candidates for the rating until a known prefix or the end of the utterance is reached. Therefore, in the utterance *1)*, *with* being a prefix defined by the *actor* leaf node, the *city* leaf nodes would use *city('san francisco')* as the value of its claim. Similarly, in utterance *2) city(miami)* would be the value produced. Finally, in example *3)* the end of the utterance is reached and *city('new york')* will be the value.

Finally, if no token triggers the creation of any rating for a leaf node, two cases are possible. First, if a default value has been specified for the leaf, it will be claimed with a hundred percent confidence by the leaf. In the absence of a default value, the leaf generates a *missing* rating with a zero confidence.

Non-terminal *gather* nodes Gather nodes are in charge of aggregating incoming ratings from their children to create structured semantic ratings. As part of the semantic parsing tree, they report their ratings to their own parents. Children nodes to be aggregated are connected to *gather* nodes with relationships of type *is member of*. As an example, the **movie** node gathers information from both its children: the **genre** and **actor** leaf nodes.

Whenever one of its children node writes a new information into its pipe, the gather node processing triggers. It starts by reading the latest information from its children nodes to update its own semantic rating. First, the value of the rating is updated by combining the values claimed by the children. A compound object representing all children contributions is created. For instance, in response to the utterance *"find comedies with john wayne"*, the *movie* node would generate the following value for its rating :

movie(actor('john wayne'), genre(comedy))

.

Once the aggregated value of the rating has been generated, its confidence needs to be updated as well. In the simplest case, it consists of the average of the children confidences. Assuming C the set of all children connected to a *gather* node, the confidence is:

$$GatherConfidence = \frac{\sum_{i \in C} ChildConfidence_i}{NbChildren \in C}$$

In the spirit of Active, *is member of* relationships not only connect children to the semantic super structure where they belong, but also provide information useful at processing time. In addition to connecting a source and a destination, relationships contain an attribute set used to further characterize their role in the semantic network. For instance, *is member of* relationships carry a *weight* attribute. The weight attribute is used by *gather* nodes to adjust the contribution of each child involved in the determination of their overall confidence. The confidence formula becomes:

$$GatherConfidence = \frac{\sum_{i \in C} ChildWeight_i * ChildConfidence_i}{\sum_{i \in C} ChildWeight_i}$$

Now that we have seen how complex structures get created through parsing, let us see how decisions are made to select parsing branches.

Non-terminal *select* nodes As the second type of non-terminal nodes, *select* nodes pick one of their children as their rating. In our example, the *subject* node is in charge of deciding whether the user is talking about movies or restaurants. It has therefore to choose which child will prevail and be communicated up as a semantic rating. Children candidates to be selected are connected to *select* nodes with relationships of type *is a*.

Whenever one of its children node writes a new semantic rating into its pipe, the *select* node processing triggers. Its goal is to get ratings from all its children to select the best one reflecting the intentions expressed by incoming utterances. The basic heuristic, which will be further refined, consists of picking the semantic rating that carries the highest confidence. The overall confidence of the rating made by our select node would be the confidence of the selected child.

For instance, in response to the utterance *"find comedies with john wayne"*, the *subject* node would select its child with the highest confidence, the **movie** node. It would then generate the following value for its rating :

```
subject(movie(actor('john wayne'), genre(comedy)))
```

Summary At this point of our description, all basic elements of our parsing technique based on semantic trees have been introduced. If the utterance *"find comedies with john wayne"* is submitted to our sample tree, the semantic ratings shown on table 5.1 would be produced.

Let us briefly describe the chronological chain of events. First, five leaves (**actor**, **genre**, **movie**, **style** and **action**) create and report their semantic ratings. Then, *gather* nodes **restaurant** and **movie** take over and produce their ratings. The **subject** node of type *select* picks the most likely candidate among its children (**movie** in our example) as its rating. Finally, the **command** node, of type *gather,* produces the overall result of the parsing.

Node name	Semantic rating
actor	pipe(actor, movie, actor('john wayne'), [conf(100)])
genre	pipe(genre, movie, genre(comedy), [conf(100)])
movie	pipe(movie, subject, movie(genre(comedy), actor('john wayne')), [conf(100)])
style	pipe(style, restaurant, style($), [conf(0)])
rating	pipe(rating, restaurant, rating($), [conf(0)])
restaurant	pipe(restaurant, subject, restaurant(style($), rating($)), [conf(0)])
subject	pipe(subject, command, subject(movie, subject, movie(genre(comedy), actor('john wayne'))), [conf(100)])
action	pipe(action, command, action(get), [conf(100)])
command	pipe(command, $, command(subject(movie, subject, movie(genre(comedy), actor('john wayne'))),action(get)), [conf(100)])

Table 5.1: Semantic Tree State

At this stage, we can already point some advantages of our approach over a more strict chart parser-like approach.

- The system is more robust to disfluencies and recognition errors. Interjections, repetitions or misrecognized words are gracefully ignored or claimed with low confidence, allowing the system produce partial results and give a chance to the user to either repeat or more clearly express her or his needs.
- Often, natural language applications create parsers based on the syntax and semantics of each language they need to support, and then create linguistic definitions for words in the domain of use. This requires specialized linguists and a large development time to map domain knowledge into the linguistic model. By contrast, in Active, a domain is first modeled in terms of concepts and then a relatively light layer of language is added to it. Most of the domain modeling is completely reusable across languages, and hence adding a new language to the application is achieved much more quickly.

Based on these basic components and preliminary results, the following sections describe further improvements and added features to our innovative parsing technique.

Helper leaves

This paragraph introduces the notion of leaf nodes providing auxiliary information. In some cases, words expressed by users provide information about which part of the parsing tree should be given more confidence, without contributing to the data structure to be created. For instance, an utterance containing the word *"movies"* indicates the topic the user is talking about, without providing any specific information such as the genre nor the name of an actor. The word *"movies"* should only bring more confidence the **movie** node without being part of its rating.

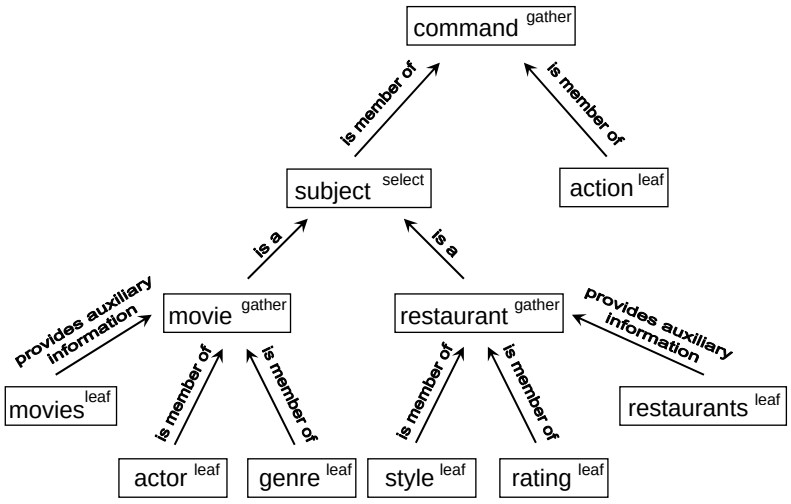


Figure 5.13: Semantic parsing tree with helpers

To implement this behavior, leaf node can be connected to a *gather* node using *a provide auxiliary* information relationship. Such children will contribute to the overall confidence of their parent *gather* node in a similar way as their siblings connected as *is member of* children. The only difference between the two types of children is that *provide auxiliary information* leaf nodes will not be part of the structure created as part of the rating created by the parent *gather* node.

To illustrate this concept, figure 5.13 shows an extended version of our sample Active Ontology, where two new *leaf* nodes have been added. The nodes **restaurants** and **movies** will be in charge of detecting the words *restaurants, restaurant* and *movies*, *films*.

Therefore, the sentence: "find movies in palo alto" will produce the final rating:

```
pipe(command, $, command(subject(movie, subject, movie(genre($),
actor($))),action(get)), [conf(100)])
```

Event if no movie type nor actors have been specified, the **movies** node quietly contributed to the confidence of the **movie** node.

Mandatory or optional children

Some nodes of the semantic parsing tree represent pieces of information that are required for the overall understanding of utterances. Another set of nodes are optional and provide additional non-crucial data about the user intentions. Through a simple example, this section shows how the optional or mandatory status of nodes influence confidence ratings and are used to build additional parsing information.

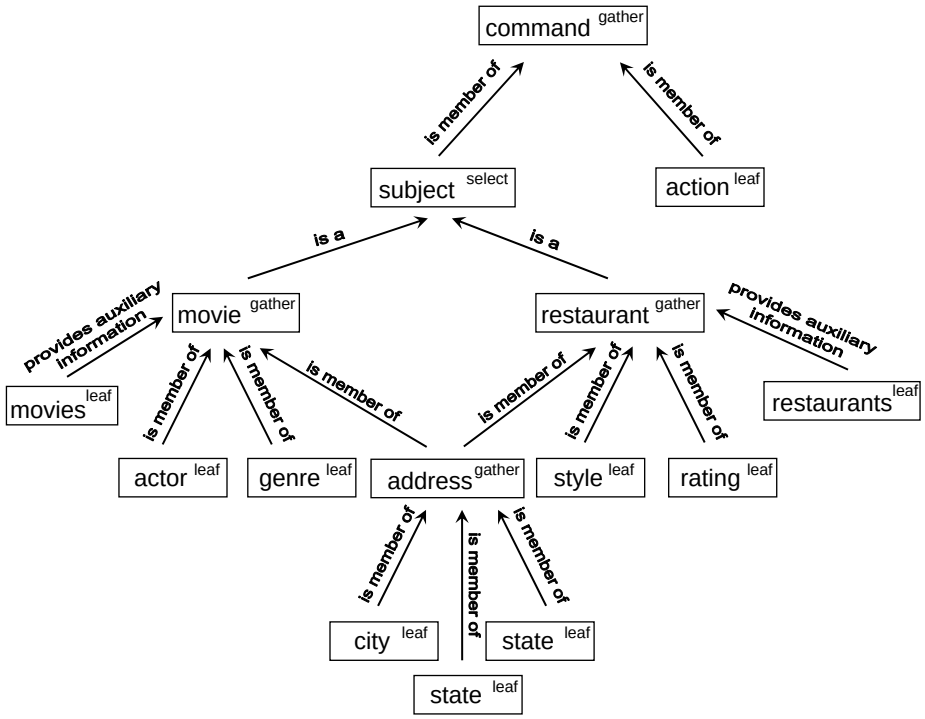


Figure 5.14: Semantic parsing tree with address

To illustrate this concept, let us extend the domain of our sample application. Figure 5.14 shows a semantic network enhanced with a new **address** node, of type *gather*, made out of a **city**, **state** and **zipcode**. Before developing more on the concept of mandatory and optional elements, it is interesting to describe two features of the **address** node. First, note that its children implement different techniques to produce ratings out of incoming tokens. The **city** leaf node uses a prefixed technique, where any word following the prefix *in* would be rated as a city name. The **zipcode** leaf node uses a regular expression to detect zipcodes among tokens that are injected into the system. Finally, the **state** node uses a vocabulary list containing the list of fifty US states and their synonyms. Secondly, the **address** node is connected to two parents (both the **movie** and **restaurant** nodes), which is a valid and useful feature of semantic parsing trees. In this configuration, the **address** node creates similar ratings reported through to two separate communication pipes.

Now that the **address** node has been introduced, let us focus on optional and mandatory elements. As an application designed to retrieve information about hotels and restaurants, the system shown in our example should report an error if the **address** structure is not specified. To model this behavior in the Active context we use a relationship attribute. The *is member of* relationship type, that connects *gather* nodes to their children, carry a boolean attribute named *is mandatory*. If checked, the attribute indicates that the child node has to be provided for the structure to be valid. If not, the child is considered optional

and not crucial for expression the intention of the user. This information is used in two aspects of our language parsing technique: *gather* nodes confidence computation and parsing information lists.

Confidence computation of *gather* nodes This paragraph explains how contributions vary between optional and mandatory children of a *gather* node. As previously described, *gather* nodes create nested structures by assembling values of their children. The overall confidence of their claims is computed by combining their children individual confidence ratings. We have also seen that the contribution of some children can be weighted to modulate their overall impact on the global rating. Similarly, the optional or mandatory status of children influences their contribution to the overall confidence rating generated by a *gather* node.

Mandatory children are required information for the parser to produce a valid command, whereas optional children provide additional, non-critical information. Therefore, a missing optional child should be less penalizing than a missing mandatory child to the overall confidence. The confidence formula to consider can be expressed as follows.
The contribution of mandatory children is:

$$C1 = \sum_{i \in C} ChildWeight_i * ChildConf_i \parallel Child_i isMandatory$$

The contribution of optional children is:

$$C2 = \sum_{i \in C} ChildWeight_i * ChildConf_i \parallel Child_i isOptional, ChildConf_i > 0$$

The overall confidence of the gather node could then be expressed as:

$$GatherConfidence = \frac{C1+C2}{\sum_{i \in Cm} ChildWeight_i}$$

The approach can be further refined. Even if they are tagged optional, missing children should incur some penalty to the confidence rating of a *gather* node. Our formula becomes:

$$C1 = \sum_{i \in C} ChildWeight_i * ChildConf_i \parallel Child_i isMandatory$$

The contribution of optional children that have been detected is:

$$C2 = \sum_{i \in C} ChildWeight_i * ChildConf_i \parallel$$
$$Child_i isOptional, ChildConf_i > 0$$

The penalty incurred by optional children that have not been detected is:

$$C3 = Mp * \sum_{i \in C} \parallel Child_i isOptional, ChildConf_i < 0$$

Where *Mp* is the penalty for a non contributing child. The overall confidence of the gather node could then be expressed as:

$$GatherConfidence = \frac{C1+C2+C3}{\sum_{i \in Cm} ChildWeight_i}$$

Now that we have explained how mandatory and optional attributes influence confidence ratings, we will see how they contribute to the creation of additional parsing information.

Parsing information lists In addition to providing a value and a confidence in their ratings, nodes provide extra information about the parsing process. The data is organized into three lists:

- Errors list. A list of error messages that prevented the parsing from succeeding. As contributors, *gather* nodes insert to the *error list* the names of non-specified mandatory children (mandatory children whose confidence is rated as zero). This information is typically used to inform the user about missing pieces of information.
- Suggestions list. This list provides non critical information gathered through the parsing process. For instance, *gather* nodes would add to the suggestion list all optional children that were not specified. Using this list, applications could inform about what could be specified, thus helping users to learn about their options and what can be expressed.
- History list. Each rating provides the list of all the nodes visited before its contribution. This history list keeps track of the processing path over the tree is useful at different levels.
  First, when a *select* node picks one of its children as its claim, both the *errors* and *suggestions* lists need to be updated to only reflect the contributions of the branch that has been selected. For instance, if a user expresses a request about restaurants, all contributions to the lists made by the *movie* branch become irrelevant.
  This information is also used for debugging, helping Active programmers track down which nodes contributed to final processing results.

Cardinality

As described above, relationships determine if a child node is optional or mandatory. Similarly, relationships are used to define the *cardinality* of children nodes. In our case, the cardinality is a boolean attribute of a relationship that specifies if the relationships holds a single unique rating or accumulate multiple ratings over a dialog. For instance, to model that a movie consists of a set of actors, the relationship *is member of* that connects the leaf **actor** to the *gather* node **movie** in figure 5.14 bears a positive cardinality.

Context management

An important feature of the semantic tree parsing technique is context building and management. Semantic ratings are persisted in their communication channels over multiple utterances, allowing users to update, complete or reset the

context of their *dialog*. We define a dialog as a sequence of utterances expressed by a user to communicate intentions or answer questions. We define a *context,* or a *session,* as a set of data gathered about a user dialog over multiple utterances. In our case, a context consists of ratings stored in communication pipes that connect the nodes of the semantic tree.

Considering our example shown in figure 5.14, let us examine what happens when the following sequence of utterances is submitted to the tree:

1) "find comedies in san francisco"
2) "no, get me thrillers instead"
3) "ughhh, no, i will actually be in palo alto tonight"

When the first utterance is submitted, leaf nodes **genre**, **city** and **actor** generate ratings, helping their parent nodes to create their own ratings up the tree to finally produce a global rating:

```
pipe(command, $, command(subject(address(city('san francisco'), zipcode($),
state($)), movie(genre(comedy), actor($))),action(get)), [conf(100)])
```

When the second utterance is asserted, only the **genre** leaf node reacts to update its rating with *thriller* instead of *comedy*. An upward sequence of partial updates propagates up the tree affecting the **genre**, **movie**, **subject** and **command** nodes. The final result is updated with:

```
pipe(command, $, command(subject(address(city('san francisco'), zipcode($),
state($)), movie(genre(thriller), actor($))),action(get)), [conf(100)])
```

Similarly, as the third utterance is asserted, a partial update initiated by the **city** node triggers an upward cascade of updates to generate:

```
pipe(command, $, command(subject(address(city('palo alto'), zipcode($),
state($)), movie(genre(thriller), actor($))),action(get)), [conf(100)])
```

This example illustrates two features of our semantic tree parsing techniques. First, as already shown previously, the technique gracefully ignores disfluencies such as *"ughh, no"* and skips unknown words such as *"i will actually", "instead"* or "*tonight"*. Secondly, the system supports partial utterances through a dialog context. In our example, the second utterance only brings one relevant piece of information, the type of movie to look for. Leveraging the session context that is incrementally built and updated as utterances come in, the global parsing result still contains the full information (city name and movie type).

Disambiguation

Using a context incrementally built over multiple utterances is a powerful feature that brings many benefits and raises interesting problems, one of them being *disambiguation*. We have seen in previous sections how ratings are remembered and updated as user utterances arrive to build up a session context. In the

case of *select* nodes, as utterances are processed and stored, some children may express ratings baring the same confidence, leading to an ambiguous situation.

To illustrate our presentation, let us consider the following sequence of utterances sent to the network shown in figure 5.14:

1) "find movies in san francisco"
2) "good, now get me a restaurant"

When the first utterance is submitted, all is well and the leaf nodes **genre**, **city** and **actor** generate and propagate initial ratings, leading to the following overall rating:

```
pipe(command, $, command(subject(address(city('san francisco'), zipcode($),
state($)), movie(genre(comedy), actor($))),action(get)), [conf(100)])
```

Let us examine what happens when the second utterance arrives. Both **restaurants** and **style** leaf nodes react and generate ratings, leading the **restaurant** node to create a rating as well. Since the previous result of the **movie** node is still stored in its pipe as part of the context, both children of the **subject** node have ratings. Worse, both show the same hundred-percent confidence. Which one to pick? There is an ambiguous case that needs to be solved by the *select* node. The following paragraphs introduce disambiguation techniques designed and implemented in our Active-based language processing technique.

Simple strategy : use utterance context The first strategy used by *select* nodes to disambiguate children consists of testing the age of their semantic ratings. The idea is to choose the child about which the user talked most recently.

The age of a rating is given by the age of the utterance that triggered its creation. Within a dialog, each incoming utterance generates a burst of events leading to the creation or update of ratings over the semantic tree. As it is processed, each utterance is tagged with an incremental counter used to order them over time. The counter is also used to tag ratings, therefore the smaller the counter, the *older* a rating is. In the case of disambiguation, if more than one child have the same confidence, picking the most recent rating ensures that we select the most recent topic of the dialog.

In our example:

1) "find movies in san francisco"
2) "good, now get me a restaurant"
3) "i am interested in comedies"

As the second utterance arrives, even if both the **movie** and **restaurant** are rated with the same confidence, the **restaurant** child is selected because it is the youngest.

As a dialog unfolds, the role of *select* nodes is similar to a *semantic switch*, deciding which sub-branch is the topic of interest expressed by the user. For instance, in our example as utterance *3)* is submitted, the topic swings back to the movie branch for two reasons. First, its confidence would be higher that restaurants (two children **movies** and **genre** have a confidence greater then

zero). Secondly, it is younger than the rating coming out of the restaurant branch.

To ensure that the age information is carried over through the parsing process, both *gather* and *select* nodes have to integrate it into their ratings. Ratings generated by *gather* nodes are tagged with the age of the youngest rating coming from their children. Ratings that come out of *select* nodes carry the age of the selected child.

More complex strategies There are situations where multiple children of a *select* node have both the same confidence ratings and the same age, preventing the previously described age-based technique from disambiguating. First, we describe when such situations occur, then we provide a list of techniques and strategies developed to resolve the ambiguity.

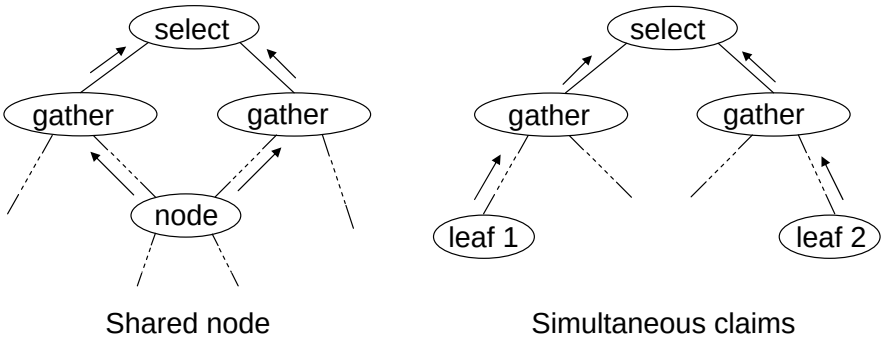


Figure 5.15: Ambiguous networks

Two semantic network configurations lead to such situations (see figure 5.15). First, contributors to a *select* node may have a common child at a lower level in the network. In such case, any rating generated by the common child will update the age of ratings all the way up to the children feeding the *select* node, this defeating the age-based disambiguation technique. As an example of this scenario, let us consider the following sequence of utterances:

Dialog 1:
1) "find movies in san francisco"
2) "book me a restaurant as well"
3) "actually, I'll be in sunnyvale"

As shown in previous sections, utterances *1)* and *2)* do not lead to any problems. The third utterance triggers a reaction from the **city** leaf node, cascading to the **address** gather node, shared by both the **restaurant** and **movie** branches. When ratings reach the select node, they have the same age and may possibly carry the same confidence.

The second configuration involves multiple leaf nodes. If an utterance submitted to the semantic tree triggers reactions from different leaf nodes, contributing to branches ultimately connected to the same *select* node, some ratings will have the same age and potentially the same confidence. Let us describe

a configuration leading to such situation. In the semantic tree of figure 5.14, both the movie **genre** and restaurant **style** leaf nodes have the word *french* in their vocabulary set. The intention is to allow users to talk about *french movies* or *french restaurants*. Now, let us analyze the following sequence of utterances:

Dialog 2:
1) "find movies in san francisco"
2) "book me a restaurant as well"
3) "actually, i am interested by a french one"

Utterances *1)* and *2)* do not lead to any problems. The word *french* being shared by two leaves (**genre** and **style**), utterance *3)* is more interesting. Both leaves would create a rating, that will percolate up all the way to the top select node, where multiple children may claim ratings with similar confidence and ages.

To solve ambiguous cases, we have developed the following set of strategies.

- **Use last choice**. In this strategy, when several children cannot be disambiguated, a *select* node repeats its last selection. In the above examples, the third utterances would be understood as looking for restaurants in Sunnyvale for *Dialog 1* and booking a French restaurant in San Francisco for *Dialog 2*.

- **Ask the user**. Another technique implemented by *select* nodes is do make no decision. The node would simply contribute to the error list (see section 5.2.3 on page 81) to let the application know that the parsing could not be completed because of an ambiguous case. Typically, the application would notify the user and ask for more information to help the system disambiguate.

- **Leverage design**. Techniques to prevent ambiguous cases can be used at the design level of semantic networks. For instance, *is a* and *is member of* relationships have a boolean *affect parent age* attribute. If unchecked, ratings communicated over the relationship would only contribute to the confidence, not the age of the overall parent rating. After their processing phase, *select* and *gather* nodes will ignore the contribution of children connected through such relationships when computing the age of their ratings. This feature is useful when a node is connected to multiple branches that lead to a *select* node. In the example shown in *Dialog 1,* the **address** node is shared by two branches (**movie** and **restaurant**) connected to a *select* node (**subject**). If the two relationships that connect **address** to its parents carry an unchecked *affect parent age*, specifying a new city (such as in utterance *3)* of *dialog 2* above) would prevent the movies and restaurant from having the age, therefore allow the **subject** node to pick a child using the age-based technique.

- **Delegate**. The last technique consists of delegating the decision to an external component. A select node that cannot make a decision can use the invocation mechanism described in section 5.1.2 on page 59 to call out for help and hold its decision until a result comes back. Using this loose connection allows Active developer to externalize the disambiguation process into a separate extension or Active Ontology.

Semantic validation

In addition to providing a solution to the problem of ambiguities, the Active-based language processor includes a technique to deal with *semantic validation*. Semantic validation consists of checking, as early as possible in the process, if data structures constructed over the semantic network could be completed and conveys a meaningful content. Using the invocation mechanism described in section 5.1.2 on page 59, *gather* nodes can ask external providers, or *advisers*, to complete missing fields and validate their ratings before communicating them to their parents.

Data structure completion Nodes of type *gather* provide the ability to automatically fill missing values from partial utterances. For instance, the *gather* node **address** consists of a **zipcode**, a **city** name and a **state**. As discussed earlier, each leaf node is instrumented with specialized processing bits to create ratings out of incoming utterances. For instance, when processing the utterance :

"find movies in Miami"

the **address** nodes produces:

```
address(city('miami'), zipcode($), state($))
```

Before communicating its rating to its parent, the *gather* node can ask a specialized processing element, that uses database of all know US cities, to fill missing slots. In addition to filling missing slots, the adviser can access the suggestions and errors information lists (see section 5.2.3 on page 81) to provide additional information to be communicated to the user. For instance, the above example specifies the city of Miami. Since there are multiple cities named Miami in the US, the adviser will pick the most likely city in its list to automatically fill the zipcode and state slots. It will also populate the parsing suggestion list with details about other instances of Miami.

Meaningful content In addition of filling out missing slots, advisers also validate the information contained by *gather* node ratings. For instance, when processing the utterance :

"find movies in Miami, Alaska"

A *gather* node may ask an adviser to verify if the data provided is valid. The behavior to correct the information is encoded in the adviser. For instance, in the example above, an adviser would pick the most likely state for the city of Miami, and use the suggestion list to provide the user with relevant suggestions about possible states with cities named Miami.

Implementation

All features previously described about Active-based parsing with semantic networks have been implemented. Figure 5.16 shows the language processing example presented in this chapter in the Active Editor. This section first introduces fact definitions used to implement the information flowing up the semantic tree.

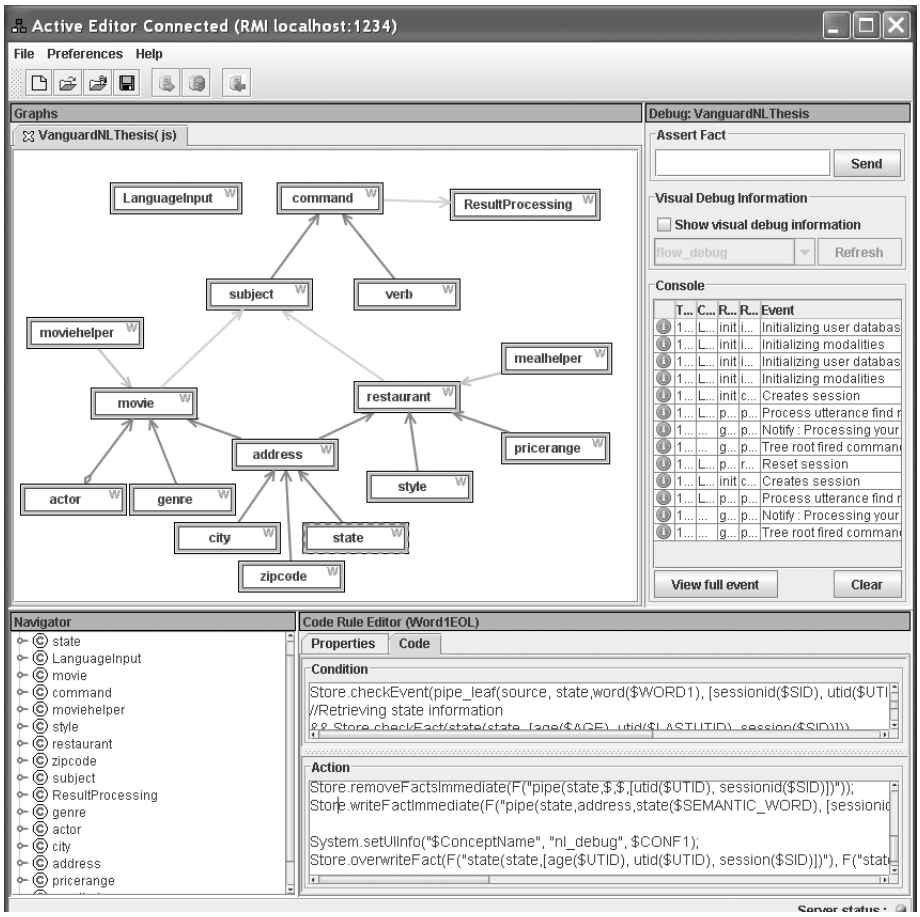


Figure 5.16: Language processing sample in Active Editor

Follows a high level description of major rules and concepts designed to be the infrastructure of parsing semantic networks. We then describe a collection of Active Editor wizards that provide high-level user interaction and automatically generate Active rules and concepts. Finally, we provide more details on the Active Server extension designed to facilitate and improve performances of language parsing.

Fact definitions First, we give a list of four relevant fact definitions used to model the data structures for the parsing.

First, formatted facts are used to report incoming utterances expressed by the user. Definition 7 (item *a*) describes the elements of the structure as being a tuple with two elements: the utterance to parse and a list of attributes. The main variables are:

- $sentence: The utterance to parse. (i.e. *"find movies in palo alto"*)
- $session: Session identifier giving the context of the utterance to process.

- $user: Identifier of the user who expressed the utterance.
- $modality: The type of sensor that created the utterance. (i.e speech recognizer, instant message or email)
- $conf: Confidence of the source (expressed by modality) about the accuracy of the sensed utterance. (Ranging from *0* to *100*)
- $action: What to do once the parsing tree has generated a structure out of the utterance. Current possibilities are: *reset* to erase the session context and restart a fresh dialog; *parse* to stop after parsing and notify the user with raw parsing results; *act* to pass the parsed structure to another Active Ontology in charge of actually undertaking actions expressed by the user.

Standardizing facts used to report utterances allow the Active Ontology in charge of language processing to process all utterances similarly, regardless of their origin.

A second type of standardized facts hold nodes and leaves semantic ratings. All information about the current parsing state and context information is kept in a collection of semantic ratings produced by the hierarchy of nodes. As seen in definition 6 on page 73, semantic ratings are represented as communication channels carrying a value and a set of attributes from a source node to a destination node. To implement complex features and behaviors detailed in previous sections, more attributes have been added to the list provided in definition 6. In addition to a confidence level, the attribute list contains the age of the rating, its history (a list that of all nodes that contributed to its value) and a session identifier. The session identifier is a unique value used to allow multiple simultaneous sessions, driven by different users, to be executed over the same semantic network.

Definition 7 Semantic tree fact definitions

a) input($sentence, [session($session), modality($modality), action($action), confidence($conf), user($user)]))
b) pipe($child_name, $parent_name, $value, $attribute_list)
c) voc($semantic_value, $owner, $values)
d) session($id,[user($user),word_index($windex),utterance_index($uindex)])

Third, leaf nodes use a vocabulary set and a list of separators stored in vocabulary facts. Vocabulary facts (see item *c* in definition 7) are triplets bindings a semantic value, the node that owns it and a list of possible token values. For instance, the movie genre *drama* owned by the node **genre** would be represented as:

```
voc(drama,genre,[drama,dramas],[])
```

Note that this approach where the semantic rating generated by a leaf node is decoupled from the definition of the syntactic values allows us to extend our system to multiple languages. For instance, extending the movie genre to support additional languages:

```
voc(drama,genre,[drame, drames, drama,dramas],[])
```

Finally, specialized facts are used to store a user session (see item *d* definition 7). Each user session is unique identified and carries information about the user name, the number of tokens and the number of utterances processed within the session.

Rules and Concepts This section shows how Active application design has been leveraged to express conditions and actions to turn a semantic model into an execution environment.

Several types of concepts have been designed. As presented earlier, each node of the semantic tree is an Active concept, instrumented with specific rules to implement its role (*leaf*, *select* or *gather* node). In addition, two specialized types of concepts have been created for house keeping tasks. First, similarly to the chart parser technique presented in section 5.2.2, a specialized concept has been designed to process incoming utterances. Its role is to tokenize arriving sentences and inject individual words into the leaves of the semantic tree through dedicated communication channels. Secondly, at the top of the semantic tree a specialized concept is in charge of processing the final semantic rating. It checks the final rating for errors and dispatched it to another Active Ontology for processing and execution.

Each type of concept is fitted with processing rules. Rules use facts shown in definition 7 to react and contribute to the processing as words come into the system. To briefly illustrate these techniques, let us look at two simple examples.

Rule name : Leaf (movie genre node)

Condition

```
Store.checkEvent(pipe_leaf(source, genre, word($WORD),
  [sessionid($SID), utid($UTID), index($INDEX),
  user($USERID),confidence($CONF)]))
&& Store.checkFact(voc($SEMANTIC_WORD, genre,['$WORD'],$))
```

Action

```
Store.removeFacts(F("pipe(genre,$,$,[sessionid($SID)])"));
Store.writeFact(pipe(genre,movie,genre($SEMANTIC_WORD),
  [sessionid($SID), utid($UTID), origin(nl), index($INDEX),user($USERID),
  history([item($ConceptName,[word($WORD),
  type(leaf), value($SEMANTIC_WORD), utid($UTID)])]), confidence($CONF)]));
```

Figure 5.17: Leaf node rule

First, figure 5.17 shows how the leaf in charge of claiming movie genres expresses a rule in charge of testing if an incoming token belongs to its vocabulary set. Note how the condition uses a compound rule to match the incoming token with an element of the vocabulary set using the WORD variable. Also note that the variable SID is used to pick ratings only within the space of a unique user session, allowing a semantic network to host multiple sessions simultaneously.

As a second example, let us examine in figure 5.18 how the restaurant node gathers information coming from its children. The condition is a compound expression whose first member is a checkEvent acting as a guard. The condition triggers when any source writes in to pipe whose destination is **restaurant**. On

Rule name : Gather (movie node)

Condition

```
Store.checkEvent(pipe($,restaurant,$,
  [origin(nl), sessionid($SID), history($), utid($), user($USERID)])) &&
(
  Store.checkFact(pipe(address,restaurant,$address,
      [sessionid($SID), history($HISTORY_address), utid($UTID_address),
      index($INDEX_address), user($USERID), confidence($CONF_address)]))
  && Store.checkFact(pipe(mealhelper,restaurant,$mealhelper,
      [sessionid($SID), history($HISTORY_mealhelper), utid($UTID_mealhelper),
      index($INDEX_mealhelper), user($USERID), confidence($CONF_mealhelper)]))
  && Store.checkFact(pipe(pricerange,restaurant,$pricerange,
      [sessionid($SID), history($HISTORY_pricerange), utid($UTID_pricerange),
      index($INDEX_pricerange), user($USERID), confidence($CONF_pricerange)]))
  && Store.checkFact(pipe(style,restaurant,$style,
      [sessionid($SID), history($HISTORY_style), utid($UTID_style), origin($),
      index($INDEX_style), user($USERID), confidence($CONF_style)]))
)
```

Action

```
//Updates history
var local_hist = F("[item($ConceptName,[type(gather)])]");
local_hist.addList(HISTORY_address);
local_hist.addList(HISTORY_mealhelper);
local_hist.addList(HISTORY_pricerange);
local_hist.addList(HISTORY_style);
// Computes the semantic rating
NL.gatherInit();
NL.gatherAddCandidate(address,CONF_address, 1, UTID_address, INDEX_address,
0, false, false);
NL.gatherAddCandidate(mealhelper,CONF_mealhelper, 1, UTID_mealhelper,
INDEX_mealhelper, 0, true, false);
NL.gatherAddCandidate(pricerange,CONF_pricerange, 1, UTID_pricerange,
INDEX_pricerange, 0, true, false);
NL.gatherAddCandidate(style,CONF_style, 1, UTID_style, INDEX_style, 0, true,
false);
// Computes results
NL.gatherProcess(true, true);
var local_score = NL.gatherGetScore();
var local_ut_index = NL.gatherGetIndex();
// Creates the resulting fact
var local_result = CF("restaurant");
local_result.addElement(address);
local_result.addElement(pricerange);
local_result.addElement(style);
// Notifies parent(s) with results
Store.overwriteFact(F("pipe(restaurant,subject,"+local_result+",
  [sessionid($SID), history("+local_hist+"), utid("+ local_ut_index + "),
  origin(nl), index(0), confidence("+local_score+"),user($USERID)])"),
  F("pipe(restaurant, subject,$,[sessionid($SID), utid($), origin(nl),
user($USERID)])"));
```

Figure 5.18: Gather node rule

the action part of the rule, first the history is updated to contain all children as having contributed. Secondly, an Active Server extension (see section 4.3.2 on page 53) encapsulates as a pre compiled library the logic that computes the overall confidence of the *gather* node. Finally, as all the elements of the rating to produce are created, the node updates its own channel to notify its parent about its new contribution.

Finally, a wizard has been developed to encapsulate the logic of a leaf node in charge of dealing with dates.

Wizards and Extensions This paragraph presents extensions to the Active software suite developed specifically for semantic network-based language processing.

Figure 5.19: Leaf node Wizard

First, a collection of Active Editor wizards have been developed to automatically create concepts and rules that instrument the nodes of the semantic network. For instance, to add a leaf node from the Active Editor, developers right-click on the *graph pane* area of the Active Editor to select among the list of available wizards. Under *Language Processing*, they can select *Insert NL Leaf* to pop up the wizard shown in figure 5.19. After asking for the name of the node to create, the wizard invites programmers to select and configure the leaf node logic. As the Active programmer hits the *Finish* button, the wizard analyzes the information to automatically create the concept, rulesets and rules

that implement the logic of the node. Wizards can be activated any time to update the information and re-generate the attached rules. Note that wizards also automatically update the Active code when relationships that connect nodes of the semantic tree are modified. In addition to the leaf node, Wizards have been developed to manage *date*, *gather* and *select* nodes. Finally, two specialized wizards can be used to create pre-processing (tokenizer) and post-processing (connect to the top of the tree to manage final ratings) concepts that provide house-keeping tasks around the semantic parsing tree. Using this collection of Active Editor wizards, a full blown Active-based language processing application can be designed, implemented and deployed by drag-and-drop graphic modeling and wizard-based configuration steps.

Secondly, on the server side, a language processing extension has been designed to encapsulate some functionalities and expose them for use in both rule conditions and actions. For instance, the action code snippet shown in figure 5.18 uses the NL extension to compute the overall confidence of the rating a gather node produces. As long as the language processing extension is deployed, the functions NL.gatherInit, NL.gatherAddCandidate, NL.gatherProcess and NL.gatherGetScore can be used anywhere in any rule action code. Active Server extensions also contribute to enhance the condition part of rules. Extensions can be written to not only use unification to look for matching facts in the local store but also use the same mechanism to search other data sources. An extension was written to connect to any JDBC compliant database to provide Active applications with access to large data sets. For instance, in the language processing domain, large vocabulary sets can be stored in RDBMs.

Language processing test console A tool to test Active Ontologies implementing semantic networks has been designed. The *Language Processing Test Console* has two main functionalities : interactive test and automated regression test. The interactive test mode (see figure 5.20), allows Active Developers

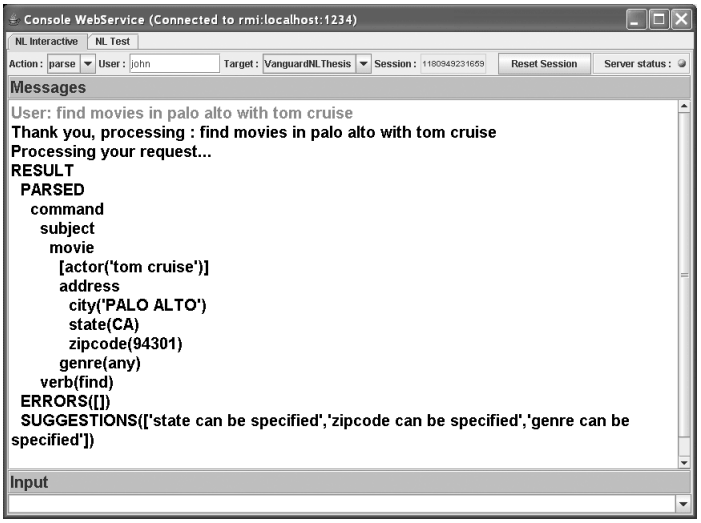


Figure 5.20: Language Processing Test Console - Interactive

to interact with semantic networks under development to quickly test features and parsing techniques. A second use of the console (see figure 5.21) offers the ability to run regression tests against a semantic network. A regression test is a ordered sequence of entries. There are two types of entries : *utterances* and *reset*. Each utterance entry contains an index, a utterance and an expected parsing result. Reset entries are used to clear the parsing context. Running the test consists of sequentially executing all entries of test. When a reset entry is reached, the tool asks the semantic network to reset its state. When an utterance entry is reached, its utterance (set of words) is sent to the semantic network for processing. Then, the console waits for the parsing results and compares it with the expected parsing result store in the test entry. As the test unfolds, response time and status are collected and graphically shown to the Active programmer. Regression tests are XML files that can be easily created, loaded into the test console and run against a semantic network.

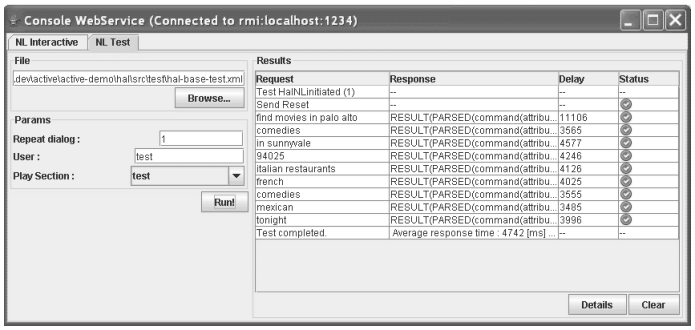


Figure 5.21: Language Processing Test Console - Regression Tests

Results

This section summarizes the results of the semantic network approach to language processing. First, an overview of what was achieved presents the main characteristics of the technique. Follows, a discussion compares and contrasts our technique with interesting and relevant work in the field of language processing. Note that a performance evaluation in terms of response time of this technique is presented in section 7.3.2 on page 182.

Features As the second Active-based methodology presented in this document, the semantic network approach to language processing is more complete than the simple chart parser approach introduced in section 5.2.2. The technique not only defines how to implement an AI component within the Active framework, but it also comes with a collection of Active Editor wizards and Active Server extensions. The main features of our implementation are:

- Ease of programming. Using Active Editor Wizards, it is possible to create a language processing application without writing any code. Programmers model the application domain of their application by creating nodes using the palette of Wizards for language processing and graphically connect them with relationships.

- Multiple detection techniques. Leaf nodes in charge of detecting words and generate seminal ratings offer multiple detection techniques including testing values against a vocabulary (either locally or remotely from a database), using regular expressions or separators (prefixes and postfixes).
- Robust parsing. The approach described implements a robust parser that ignores unknown words that may come from disfluencies or erroneous speech or handwriting recognition.
- Context management. A dialog context is incrementally built and kept in the semantic tree. Users can issue multiple incomplete utterances to drive a dialog, provide more details or switch the topic of interest.
- Built-in suggestion and error lists. As the parser integrates information from the user, it generates lists to inform about its processing state. Two lists are generated: an error list to inform about missing mandatory elements and a suggestion list providing information about what could be specified.
- Built-in disambiguation. A built in disambiguation technique leverages context and user preferences to make decisions about the intentions of users.
- Reference resolution. A reference resolution mechanism allows external components to help validate the semantic ratings generated by nodes of the semantic network.
- Flexibility. Changes to language domain definition and processing are easily and graphically done through the Active Editor. There is no code to edit, system to shut down, nor program to recompile. Existing node of the semantic network can be updated, new ones created using Wizards. The topology of the network can also be updated, existing nodes re-wired, new ones connected, the Active Editor will automatically generate the underlying code.
- Multi-lingual support. One the domain model is constructed for an application, it is an easy matter to add additional vocabulary to support multiple human languages; the domain model is completely reused for each. Contrast this with grammar-based approaches, where for each human language accepted, the grammar must be re-written from scratch, with little reuse.

Discussion

Now that it has been more formally described, it is interesting to compare our Active-based language processing technique with similar research efforts.

In some aspects, our approach is similar to *partial* or *shallow* parsing techniques. The idea consists of parsing small chunks of incoming utterances instead of fully matching them with a grammar. This technique is useful for human driven applications and information retrieval from heterogeneous data sources [1][31]. Shallow parsers lead to robust designs but have to be coupled with separate components that will assemble results into a specific application domain.

For instance, the *Smart Sight* tourist assistant[86] system couples a grammar-based shallow parser with a semantic engine to provide post-processing and validate the output of the parser with the application domain. These approaches, while supporting the robustness required for practical use, start with a general language grammar and then try to fit domain and semantic knowledge into this; our Active Semantic Network approach conversely starts with the domain model and lays a light-layer of language over this. Our approach produces a network that is easily understandable from a programmer's point of view, and can be easily ported to multiple human languages with maximal reuse.

Perhaps the closest work to our approach is described in [36], in which a system called AAOSA provides a flexible framework for building natural language interfaces based on networks of interconnected agents. Similar to our approach, a semantic network is created and various nodes produce claims about incoming information. Like Active, in addition to a domain-oriented Semantic Network approach, AAOSA can also be used to implement grammar-based parsing if desired [35]. Our approaches differ in several respects:

- At an algorithmic level, Active processes utterances by parsing, validating, and producing a semantic representation in a single pass through the semantic structure, whereas AAOSA uses multiple passes to achieve the same result, first interpreting claims and then validating and generating a semantic representation;
- AAOSA is a specialized tool suitable only for natural language interpretation (producing a structured representation of an input utterance), whereas Active is a general-purpose AI framework that provides an underlying rule-engine framework and numerous AI methodologies;
- Thanks to the previous point, during the language interpretation process, as Active interprets an utterance, it seamlessly blends and integrates context and dialog management, business logic (for error validation and partial structure completion), and service delegation into one seamless process. For example, as a city is mentioned by the user (e.g. "make reservation at Il Fornaio in san jose"), external services are delegated to lookup the zipcode and state for the city, ambiguities are resolved (e.g. San Jose, CA or San Jose, NM) by leveraging context and dialog, and numerous validations (e.g. missing time, state changed so city context becomes invalidated) are processed through business flow logic.

It is Active's deep integration of data modeling, language interpretation, context and dialog management, service delegation, and anticipation and execution of logic flows that makes Active truly unique. Not only are all these capabilities important for constructing assistant applications, they bleed into each other, such that individual capabilities such as language interpretation rely on other methodologies as part of what they do.

5.2.4 Conclusion

This section has presented two language processing techniques implemented in the Active framework. For both techniques, we presented the underlying design and the actual implementation using Active rules and concepts. This

important step validated our approach and helped us design and implement the Active framework software suite. Language processing is an important element of intelligent assistant applications. Now that we have implemented language processing in Active, the next step is to connect a user interface to allow users interaction with the system. As explained in our introduction, an Active powered system consists of a community of loosely couple services, including user interfaces. The next section introduces how Active combines a group of services to form a user-centric application.

5.3 Service Management

5.3.1 Introduction

As presented in our introduction (see section 1.3.4 on page 11), an Active-based system consists of a set of services working with one or more Active Ontologies in charge of core reasoning tasks. So far, we have presented how to use Active Ontologies to create a language understanding component. In this section, we will explain how Active Ontologies are used to manage a community of services.

First, we introduce the notion of *service oriented architectures* (SOA), then we extend it with the concept of *delegated computing*. We continue by showing how these notions have been designed and implemented within the Active framework. To illustrate these concepts, various user interfaces will be integrated as services that collaborate with the language understanding module built in section 5.2.3 on page 71. The section ends with a presentation of results, a discussion and a conclusion.

Service oriented architectures

Software applications designed around an SOA architecture consist of a community of *loosely coupled services,* whose collective actions implement the overall behavior of the system.

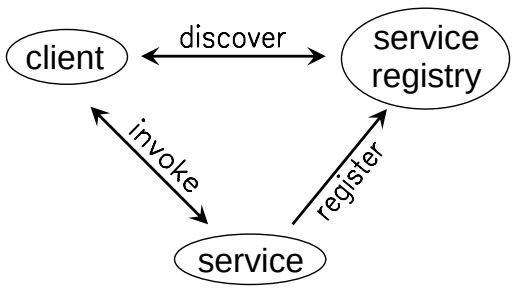


Figure 5.22: Three steps for an SOA invocation

Loosely coupled services are components that do not need to know about each other at design time. They nevertheless comply with some standard that allows for their discovery and remote invocation. Therefore, as long as they comply with invocation standards, services can be written in any programming

language, run on various hardware platforms and even be distributed over multiple hosts. For discovery, services can advertise their capabilities and specific communication details into public registries. Clients can query registries at runtime to discover and invoke available services that suit their needs.

Figure 5.22 summarizes the three steps involved in service interactions within an SOA system. Step one, a service provider registers with a service registry to advertise its capabilities. Step two, a client contacts the service registry to find a service provider that suits its needs. Finally, step three, the client invokes a method on the service provider to perform the required operation.

As with any software architecture, this approach has certain advantages and drawbacks. The main benefits of such approach are:

- Reusability. Services can easily be shared and reused across multiple applications.
- Integration. Modern software tends to be composite and therefore requires large *integration* efforts at both design and implementation phases. Integration is the process of linking heterogeneous, sometimes distributed applications in order to implement a global behavior. Leveraging SOA standards and best practices facilitates integration at all levels, ranging from communications protocol to higher level concerns such as security and reliability.
- Design flexibility. Components of a complex application can be shared, replaced, upgraded or even removed with a limited impact on other modules of the system.
- Widely available technology. For twenty years, the CORBA[78] platform has been used to design and implement service-based applications. During the last decade, the explosion of the Internet and XML technologies has been driving the growth of *web services*[13]. This technology uses SOAP for inter-service communication, WSDL to describe services capabilities, UDDI for service registry and HTTP as the transport mechanism. More recently, web 2.0 technologies called for lighter HTTP-based protocols such as REST or XML-RPC to expose backend systems and services over the Internet. This wealth of technologies is available and mature enough to be used as the foundation of service-based architectures.
- Robustness. SOA architectures facilitate the design of backup policies where a main application is in production while a backup system serves as an on-line mirror, ready to take over should the primary service fail.

There are nevertheless tradeoffs to be taken into consideration when building SOA systems:

- Performance. The process of data marshaling (converting all data flowing across and application into a shared standard) has an impact on the overall performance of the system. Additionally, SOA applications are often distributed over a network, adding transmission times to the overall response time.

- Interoperability. Since all components must comply to a common standard for data exchange, communication protocols, and service definitions, some specific implementations may not be fully compatible with others. Therefore, SOA applications always require thorough interoperability tests before deployment.

- Security. Connecting multiple heterogeneous systems over a network, sometimes the open Internet, calls for carefully designed security policies.

SOA for user-centric applications

As explained in our introduction, service oriented architectures play a key role in our approach to design user-centric intelligent assistant applications. In this context, we categorize relevant services into fours classes: *sensors*, *effectors*, *information sources* and *processing agents*.

- Sensors: Any service in charge of observing the environment and report information to the Active Server for processing. This category encompasses a wide range of service types: user interfaces, where users type, click and manipulate graphical objects; speech, gestures or handwriting recognizers fall into this category. In also includes any type of environment sensor able to measure values such as temperature, humidity, speed or acceleration. Finally, services able to read emails, receive instant messages or text messages also belong to the sensor category.

- Effectors: Services that allow Active to send a signal back to the environment, or to otherwise act on the environment. For user communication services, there is a symmetry between sensors and effectors. A user interface that prints information, a service able to send emails, instant or text messages are all effectors. Additionally, powered systems or robots are effectors as well.

- Information sources. Many instances of intelligent assistants are in charge of retrieving data from information sources. These applications are typically designed as wrappers able to extract content from databases, the Internet (through *web scrapers*), or any custom data source.

- Processing agents. Complex applications may need to delegate processing-intensive tasks. For instance, a message may have to be encrypted before being sent out by an email effector service. Symmetrically, incoming messages may need to be decrypted. Another example would be a translating service, transforming messages from one language to another.

In the context of user-applications, an SOA approach brings specific advantages:

- Multi-modality. Users express commands and intentions through multiple modalities. For instance, they can type, talk, gesture or write to communicate goals to an assistant. In an SOA approach, each modality is managed by a specific service, all contributing their information to the Active Server for processing.

- Ubiquitous. Intelligent assistants need to be ubiquitous. For instance, in the scenario presented in the introduction of this document, the user communicates with his assistant from various locations: his home, his office and his car. The set of services used by the assistant to interact with the user changes over time and location. Dynamic services detection and selection is therefore required, and greatly facilitated by an SOA architecture.

Delegated computing

At this point, we have presented SOA and how we intend to use this design approach in the context of Active-based applications. The present section introduces *delegated computing*, a technique designed to fully leverage the power of service-based architectures.

The flexibility of SOA designs comes with many challenges, one being the dynamic management of services. Since services can be dynamically registered and discovered, multiple providers of the same service function may be available at the same time. What should happen then? When a client has multiple candidates for invocation, which service should be invoked? Our design tackles this problem by using *delegated computing*, a technique where clients *delegate* the selection and invocation of services to a *broker*.

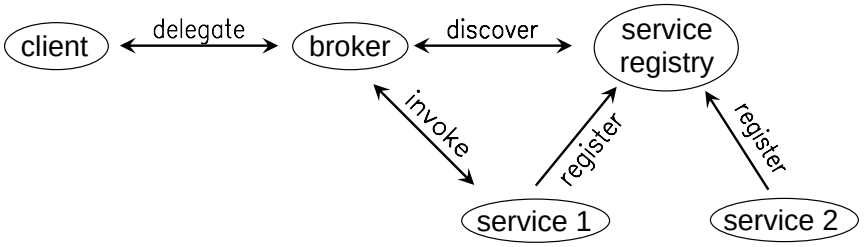


Figure 5.23: Delegated Computing

Figure 5.23 shows a design where SOA is enhanced with the concept of delegation. First, service providers register with the service registry to advertise their capabilities. When a client needs a service, it would not directly contact the service registry to get a provider and perform an invocation. Instead, it *delegates* the invocation to a *broker*. The broker implements all the logic required to get the list of suitable service providers, select the most appropriate candidate, perform the invocation, sort responses and report the results to the client. When invoking a service through delegation the client describe *what* is needed, instead of *who* to call.

Delegation cannot be implemented without *normalization*. When using a broker, clients make requests for a specific type (category) of service to call, not for a specific provider. This notion implies that service categories have to be defined and *normalized*. The definition of a service category consists of four elements. The name of the category, the input parameters to provide when requesting an invocation, the output parameters returned after processing and a list of attributes used to qualify service providers. Therefore, the service

registry does not provide a flat list of service providers, but organizes them into well defined categories to which providers need to comply.

In the context of delegation, service registration requires more information than simply a name and an address. Upon registration service providers provide the following information:

- *Category and qualifiers.* First, service providers specify the category of service where they belong with its own unique identifier. Optionally, a list of qualifying attributes may be added to further define the service. These would for instance be the response time, the cost, and the reliability, and could be used by the broker to make decisions about the best provider to pick.

- *Data transformation* In the context of delegated computing, third party services are registered under a category that defines a normalized set of input and output parameters. These services were not designed to comply to any specific API, therefore their proprietary API is not likely to match the one imposed by the category under which they register. Therefore, service providers need to provide translation information used to convert data, back and forth, from category normalized data into the specific service provider API.

- *Data transmission* protocol. To perform an invocation with a service provider, the broker needs to have information about the communication protocol to use. For instance, if a service is exposed using SOAP, a WSDL file needs to be provided. Similarly, CORBA, RMI or REST technologies require information about the protocol and address where the service can be contacted.

The broker performs delegated invocation as a three-step process. First, it receives a requests from the client about the invocation to perform. The caller provides invocation details expressed in the normalized category definition, including the type of service to invoke, input/outputs parameters and timeout. Optionally, callers can provide additional information to help the broker decide on the specific provider to call. Such details would include expected reliability, response time, cost or any relevant attribute that describes a provider.

At the second processing step, the broker contacts the service registry to get a list of all service providers able to provide the functionality expressed by the caller. Using additional information, the list is trimmed and sorted so that best candidates appear on top. Meta-agents can additionally provide third party recommendations about the value of specific service providers.

Next, the broker actually invokes service providers. Using the list of best candidates, the broker invokes service providers using one of two techniques:

- *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 are able to 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 data sources.

Finally, the last step consists of reporting results to the caller. If providers cannot report results within a time limit provided by the caller, the broker reports a timeout. If providers manage to provide answers and results within the timeout period, the broker reports a single result for sequential invocations, an aggregation of results for parallel invocations.

Using the delegation technique and a broker, clients are shielded from the complexity of heterogeneous APIs and various communication protocols. It allows for service providers to be dynamically removed, added, upgraded and selected without having to change anything on the client side. In the context of our work, delegated computing is important feature. For instance, it is used for modality management. As an Active Ontology in charge of language processing reaches a conclusion, it needs to communicate some information to a user. The language processing component does not have to know about specific modalities, therefore it delegates the task of notifying a user to another Active Ontology that uses delegation to pick the best modality to deliver the message.

5.3.2 Active implementation

This section presents how service management through delegation is implemented in the Active framework. First we show how Active Ontologies are used to create a service registry and a broker. We then briefly describe Active Editor plugins and Active Server extensions created for that purpose, before discussing our results. To illustrate the content of this section, we will connect various user interfaces to the language processing module presented in section 5.2 on page 64.

Service categories

To implement delegation of services in Active, the first element to design is a service registry. The registry is more than a flat list of services, it defines a set of *service categories* where actual service providers can register. A service category consists of a name, *input parameters* and *output parameters*. Input parameters define the data structure a service registered under the category will need when invoked, output parameters define the data returned by the service after processing.

In the context of Active, service categories are modeled with concepts and relationships. For instance, figure 5.24 shows how a user notification service category can be represented. The top root concept of the structure represents the category and bears its name, **notify user** in our example. Relationships of type *is category member* define invocation parameters of the category. Inward relationships represent input attributes (a **request** structure in our example), outward relationships point to return parameters of services that belong to the category (**response** structure in the example). Input and output parameters are represented with complex structures of concepts connected with *is member of* relationships, already used in the context of language processing applications.

The unified approach of the Active platform allows programmers to use the same technique to model both service categories and the spoken domain of an application. This unique innovative design brings multiple advantages. First, the comprehensive tools used to create language processing semantic networks

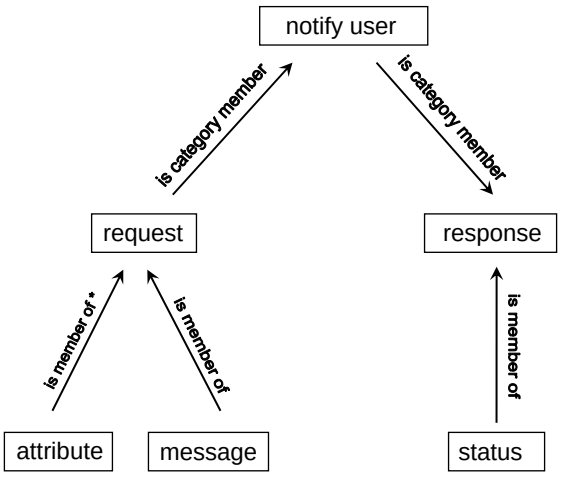


Figure 5.24: User Notification Service Category

can be leveraged to model input/output parameters of service categories. Developers can create tree-like hierarchical structures made out of nodes and relationships to model the normalized API of a service category. Secondly, the same structure definitions can be referenced and shared across all Active Ontologies without redefinition. For instance, if a service category requires an address as part of its API, one can directly make a reference to the **address** node defined in the semantic network.

Service providers

The previous section introduced the notion of service categories modeled with concepts and relationships. This section explains how Active models actual service providers registered as service category providers.

Invocation of service providers Service providers are called using the Active invocation mechanism exposed in section 5.1.2 on page 59. Therefore, a registered service provider exposes a rule whose condition is compliant with the definition 2 on page 59.

Definition 8 Service provider invocation fact

```
invoke($operation_name,$input_params,
  [provider_name($provider_name), category($category),
   tx_id($tx_id), timeout($timeout)]))
```

In addition to the basic set of invocation attributes, service providers expose two additional attributes (see Definition 8):

- provider_name : The unique name of the service provider
- category : The category in which the service is registered

Note that exposing these two additional attributes does not prevent service providers from being called directly, without using the broker.

API normalization and communication protocol In addition to a rule whose condition complies with the Active invocation technique, each service provider provides two sets of information: data about API transformation, the second about its communication protocol.

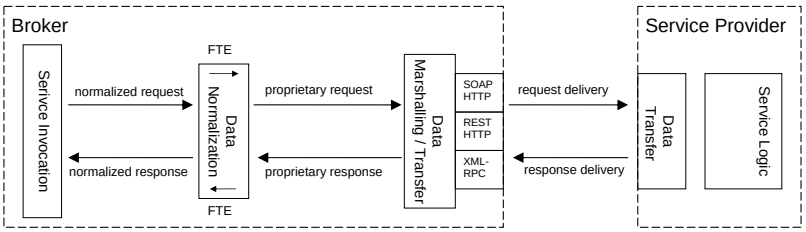


Figure 5.25: Data Normalization and Transfer

As explained earlier, service providers have their own proprietary API that is unlikely to match the normalized API imposed by the category under which they are registered. Therefore, service providers need to provide translation information used to transform data flowing in and out the normalized API used by clients. To implement this mechanism, an Active Server extension provides a transformation engine designed to convert a source fact into a destination fact using a conversion chart. Similarly to XSLT, the new destination fact is produced based on the source fact which is not affected by the operation. A simple *fact transformation language* (FTL) has been designed and is used to instruct the *fact transformation engine* (FTE) on how to create a destination fact based on the content and structure of a source fact. Using the FTE, service providers register two snippets of FTL code, one to convert input parameters from normalized requests into proprietary service requests, another one to convert proprietary service responses into normalized output parameters.

Once a request has been converted into the right format using the FTE, it has to be delivered to the service provider. Symmetrically, responses need to be communicated back from the service to the Active Server. To perform this operation the Active Server needs to support the communication protocol used by services providers to expose their functionalities (SOAP, REST, XML-RPC, etc). The SOAP protocol is becoming the *de facto* standard for SOA applications, therefore an Active Server extension has been developed to implement SOAP-based data exchanges over HTTP. Its core consists of two converters, a Fact-to-SOAP (*serializer*) for outgoing messages and SOAP-to-Fact (*deserializer*) for incoming responses.

Figure 5.25 shows how the combination of the FTE and the Active Server SOAP extension allow us to implement the normalization and communication aspects of delegated computing.

Service Invocation through brokering

Now that we have presented the mechanisms provided by the broker, let us describe the high level interface used by caller to use it. As shown in figure 5.23 on page 99, in the context of delegated computing, clients do not contact service providers directly, but delegate the selection and invocation of providers to a broker. The method used by clients to communicate with a broker is very similar to the invocation technique. To request the delegated invocation of a service type, a caller asserts a fact of the form show in definition 9, where:

- category_name : The name of the service category to invoke
- input_params : Standardized input parameters defined for the category
- tx_id: Unique transaction id for the delegation
- timeout: How long should the caller wait until the delegated invocation is considered to have failed
- policy: Informs the broker on how to invoke selected service providers. (sequential or parallel)
- broker_information_list : An attribute list providing any information the broker can use to select the most suitable service providers

Definition 9 Delegation fact

delegate($category_name, $input_params, [tx_id($tx_id), timeout($timeout), policy($policy), $broker_information_list]))

Once the delegation fact is asserted by the caller, a process similar to the invocation technique unfolds. Figure 5.26 shows the main steps of the delegation process:

1. The caller creates and asserts a delegation fact that specifies the type of service to invoke, the input parameters and delegation attributes such as a unique transaction identifier, a timeout. Additionally, a list of attributes can be provided to help the broker select the best provider.
2. The broker exposes an Active rule whose condition fires upon assertion of delegation facts compatible with definition 9. The rule action starts the brokering logic by getting the list of all service providers registered with the category to invoke.
3. Based on the policy specified by the caller, the broker invoke selected service providers.
4. Service providers are invoked and provide requested results.
5. The broker gathers results and notifies the caller.

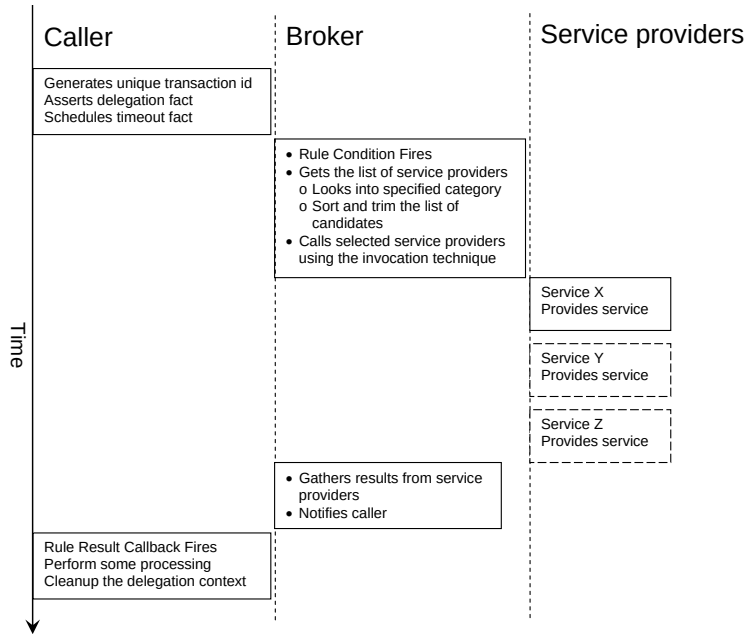


Figure 5.26: Delegated Invocation Timeline

Wizards

To facilitate the use of the technique presented in this section, two Active Editor wizards have been implemented.

First, a wizard has been created to create a service category. When activated the wizard asks the user about the name of the category to automatically create the top concept of the category model.

The second wizard allows programmers to register SOAP service providers. Through a sequence of interactive screens, the wizard invites Active users to describe each step of the service registration. First, the wizard asks for the name of the service to register and the category where it belongs. Since the wizard deals with SOAP services, a pointer to a WSDL file (local file or URL) is also required. The Wizard automatically retrieves and introspects the WSDL file to gather relevant information about the service to register. The user is then invited to pick an operation among the list of methods described in the WSDL file. Next, the wizard reads the input parameters of the category where the service is to be registered and the input parameters of the selected operation as defined in the WSDL file. Through an interactive mapping tool (see figure 5.27), the Wizard invites the user to map the normalized input parameters defined by the service category into the proprietary definition retrieved from the WSDL file. The same mapping operation is then performed for output parameters. The last step is a test screen, where Active programmers can test the service registration by actually invoking with sample requests. Finally, the wizard uses the collected information to automatically create an Active concept, instrumented with all rules necessary for the service to be invoked through the delegation technique described in this section. The wizard can be re-activated any time to update

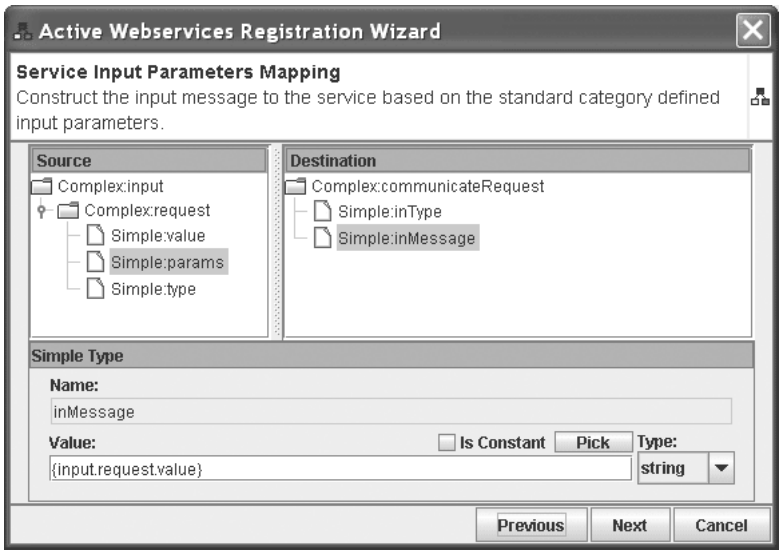


Figure 5.27: SOAP Service Registration Wizard (Mapping step)

the service provider definition.

5.3.3 Practical example

Prototype

The method and implementation defined so far in this chapter have been used to connect user interfaces, as services, with the language processing Active Ontology shown in section 5.2.3 on page 71. The system allows users to submit utterances for parsing, and get a response as a parsed view of the utterance. The application, which consists of two Active Ontologies and a set of services, is our first Active-based system as initially envisioned in section 1.3.4 on page 11. This system has been designed and implemented as a test tool for both language processing and delegation of services.

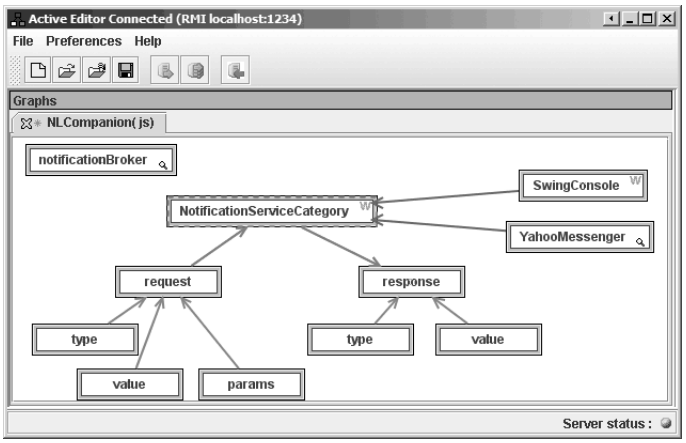


Figure 5.28: Service Management Active Ontology

To realize this prototype, an Active Ontology in charge of service management has been created to implement service delegation. The Ontology models a *notification* service category, that takes a request structure a input and returns an acknowledge structure. Figure 5.28 shows the service management Active Ontology loaded in the Active Editor.

For our example, two service providers have been registered under the notification service category:

First, the language processing test console (see section 5.2.3 on page 92) interacts with the system through SOAP messages. The console is used in two modes. First, an interactive mode allows Active programmers to quickly type and send utterances for processing. Parsing results are sent back to the console through the delegation mechanism. The console can also be used to run regressions tests, consisting of sequences of utterances and expected parsing results. Regression tests are an important component of the design and development of Active-based systems (see section 6.1 on page 122).

The second service provider connected to the system is an Active Server extension able to connect to the Yahoo Instant Messenger (YIM) network. The extension provides an Active impersonation on YIM with whom anyone with an account can start a dialog. The extension reports incoming utterances to the Active Server fact store for processing, and reports results as a service registered under the *notification* service category.

In addition to modeling a service category, the delegation Active Ontology implements a broker, in charge of processing user notification through delegation. Figure 5.29 describes the sequence of events unfolding for each user request:

- An utterance is submitted from the user interface *(step 1)*. In the case of any interface able to communicate with SOAP, the utterance is reported as a fact asserted in a fact store *(step 2)* through the built-in SOAP extension exposed by the Active Server. For more proprietary interfaces, such as the YIM, a specific Active Server extension has been implemented to connect to the YIM and create facts out of incoming events. Facts asserted to

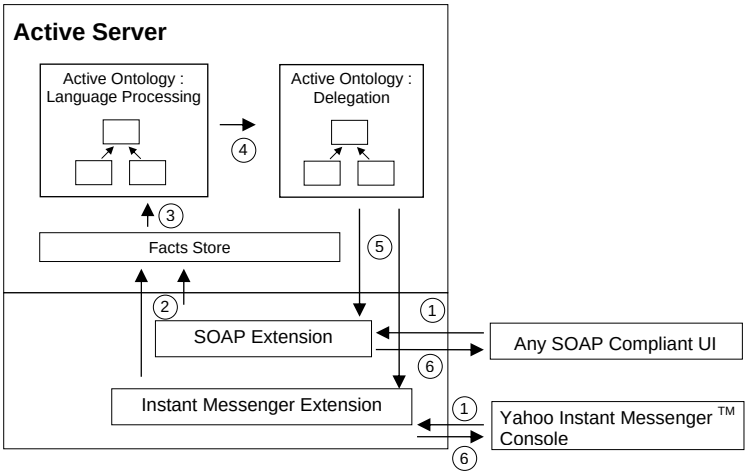


Figure 5.29: Prototype Architecture

report an utterance are standardized in the definition 7 on page 88, item *d*. Therefore, the Active Ontology in charge of language processing will process all utterances similarly, regardless of their origin.

- The Active Ontology in charge of language processing reacts *(step 3)* and produces a parsed command that takes into account the contribution of the new utterance. The results are asserted as facts to the Active Server store intended to trigger the Active Ontology in charge of delegation *(step 4)*.

- The assertion of the language processing Active Ontology triggers the notification broker of the delegation Active Ontology to decide which service to used to deliver the parsing results to the user *(step 5)*. In this example, the broker uses the simplest technique, which consists of using the same modality as the one used to deliver the utterance. The actual delivery of the response is performed by specialized extensions that know how to communicate with specific interfaces registered as service providers *(step 6)*.

5.3.4 Conclusion

After introducing the concepts of service oriented architectures and delegated computing, this section describes how these techniques are implemented using the Active platform. It defines the messages exchanged among components and the extensions, plugins, and wizards developed to ease the registration and invocation of services. The section also describes a practical example, where an Active-based system uses delegation to dynamically select the means to deliver information to users.

5.4 Process Management

5.4.1 Introduction

We have so far presented how Active is used to perform natural language processing and manage a community of services. In order to build full applications, the next step is to provide a component that encapsulates the core logic of the system. In user-centric applications such as intelligent assistants, process management is used to model the dialogs, or sequence of actions, to undertake when accomplishing complex tasks on behalf of the user.

This section starts by introducing workflow programming, tools and implementations of the technique. Next it presents how workflows are implemented in the context of the Active framework. Finally, a conclusion discusses the results and applications of Active-based workflows.

Workflow programming

The most natural way to implement process modelling with Active is through *workflow-based programming*. This technique can be easily and elegantly implemented using the event-rule paradigm at the core of the Active system.

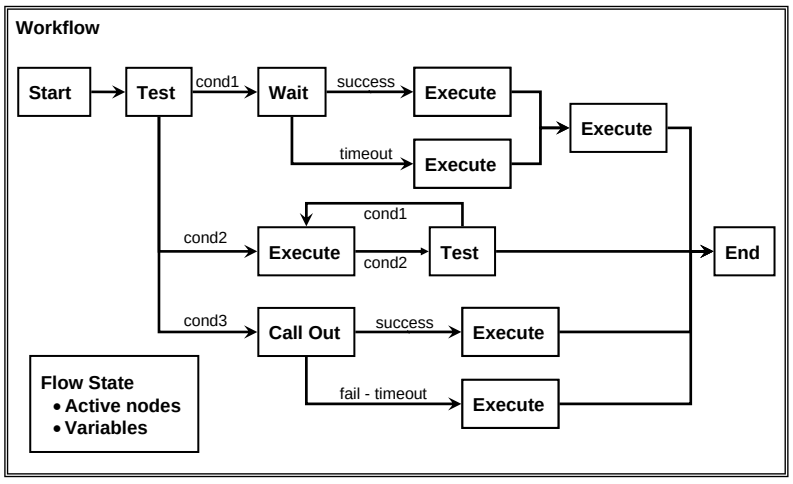


Figure 5.30: Workflow modeling

A *workflow* models a process as a collection of basic *work units*, whose execution sequence are represented with *links* (see figure 5.30). Work units can be considered as basic building blocks that can be combined to model complex activities, activated along directed links. Work units are connected to the network with incoming and outgoing directed arcs called *links*. Incoming links are use to convey an execution signal that trigger the activation of a work unit. When a unit has performed its task, it notifies all units connected through outgoing links.

Workflows are executed by a *workflow engine*, in charge of creating, running and hosting multiple *instances* of a given workflow model. Each instance of the

flow holds its own *workflow state* that consists of the list of active work units and *instance variables*. Instance variables are key-value pairs used to share information among work units.

Typically, workflows are made of the following work unit types:

Start. Wait for an event to trigger the creation and execution of a new instance of a workflow. The trigger condition can come from the outside world, or be internally sent by another workflow. Start units can be connected to one or more units to be activated.

Execution. Once activated, some processing task is performed. Any type of processing can be performed by execution activities, for example processing the flow state variables, connecting to a database, reading a file or sending an email.

Wait. On activation, *wait* units suspend their activity until a specified event occurs. There are two classes of outgoing links from a *wait* unit. If the expected event occurs before a given timeout value, a *success* link points to the next work unit to activate. On the other hand, if the event does not occur within the specified timeout, a *timeout* link directs execution to a dedicated work unit.

Callout. Workflows are designed to be integrated with external components through communication standards such as SOAP, RMI or CORBA. Call outs are mechanisms where an external component is invoked through an RPC technique. There are multiple ways out of a callout work unit depending of the status of the transaction (success, failed, timeout). Each situation triggers the activation of specific outgoing links to activate the appropriate work unit in charge of ongoing operations.

Test. Test work units provide conditional branching by holding a condition to each one of their outgoing links. When activated, conditions are evaluated and only the links whose conditions are valid will be activated.

End. Work units designed to terminate the flow execution, cleaning all flow state variables.

Workflow-based programming is a mature technique. On the theory side, workflows have been well studied and can be modeled as petrinets[59, 76] for simulation, validation and performance predictions. On the application side, many popular tools, products and standards are used in various industries where workflows are required. Many products offer visual programming graphical interfaces, allowing developers and non-developers to model business processes as workflows by dragging, dropping and connecting work units. In the industrial automation space, the Labview[40] system is widely used to model processes interacting with equipments to control. In the software industry, leveraging the rapid growth of service oriented architectures, multiple workflow techniques are used to orchestrate the actions of loosely coupled components. The BPEL[66] specification has emerged as the de facto standard and is the basis of several workflow system implementations[54].

Workflow-based process modeling with Active

Workflows are well suited to be the programming paradigm used to model processes within the Active framework.

First, the structure of a workflow consists of processing elements connected by oriented links. Since an Active Ontology is made out of concepts and relationships, workflow modeling naturally fits with the philosophy and development tools of the Active framework. As described previously, existing workflow tools offer a graphical programming IDE allowing users to model applications by dragging, dropping and connecting processing elements. This approach is aligned with what Active is trying to achieve, by easing the development of complex systems though high-level graphical tools.

Secondly, the underlying processing core of workflows is similar to the core of the Active rule-base engine. Workflow work units have conditions that can become valid anytime to trigger their execution. Such event-based paradigm can elegantly be modeled and implemented as a set of Active rules. Some work units (wait, callout) use timeouts to control subsequent actions. The Active fact store natively supports a life span of facts, thus providing a way of implementing timeouts. Finally, workflow instances have their own data space, where their state and local variables are kept. Another feature naturally fitting the Active framework, where a series of facts can be used to persist the state of an on going workflow instance.

Third, call out work units of flows can leverage the invocation and delegation mechanisms of Active. In addition of calling specific services using the Active invocations techniques, workflows callouts use the Active-based delegation technique to provide on the fly selection of service providers.

Finally, in the context of user-centric applications, workflows are excellent means of modeling user activities and interaction dialogs. Since multiple workflow instances can run at the same time, each having its own data space, multiple users can use the system simultaneously, each having their own private sessions.

5.4.2 Active implementation

This section presents how an Active technique to model processes as workflows has been designed and implemented. First, the data structures to hold the state of a flow instance is introduced. Then, we show how workflow work units have been designed and implemented with Active concepts and rules. Finally, we present the set of Active Editor wizards designed to model and run complex workflows with Active through wizard activation and graphical editing.

Work units

Work units are implemented as concepts, instrumented with specialized rules.

Figure 5.31 show the structure of the rule controlling the execution of a work unit. The condition waits for a flow_pipe facts whose destination matched the name of the work unit to trigger its execution. Once its task performed, the work unit passes control the next element of the flow by asserting a flow_pipe with its name as the source and its successor as the destination.

The activation signal sent between work units is implemented as flow_pipe facts (see definition 10, item a) asserted as events. When a work unit has finished

its execution, it passes the control to its successor by asserting a flow_pipe fact, using its name as the source and the name of the work unit to activate as the destination. Since flow_pipe facts are asserted as events, they do not remain in the fact store for more than one evaluation cycle.

Definition 10 Standardized facts for workflow modeling

(a) flow_pipe($source, $destination, [flowid($fid)])
(b) flow_var($var_name, $value, [flowid($fid)])

Instance variables of a workflow are also implemented as Active facts (see definition 10, item b). Each value is a triplet, with the name of the variable, its value and the unique instance of the flow where they are stored.

Note that all flow_pipe and flow_var facts are tagged with a unique flow instance identifier, allowing multiple instances of a workflow to simultaneously be executed and hosted on an Active server, each instance having its own data and state space.

Rule name	FlowWorkUnit
Condition	Store.checkEvent(flow_pipe($source, workunit_name, [flowid($flowid)]))
Action	.... perform the actions of the work unit .... Store.writeEvent(pipe(workunit_name, destination_name, [flowid($flowid)]))

Figure 5.31: Work unit structure

Using this technique, different types of work units have been implemented with Active rules and concepts:

Start Start elements define entry points that will trigger the execution of a process. Unlike other work units, start elements do not wait on work units to be activated, but on any event happening on the fact store. Whenever an event validates the condition, the start element creates a unique process instance identification and passes the control to all connected work units. Note that a workflow can have multiple entry points.

Execution Execution elements provide general purpose processing at any point of the workflow. Execution nodes contain Javascript code to be executed, before passing the control to the next element of the flow.

Wait Wait nodes suspend the execution of the workflow until a specific event occurs. The even to wait for is represented as a unification fact pattern. Therefore, any fact asserted to the store that matches the pattern, triggers the wait node which then resumes the workflow execution by passing control to all connected work units. A timeout can be specified to undertake action when an awaited event does not occur.
This node has two types of outgoing links, represented as two types of

Active relationships. Relationships of type *flow event ok* are used to activate work units when the awaited event actually occurred. Relationships of type *flow event timeout* point to work units to activate when the event does not occur within the specified timeout period.

Invocation Invocation nodes provide asynchronous calls based on the Active invocation (see section 5.1.2 on page 59) or delegation (see section 5.3.2 on page 104) mechanisms. Once activated, the work unit gathers parameters from the local workflow instance variables and uses them to invoke (or delegate) external processing. Once results are received, they are asserted as new workflow instance variables.
Alike the Wait work units, different types of links control the execution flow out of invocation units. Relationships of type *flow event ok* are used to activate work units after successful invocation. Relationships of type *flow event timeout* point to work units to activate when the invocation did not provide any results within a specified timeout period. Finally, *flow event failed* point to work units to activate when the invocation fails due to an unexpected problem.

Test Test work units provide conditional branching. Test nodes hold a condition, based on any processing of instance flow variables, for each outgoing link. Upon activation, it evaluates the link conditions to route the execution flow on specific branches.
Two behaviors have been implemented: *unique dispatch* and *multiple dispatch*. For the *unique dispatch* technique, branches are evaluated until one of them succeeds. In the multiple dispatch case, once a branch succeeds, the evaluation process continues to potentially active more than one parallel execution path. For both techniques, if no condition is met, a default link is activated. Similarly to Invocation and Wait units, multiple types of outgoing links are represented with classes of Active relationships.

End End elements define end of a process execution. Once activated, an End unit cleans up all the workflow instance related information.

Note that our implementation does not have explicit *fork* and *merge* elements. Work units can be the source of multiple links, thus activating multiple parallel execution paths. Symmetrically, any work unit can be the destination of multiple links, therefore working as an execution control merge. To activate work units with multiple incoming links, the workflow Active-based implementation supports two policies: *conjunctive activation* or *disjunctive activation*. Conjunctive, or synchronized, activation requires all incoming links to be activated for the activation of a work unit. Disjunctive, or loose, activation only need one of the incoming links to be activated to trigger a work unit execution.

Active Editor wizards and Active Server extension

As with other Active methodologies, this technique has been encapsulated into a set of interactive Active Editor wizards and Active Server extensions allowing programmers to graphically model complex processes without writing any Active code.

First, a collection of workflow-related Active Editor wizards have been created, allowing for the creation of each work unit type. Each wizard contains

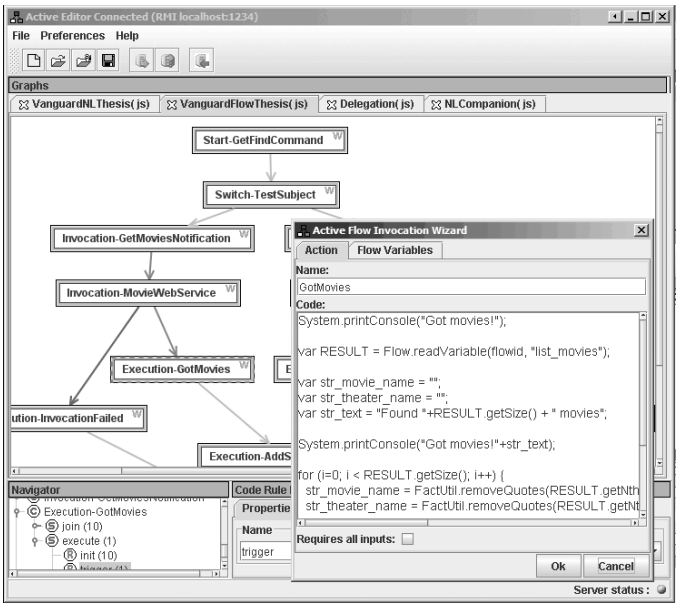


Figure 5.32: Workflow Active Editor Wizard

a sequence of interactive screens, designed to for Active users to configure the workflow steps to create. As the wizard is closed, an Active concepts and all required rules and rulesets is automatically created and added to the current Active Ontology. Similarly, a set of specialized Active relationships can be used to connect concepts and model the execution path of the workflow. Figure 5.32 show how the Active Editor is used to interactively model an invocation work unit.

Additionally, an Active Sever extension extends the target scripting language primitives so that Active programmers can read and write workflow instance variables from their code snippets.

5.4.3 Evaluation

We propose to evaluate the Active-based workflow along three axes. The goal of the evaluation is to see how the Active system, at its current stage of development and feature set, compares with existing workflow systems and their functional definitions. First we assess how flexible our approach is in terms of modeling workflows. Then, we compare the technical and user experience aspects of our system, to finally provide some information about the pure performances of our system.

Modeling

In his reference paper, Aalst [77] describes workflow modeling techniques as a list of workflow design patterns. We propose to go over each pattern and analyze if and how it can be implemented with the Active-based workflow technique.

Pattern 1 (basic) : Sequence - supported.
This is the simplest pattern, which consists of sequentially passing control among work units.

Pattern 2 (basic): Parallel split - supported.
A single thread of control splits into multiple independent threads of control. As described previously, any work unit of an Active-based workflow can be connected to multiple work units and thus create multiple parallel execution flows.

Pattern 3 (basic): Synchronization - supported.
A point in the workflow where multiple parallel branches converge into on single thread of control. This pattern corresponds to the *conjunctive activation* technique described in the previous section.

Pattern 4 (basic): Exclusive choice - supported.
A point of the workflow where control is conditionally passed to one single branch. This is the *unique dispatch* behavior implemented by the Test work unit of the Active-based workflow technique.

Pattern 5 (basic): Simple merge - supported.
A point where multiple alternative branches merge without synchronization. This situation assumes the alternative branches never execute in parallel. This pattern corresponds to the *disjunctive activation* technique described in the previous section.

Pattern 6 (advanced): Multi-choice - supported.
A point where, based on conditional processing, a number of branches are chosen. This is the *multiple dispatch* behavior implemented by the Test work unit of the Active-based workflow technique.

Pattern 7 (advanced): Synchronizing merge - not supported.
During a merge, the system should make sure that candidate branches are not executed twice. Each branch that leads to a merge should be considered as a critical section whose execution needs to be forbidden until the merge has taken place.

Pattern 8 (advanced): Multi merge - supported.
A point in a workflow process where two or more branches reconverge without synchronization. If more than one branch becomes activated, possibly concurrently, the activity following the merge is started for every activation of every incoming branch. This is a simple, naive default behavior of the Active-based merge technique.

Pattern 9 (advanced): Discriminator - not supported.
A point that waits for one of the incoming branches to complete before activating the subsequent activity. From that moment on, it waits for all remaining branches to complete and ignores them. The default merging behavior of our approach implements pattern #8.

Pattern 10 (basic) : Arbitrary cycles- supported.
A point in a workflow process where one or more activities can be done repeatedly. In our model, relationships can point back to work units, thus

creating loops. Note that the number of iterations can be stored as flow instance variables and the exit condition of the loop modeled with a Test work unit.

Pattern 11 : Implicit termination - not supported.
A given subprocess should be terminated when there is nothing else to be done. In other words, there are no active activities in the workflow and no other activity can be made active. Our model only supports explicit termination using the End work unit. However, adding implicit termination of a flow execution could be done by analyzing the state of the workflow through a set of specialized Active rules.

Pattern 12 : Multiple instances without synchronization - not supported.
Within the context workflow instance, multiple instances of an activity can be created, new threads of control are spawned. Our implementation does not support this feature natively, but a solution could be found by representing the work unit to replicate in a separate workflow and trigger multiple instances of it.

Pattern 13 : Multiple instances with a priori design time knowledge - supported.
If the number of instances is known at design time, the work unit can simply be replicated at design time in the Active Editor.

Pattern 14 : Multiple instances with a priori runtime knowledge - not supported.
Event if this feature could be supported by triggering a subflow, there would not be any ways of synchronizing subflow instances with our Active-based technique.

Pattern 15 : Multiple instances without a priori runtime knowledge - not supported.
This pattern is not supported for the same reasons as pattern #14.

Pattern 16 : Deferred choice - not supported.
A point of the workflow where one execution branch is chosen over many others. The execution of unselected branches is stopped.

Pattern 17 : Interleaved parallel routing - not supported.
A set of work units can be run in any order but cannot be executed at the same time. The current implementation of the Active-based workflow technique does not support synchronization of work units that belong to parallel execution branches.

Pattern 18 : Milestone - supported.
The activation of a work unit depends on the state of a workflow instance. The work unit is only executed if a certain milestone has been reached which did not expire yet. In our implementation, a Wait work unit enables the modeling of this pattern.

Pattern 19 : Cancel activity - not - supported.
An enabled work unit is disabled. The is currently no way of canceling an activity that triggered its execution.

Pattern 20 : Cancel case - supported.

A running instance of a workflow is canceled and completely removed. In our Active-based implementation, the complete state of a flow instance is stored as a series of facts, all tagged with the unique instance identifier. Therefore, canceling the execution of a workflow consists of removing all facts tagged with its identifier.

Pattern	Staffware	COSA	InConcert	Eastman	FLOWer	Domino	Meteor	Mobile	Active
1	+	+	+	+	+	+	+	+	+
2	+	+	+	+	+	+	+	+	+
3	+	+	+	+	+	+	+	+	+
4	+	+	+/-	+	+	+	+	+	+
5	+	+	+/-	+	+	+	+	+	+
6	-	+	+/-	+/-	-	+	+	+	+
7	-		+	+	-	+	-	-	-
8	-	-	-	+	+/-	+/-	+	-	+
9	-	-	-	+	+/-	-	+/-	+	-
10	+	+	-	+	-	+	+	-	+
11	+	-	+	+	-	+	-	-	-
12	-		-	+	+	+/-	+	-	-
13	+	+	+	+	+	+	+	+	+
14	-	-	-	-	+	-	-	-	-
15	-	-	-	-	+	-	-	-	-
16	-	+	-	-	+/-	-	-	-	-
17	-	+	-	-	+/-	-	-	+	-
18	-	+	-	-	+/-	-	-	-	+
19	+	+	-	-	+/-	-	-	-	-
20	-	-	-	-	+/-	+	-	-	+

Figure 5.33: Modeling patterns comparison chart

Figure 5.33 is a comparison chart showing how different workflow engines implement the twenty patterns defined by of Aalts [77]. The table has been extended with a new column that summarizes our analysis of the Active flow system. The result shows that our implementation compares favorably with most workflow engines. All basic patterns (1 to 6) are supported. As with most workflow engines, our implementation does not support spawning of new threads to provide parallel execution the same work-unit (patterns 14 to 17). The remaining patterns are either already fully or partially supported (patterns 8, 10, 13, 18 and 20). For non supported patterns (7, 9, 11, 12 and 19), our detailed pattern analysis shows that our approach allows for three of them to be implemented.

Feature set

In addition to workflow modeling capacities, the literature provides a list of major implementation features provided by workflow systems[21, 18]. This section enumerates major characteristics of workflow management systems for comparison with the Active-based technique.

Process modeling and scripting language The process modeling language of our approach has been described and evaluated in the previous sections. The scripting language is used in execution work units, designed to provide general purpose processing as the actions of a workflow unfold. In our case, the scripting language is Javascript. The language has been selected for its popularity and easy integration with the underlying layer of the Active Server which is entirely written in Java.

Client/Server Since this workflow technique is based on the active framework, it naturally implements a client/server paradigm. The server being the Active Server, the clients being any components (including the Active Editor) connected through public APIs (SOAP or RMI).

Resource invocation If our approach allow for resource invocation through the Active invocation and delegation, we do not provide resource management. Resource management provides constructs such as resource pools, reuse and synchronizations. Examples would be database connections, EJB or large system files.

API languages As part of the Active framework, our workflow system supports three API languages.First, at the highest level, Execution work units are snippets of code to be executed to perform general processing as the flow execution unfolds. Choices of languages are Javascript and Java, enhanced with all available Active Server extensions, especially the process management extension offering methods to access flow instance variables. Secondly, at a lower level, for specific domains, new Active Server extensions may have to be developed. Such extensions are based on the Active Java SDK, API that provides all basic classes and interfaces required to write extensions to the Active platform. Finally, reporting events to a workflow is done through the public SOAP, RMI or REST APIs exposed by the Active Server.

Interoperability with external components Built-in Active Server extensions exist for SOAP, email and REST integration with external component. Any proprietary communication protocol may be added by creating a dedicated Active Server extension.

Event *s*ignaling Events can be reported through the external public SOAP, RMI or REST APIs of the Active Server, or internally by asserting events into the fact store.

Role/User administration There is no notion of *AAA* (Authentication, Authorization and Accounting) security in the current implementation of the Active Server. Users dot not require to be authenticated, anyone with an Active Editor can connect to any Active Server. In addition, there is no authorization, all functionalities are open to anyone. There is no notion of roles either (i.e. administrator, user), nor access control lists. Finally, there is no security logging gather information about who did what on the system.

Test tools There are currently no test tools dedicated to workflow management within the Active framework. Our flows have been simple enough to be tested directly as part of the overall application where they belong.

Process tracking Simple process tracking is currently done in two ways. First, the Active Console allows for querying of based flow related facts (see definition 10 on page 112) providing all necessary information about flow instances and their state. This information could be leveraged to created a graphical representation of the state of a flow. Secondly, log files are generated by the Active Server as flow processing unfolds. Analyzing the content of the log files, in real time, allows programmer to track and monitor flow activities.

Transaction coordination/recovery Our workflow approach does not support atomic transactions. The actions of work units cannot be undone (rolled back) in case of a flow failure. Since the current implementation of the Active framework does persist all of its states into databases, recovery after catastrophic failures is not guaranteed.

Performance

This section provides an high level performance evaluation of the Active-based workflow technique. A more in depth performance evaluation of the Active system and the methodologies presented here is given in section 7.3 on page 175.

Since our goal is to model processes that drive the dialog and high-level actions of user centric applications, our focus is more on flexibility than pure processing performance. Our system should nevertheless be responsive enough, exhibiting an overall response time below 15 seconds[60] to provide a tolerable experience for users.

In the case of Active-based workflow execution the performance depends on three factors. First, the frequency of evaluation cycles determines how often work units are checked for activation. By default, the Active Server evaluates each Active Ontology ten times per seconds or every 100 [ms]. Assuming that evaluating rules in charge of testing and running work units is faster than 100 [ms], our workflow execution will progress by ten steps per second. In our context, each step of the flow is an action to undertake as part of a user driven dialog. Typical actions are message formatting, data retrieval, conditional branching or user notifications. Therefore, spending 100 ms to decide which action should be executed next and notify the user does not have a significant impact on the user experience.

Secondly, the overall performance of a workflow depends on the tasks to perform, especially the code snippets specified in `execution` work units. The

current implementation allows programmer to use either Javascript or Java to define processing logic. For complex tasks that require structured code and performance, a good practice is to encapsulate it in an Active Server extension and call it from `execution` work units. It keeps the workflow simple, separates business logic of the application from its execution sequence and allows for optimization and reuse of functionalities.

Finally, intelligent assistants rely on external data to make decision or retrieve information, often in a mobile or web environment. This situation may create slower response time and need to be taken into account when designing the flow. In our case, external services are asynchronously invoked using the Active-based delegation mechanism. This technique allows us to keep the workflow running and reactive, to respond to more events and react to a timeout event should the external service fail or not respond in time.

Our prototypes presented in chapter 6 on the next page are all driven by workflows designed with the technique described in this section. Their evaluation sections show in details that workflow are designed to be fast (less then 100 [ms] per step) and do not have a significant impact on the overall response time of the system.

5.4.4 Conclusion

This section describes how to model, deploy and execute processes as workflows with the Active framework.

Following the ontology-based philosophy of Active, processes are modeled with facts and relationships. Concepts represent work units, or tasks, whose activation and execution is controlled by the relationships among them. At runtime, multiple instances of a workflow model can be run simultaneously, each within their own space. The state of each instance is represented as Active facts. This technique has been crystallized into a set of Active Editor wizards and Active Server extensions, which allow Active users to graphically model, deploy and run workflows without writing low-level code.

A comparative analysis of both the modeling and feature set aspects of workflows validates our approach. A list of reference *workflow patterns* shows that our approach supports most workflow patterns. In terms of feature set, our implementation possesses most technical elements, but does not provide industry-ready requirements such as security, high performance, scalability and tolerance to critical failures.

Finally, the performance and feature set of our process modeling tool fulfills the requirements imposed by our user-centric applications, where flows are modeled to execute user-driven actions.

Chapter 6

Applications and Prototypes

This chapter could have been titled *"putting it all together"*. It presents a basic method to build end-to-end Active-powered applications and describes how it was used to design, implement and evaluate three prototypes. Each prototype features all components and techniques described so far : *language processing*, *process modeling* and *service management*.

The three prototypes implemented and evaluated cover different application domains. First, we developed an assistant designed to help travelers gathering information using mobile devices. Historically, initial use-cases and scenarios that helped define early Active work were based on a mobile assistant designed to help users on the move. This field provided to be rich enough to encompass most of our research, implementation and practical goals. As our early implementation progressed, this choice was validated by a strong interest from commercial partners. In parallel, a second prototype implementing an assistant helping surgeons in the operating room has been created. The motivation for this second project came from the expertise and core research of the VRAI Group at EPFL, whose focus consists of providing surgeons with advanced technologies, ranging from robotics, vision systems and user interfaces. Finally, a third prototype was designed and implemented as an assistant that helps organize meetings. This project was undertaken to evaluate our ability to rapidly design and implement an intelligent assistant system that can handle long-running tasks, as opposed to the simple request-response interactions of our previous two prototypes. As meeting scheduling is a popular task within agent-based systems, with this prototype we could directly compare our system with implementations of the same problem based on other frameworks and technologies.

The chapter starts by defining design guidelines used to build our Active-based applications. It then presents three prototypes in details: a web-based assistant for mobile users, an intelligent operating room assistant, and an office assistant that helps organize meetings. For each application, a section provides an introduction, specific goals, detailed implementation and an evaluation to contrast actual results with initial requirements. A conclusion summarizes the results and reports conclusions drawn from the prototypes.

6.1 Active Application Design

6.1.1 Introduction

This section describes the technique followed to design, implement and test Active-based prototypes. First, it introduces the guidelines used to help define the requirements of user-centric applications. It then summarizes the overall software design of Active-based systems and describe their main components. Finally, guidelines about system evaluation are provided before a conclusion summarizing our findings.

6.1.2 Application requirements

Active-based applications are interactive systems that provide assistance in specific domains. When designing prototypes, we considered two sets of requirements: *functionality* and *responsiveness*. This section elaborates on both requirement sets.

Functional requirements

Functional requirements describe *what* the system provides. Since the user is at the center of the application, the starting point of the functionality definition consists of a list of requirements, converted into a set of simple scenarios describing user interaction with the assistant software to build. From the requirement list and scenarios, the core of the application is typically designed as a set of three Active Ontologies. (Figure 6.1 shows this design process)

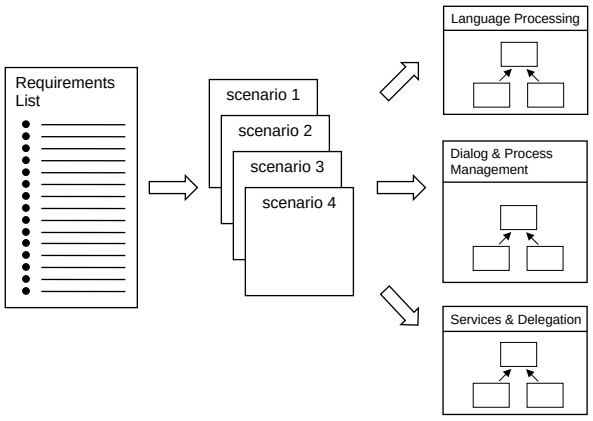


Figure 6.1: From requirements to Active Ontologies

Language processing First, the application domain, in other words *what can be said*, is modeled as an Active Ontology in charge of language processing (see section 5.2.3 on page 71). During this phase, the structure of valid utterances is designed through a collection of interconnected nodes to build a semantic network. The logic of high level structures is defined with non-terminal nodes,

matching rules and vocabulary sets are attached to leaf nodes. For applications requiring large vocabularies, typically more than 1000 words (see section 7.3.2 on page 182 for details), a database connection and data collection process is planned. Also, at the end of this phase, a set of language parsing regression tests are created. The tests are intended validate the language processing component of the system throughout the development and validation of the application.

Application logic Secondly, the processing logic of the application is designed. Actions and their relationships (i.e. sequential, parallel, conditional) are extracted from scenarios to model an Active Ontology in charge of running the dialog and plans undertaken by the assistant (see section 5.4 on page 109).

Services In parallel, as actions are defined, a third Active Ontology is modeled to represent service categories and register service providers. This phase consists of creating, or reusing, services in charge of interacting with the environment. Our applications typically have four types of services: *user interface*, *data source*, *processing nodes* and *effectors*.

User interface. Also drawn from the scenarios, the user interface involves services. In our applications, the user interface can be a single thick application, a web page or a combination of services (i.e. speech recognizer, text to speech, motion tracker) providing input and output interaction modalities. Note that, from the user point of view, the user interface is the tip of iceberg designed to hide the complexity of the application through natural delegation of tasks.

Data source services. Data source services are used in information retrieval applications. For instance, our web-based activities assistant (see section 6.2 on page 127) answers questions such as *"find me cheap Italian restaurants in Sunnyvale California"*. Behind the scene, after the language processing has produced a result and a plan is executed, services implemented as web scrapers, are used to retrieve and combine information from various web sites.

Processing services. Third, processing services encapsulate extensive processing needed by the application to get intermediate results. For instance, our operating room assistant (section 6.3 on page 141), provides a gesture recognition service that takes the trajectory of a hand motion (captured by a 3D sensor) to convert it into a known symbol.

Effector services. Finally, the fourth class of effector services provide actions that modify the environment. For instance, the operating room assistant controls a robotic arm that holds an endoscope, as other prototypes are able to make reservations and send emails on behalf of the user.

Deployment The final step consists of deploying all three Active Ontologies and testing their interactions from language processing, via plan execution to service invocation. Note that each type of Active Ontology can be designed, implemented and tested independently.

Responsiveness

An important feature of interactive user-centric systems is its responsiveness, expressed as the response time to a request. After any user input, depending on the interaction type, applications need to provide a feedback within an acceptable time range[80, 17].

What is *acceptable* depends on user expectations, which are based on the interaction model, modalities and the type of application. Interaction models can be synchronous (instant messages, thick desktop clients), asynchronous (email, text messages) or web browser-based. Modalities range from desktop computers to mobile devices. Finally, applications implement different types and levels of assistance. For instance, our operating room assistant provides a very straightforward assistance, with very little delegation. In such case, a fast response time is expected by users. On the other hand, our web activities assistant offers a deeper level of delegation, involving complex tasks such as retrieving information from various remote locations and even performing actual transactions (such as booking a reservation on behalf of the user). In this type of application, users do not expect an immediate response and are accepting to wait more before getting a response to their request.

Even if our goal is to provide response times as short as possible, the range of acceptable delays is defined on an application basis. For web related applications, Nah has shown that responses should not take more than 30 seconds[60]. However, literature shows that long queries, intermediate feedback should be provided to acknowledge reception of the query, understanding of the task to perform or even partial results [38, 67]. For non web-based applications, where all services and components are local and commands rather simple, such as our operating room assistant, users expect a much faster response time of about one second.

6.1.3 Software design

At this point, we have presented the constraints (functional and responsiveness) taken into consideration when designing Active-based applications. The present section is more technical and describes the overall architecture of an Active-based systems (see figure 6.2). Applications consist of the following software components:

Core server At the heart of the system, the Active Server hosts and executes a set of Active Ontologies. Our prototypes are typically made of two sets of Active Ontologies. First, we incorporate a group of generic system ontologies, used in all applications to perform basic tasks such as invocation, delegation and notifications tasks. A second set of ontologies, specifically designed on an application basis, provides language processing, dialog and plan management and service registration and execution.
In addition to hosting and running Active Ontologies, the Active Server uses a set of extensions. Extensions use an SDK-based plugin mechanism to extend the capabilities of the server. For instance, extensions have been developed to provide SOAP or REST communication capabilities. The SOAP extension is used to connect any service described with WSDL files, and the REST plugin has been used to build web-based user interfaces. Active-based techniques, such as language processing, come with extensions that encapsulate algorithms such as reference resolution and disambiguation, as well as database connectivity to store large vocabulary sets. To integrate multiple interaction modalities, extensions have been designed and implemented to provide email and instant messenger services. Using the email extension Active applications can read and send emails using POP and SNMP protocols. Additionally, the instance

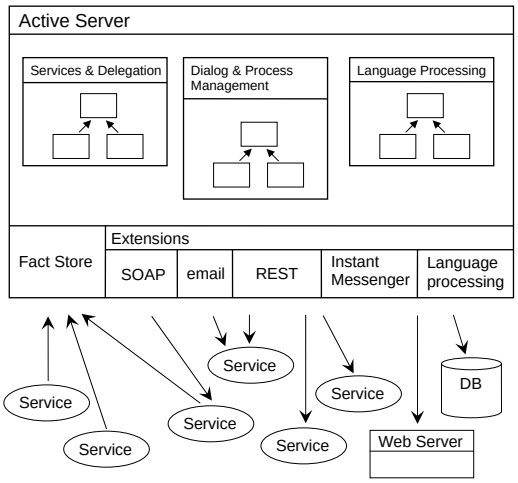


Figure 6.2: Application Design

messenger extension allows Active-based systems to be impersonated as a valid instant messaging user that can be contacted anytime for a chat-like interaction dialog.

Community of services The design of Active-based systems is service-oriented. In this context, the core server deals with a community of services in three ways. First, it is fed by sensor services listening for user inputs and relaying any relevant event happening in the application environment. Second, a set of effectors are used to undertake actions such a delivering information through a user interface or physically impacting the application environment (i.e. robotic components). Finally, a third class of services work as information sources. Such sources are used at the end of processing as the data for retrieve to server a user request (i.e. list of restaurants), or during the core processing for help (i.e. disambiguation and validation for language processing).

Development and administration tools Finally, being services themselves, the Active Editor and Active Console can join the community anytime to provide debugging, testing, tuning and monitoring.

6.1.4 System evaluation

This section provides a set of components to test for the validation of Active-based applications.

Language processing First, the language processing part of the application is tested using the regression test parsing tool (see Language Processing Test Console on page 92). Using the scenarios, a collection of regression tests have been written and successfully run against the semantic network of our application.

Functionality The second step consists of testing the functionality offered by the modeled workflow and the underlying services. This step is currently not automated. It is done by manually executing all the scenarios, step by step, to ensure that the data retrieved and presented to the user is valid.

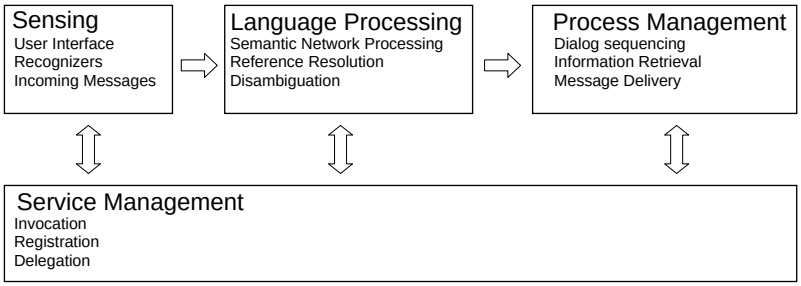


Figure 6.3: Overall response time

Responsiveness Finally, the responsiveness of the final system has to be within the time range specified for the application. The responsiveness, or response time, of the system is the sum of all components involved in the process. The whole sequence can be decomposed into four steps.

Sensing In this phase, a user interface gathers user inputs, or a sensor picks a signal from the environment and reports it to the Active Server. This is typically very fast, unless a some processing is performed by the sensor such as image processing or voice recognition. In such case, the processing has to be taken into account to compute the overall system response time.

Language processing Once an event is gathered from the environment, the Active Ontology in charge of language processing fires to determine if the reported event generates a new command. Depending on the size of the semantic network, and the use of external services for semantic validation or disambiguation, this processing time can take a significant time, typically ranging from one to five seconds. (See section 7.3.2 on page 182 for a detailed performance evaluation)

Process management Once the language processing node has generated a valid command, the Active Ontology in charge of process execution (or dialog management) executes the command by calling, sometimes through delegation, external actuator or information source services. Processes execution are not intense and require a limited number of simple rules to evaluate. Therefore, they do not have a significant impact on the overall response time of the system.

Service invocation Across all elements enumerated so far, services may be invoked. At the sensing stage, a gesture recognition service may be invoked to transform raw input signals into a know symbol. At the language processing level, services may be called for reference resolution or disambiguation. At the

core of the application, the process management Active Ontology will invoke services to perform user requests and send out notifications.

Services to invoke include user interfaces, data retrieval services to actual physical devices. Therefore their response time varies greatly and their impact on the system response time has to be evaluated on a case by case. For instance, for applications that requires access to online information and services, calling services will significantly impact the overall response time of the system. On the other hand, for applications where all services are local, the responsiveness will not be affected by service delegation and invocations.

6.1.5 Conclusion

This section introduces the guidelines followed to design and implement Active-based applications. First, the design of an application consists of defining functional requirements and the expected responsiveness of the system. To crystallize the functional requirements expressed by end-users, a set of scenarios are created. From these scenarios, three types of Active Ontologies are drawn : language processing, plan and dialog representation and service management. Scenarios are also used for user interface design and the definition of all underlying backend services.

This section also presents the main software components of Active-based systems, consisting of a core server hosting set of Active Ontologies and a community of loosely coupled services.

Finally a section provides guidelines about evaluating implementations against the set of requirements used during the design phase.

6.2 Online Activities Assistant

6.2.1 Introduction

This section proposes to study how natural language queries combined with context and dialog can improve online activities for mobile users.

As an actual tool for this exercise, we describe the prototype of an assistant that helps users with online activities. The system interacts using a combination of natural language and dialog to retrieve information and perform transactions. Multiple user interfaces (browser, instant messages, email or thick dedicated client) allow users to express queries in plain English to delegate complex tasks. For instance, users can use any of the supported interfaces to ask questions: *"find action movies near san jose"* or request actions: *"book me a table tonight at Joe's pizza in Palo Alto"*.

The section is organized as follows. First, we describe the goals behind the design and implementation of this prototype. Then, we present the system requirements. As described in section 6.1 of this chapter, a list of requirements is translated into scenarios used to implement a set of Active Ontologies, a community of services and a user interface. The third part of this section presents the implementation of the prototype in details. The fourth section provides a two-phase evaluation of the system. First, functional requirements, including responsiveness, are verified to demonstrate that the Active platform and its programming techniques can be used to model and implement end-to-end user-

centric applications. Secondly, our prototype is compared with other similar existing systems (Google Mobile^{TM}, Ask Mobile) to prove that natural dialog improves the experience of mobile users for online activities. Finally a conclusion summarizes what was achieved and presents the results of our evaluation. The conclusion also describes the positive feedback received from the commercial world about our research, and how our prototype is inspiring the design of a commercial intelligent portal for mobile users.

6.2.2 Prototype goals

This prototype has two goals. First, it is a tool to validate our approach of using a unified framework, the Active platform, to build applications encompassing language processing, dialog management and service orchestration. Secondly, we intend to demonstrate that a combination of natural language and context-based dialog can improve the way mobile users interact with online information and services.

Validation of the Active approach The prototype is intended to demonstrate that an end-to-end user-centric system can be built using the Active framework. More specifically, three aspects of our approach are validated : *methods*, *implementation* and *application design*. First, the methods described in chapter 5 to build Active Ontologies providing language processing, dialog management, process execution and dynamic service invocation are going to be used and tested in the prototype. Secondly, at the core of the system, the current implementation of the Active platform is going to be measured and tested.

On the programmer side, both the Active Editor and Active Console will be used to edit, deploy and test Active Ontologies. On the backend side, the Active Server and its extensions will host and run a set of Active Ontologies, to react and provide valid responses to users within the specific functional and time constraints of the application. Finally, the effectiveness of the design approach introduced in section 6.1 on page 122 will also be put to the test.

Improved mobile user interaction The second goal of the prototype is to evaluate how mobile users can improve their online-experience through a more natural interaction scheme. As mobile phones and devices are getting more affordable and powerful, more mobile users need to access online information and services. To accommodate the specific constraints of mobiles devices, limited screen size, low usability and reduced bandwidth, popular search engines and portals offer specialized interfaces.

Adapting the user interface to mobile devices is a first step, we believe that interaction techniques also need to change. The current context-less and keyword-based paradigm to retrieve online information has limitations. We propose to use a combination of natural language and context to not only ease online information retrieval, but also allow users to actually use online services to execute transactions such as purchasing good or booking reservations. This goal is aligned with the commercial world, which has determined that mobile Internet activities show a tremendous potential to not only offer paying online services, but also in terms of geolocated and contextual advertising.

For instance, a traveler may want to check the status of a flight from a cell phone. Instead of getting online from an embedded browser with limited capabilities, we propose to type a message, in plain English such as *"Is flight UAL 1978 on time?"*. In return, the user gets not only the requested information, but added contextual suggestions such as *"Flight United 1978 is delayed. Would you like to find a restaurant near the airport?"*

The goal of this prototype is to explore new ways of conducting two types of online activities: *information retrieval* and *transactions* over simple *context-based dialogs*.

For information retrieval, in addition to conventional use of keywords, we propose to retrieve data from the web through a natural language-based dialog with an intelligent information retrieval assistant. Looking for information on the web using keywords is effective for simple *navigational* queries, where users are searching for specialized web sites or portals. Chai demonstrated that, in the field of intranet or information retrieval over large sets of data, natural language and dialog based interaction is faster and more effective than menu or keyword-based queries[87].

Broder[6] showed that in addition to *navigational* searches, a large number of queries are *transactional*. In this case, users do not have a specific portal or web page in mind, but are trying to retrieve a specific piece of information off the Internet to undertake transactions. By transaction we mean actual actions such as purchasing goods or booking reservations.

Unlike conventional keyword-based search engines that return URLs lists, our deeper understanding of the user intentions allows us to provide relevant and contextual information. In this context, our domain specific assistants perform much better than generic keyword driven search engines. This claim is demonstrated through a user study presented in section 7.1 on page 161.

In addition to support navigational and transactional interactions, we intend to support context over simple dialogs. For instance, one might want to express queries such as: "*what thrillers are playing tonight in palo alto*?" followed with "*is there any chinese restaurant nearby?*".

Going online using a search engine to answer these questions, sometimes from a mobile computer or cell phone, may be a challenge. Our intention with the prototype presented in this section is to provide a tool where such queries could be expressed *as-is*, in plain English, using any modality at hand such as email, instant messages or a regular browser if available. Answers would also be delivered and formatted to the right modality to fit the devices of the user.

6.2.3 Requirements definition

This section presents the requirement list and a sample scenario used to model the set of Active Ontologies that power the application.

First, let us describe the overall domain of our applications. The system offers *information retrieval* capabilities covering restaurants, movies, points of interests, weather and flight information in the US. Along with retrieved information, our system makes pertinent *suggestions* and can undertake *transactions* on behalf of the user. Finally, in addition to retrieving relevant information, our system is able to perform actual transactions such as booking reservations. The following list enumerates the main functional constraints of the system to implement:

- Information retrieval through dialog-based interaction. The main feature of our system consists of retrieving online information through queries expressed in natural language. For instance, queries such as *"find best italian restaurants in palo alto"*, *"what is the weather in miami"* or *"is flight united 981 delayed?"* should return relevant information.

- Context management. As queries, or utterances, are submitted to the system, a context is incrementally built. As presented in section 5.2.3 on page 71, the use of semantic networks for language processing naturally maintains a context over the nodes and communication channels of the network. For instance, the sequence *"find restaurants in memphis"* followed by *"give me only italians"* should first return the list of all known restaurants in Memphis, where the second utterance simply adds a constraint to domain and results in a shorter list of solutions.

- Transactions. In addition to retrieving information, the system is able to undertake transactions on behalf of the user. After getting a list of restaurants through a query such as *"find best italian restaurants in palo alto"* , one may wish to actually book a table. Our system should offer that functionality by accepting an utterance of the form : *"good, book me a table tonight at Joe's pizza for two people"*. Helping users to not only make a decision based on the most relevant information, but also carry out actions and transactions is the essence of a good assistant.

- Semantic validation. Semantic validation performs validation of the information collected by the system. In the present prototype, two types of validation are required : *addresses* and *airport names*.
  In the case of addresses, validation is used to automatically fill missing or adjust information slots. For instance, if a valid zipcode is specified, the city name and state can be known. If a city name is specified, the most likely zipcode and state should be inferred. If there are multiple solutions, the user should be notified about the existence of another match.
  For airport names, a known three-letter airport acronym should be resolved as a valid address, and symmetrically, a known city name should be translated into the most likely airport code.

- Contextual suggestions. The assistant should not only perform tasks delegated by users, but also proactively suggest relevant followup actions. Such anticipated options, also called *golden delights*, would for instance check among restaurants listed which ones offer online reservation and offer the possibility of performing a reservation on behalf of the user. Other suggestions would for instance consist of providing a list of nearby florists or ATMs when booking a table at an expensive French restaurant.

- User interface. To satisfy the needs of mobile users, the interface needs to be lightweight and intuitive. At least two modalities tailored for mobile users need to be implemented: browser-based interface and email-based dialog.

To illustrate the constraints defined above, a set of scenarios has been created directly with potential end-users of the application. As an example, figure 6.12 shows a simple interaction sample.

Utterance #1	***find restaurants in palo alto***
User feedback	Provides a list of restaurants in Palo Alto, CA and provides relevant suggestions. (bookings, nearby points of interests)
Utterance #2	***get me only french ones***
Actions:	Provides an updated list of french restaurants only and relevant suggestions.
Utterance #3	***book a table tonight at il fornaio at 8:30 pm***
Actions:	Provides a booking confirmation and sends an email.
Utterance #4	***find nearby florist***
Actions:	Provides a list of florists in Palo Alto, CA
Utterance #5	***what is the weather in miami***
Actions:	Provides a forecast and current conditions in Miami, Florida. Also suggests that there are Miamis in different states.

Figure 6.4: Mobile Assistant Sample Scenario

In addition to functional requirements, performances constraints have been defined for the project.

- Responsiveness. As shown in section 6.1.2 on page 123 of this chapter, the responsiveness of an application depends on the context and user expectations. This prototype retrieving information from the Internet through complex queries, users do not expect an answer to come back immediately. According to the litterature [67, 38, 60], it is nevertheless necessary to provide feedback (confirmation, partial results) after 5 to 10 seconds, even if the overall response time of the system takes more time.

- Large vocabularies. The application needs to deal with large vocabulary sets and yet provide fast response time. For instance, address validation and related suggestions, the system needs a list of all major US cities. Similarly, since we would like to provide answers to utterances such as *"good, book me a table tonight at Joe's pizza for two people"*, a large list of restaurant names also needs to be part of the application vocabulary.

- Multiple users support. The system should be able to serve multiple users at the same time. In addition of being practical, this feature will allow us to study how our system behaves as its processing load grows. All components of the Active framework will be monitored to pinpoint potential bottleneck and provide suggestions about how to improve the system overall performance.

6.2.4 Implementation

The assistant described so far has been fully implemented. Figure 6.5 shows screen captures of the application actually run from a mobile phone emulator. We choose to evaluate our system via an emulator for practical reasons. First, it was easy to install the emulator on subjects machines to let them run the

evaluation when ever convenient. Also, the emulator was instrumented to log and time stamp all user requests and responses from tested systems.

The three examples show queries about finding restaurants, weather forecast and flight information. This section details the implementation of the system,

Figure 6.5: Screen shots of the "Active Mobile" application

describing its overall architecture and most important components.

Overall design

The overall design of our application, shown in figure 6.6, consists of two main elements : the Active Server and a community of loosely coupled services. At the center of the system, the Active Server and its extensions host and run three types of Active Ontologies: language processing, process management and service management. At the periphery of the system, a constellation of external services are dynamically orchestrated to interact with users, provide information and perform processing tasks.

Before talking in more details about individual elements, let us examine events and processing flow of the system. Processing usually start from a UI related service reporting a user utterance. The utterance triggers an evaluation of the language processing Active Ontology, which produces a command. If valid, the command is passed to the Active Ontology representing dialog management, which undertakes a series of actions to respond to the user request. As the plan unfolds, external services are dynamically selected and invoked to provide information or run transactions. Throughout the process, UI related services keep the user informed about results.

User Interface

Leveraging our service oriented design, the prototype supports four different types of user interface. Thick client interfaces, such as the language processing test console introduced on page 92, use SOAP messages to submit utterances

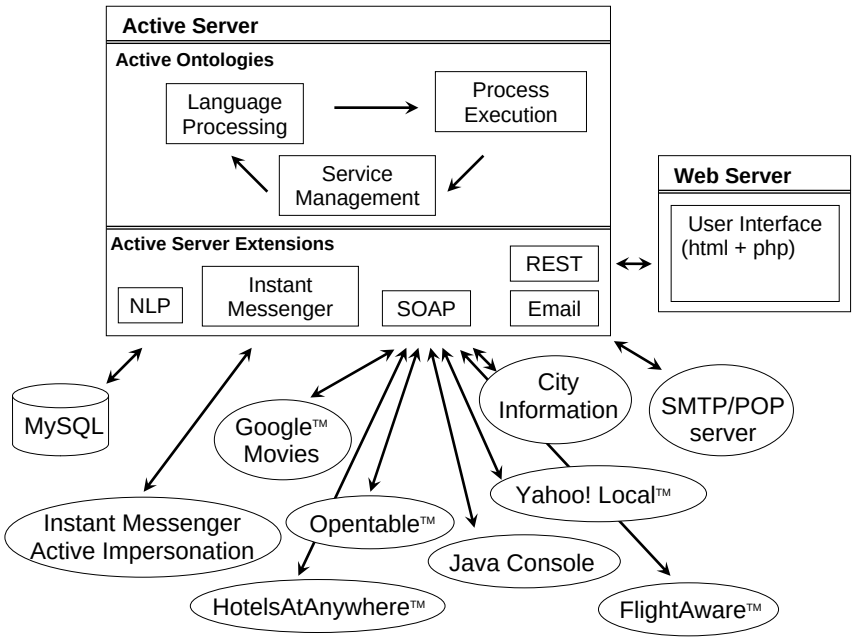


Figure 6.6: Online Activities Assistant Overall Architecture

and display results. Using the Yahoo Instant Messenger^{TM}extension, anyone with a valid account can interact with our prototype through short instant messages. Through the Active Server email extension, the system is capable of checking for incoming emails (using the POP protocol) and deliver information by responding directly by email to the user. Finally, using the REST API exposed by the Active Server, a web-based user interface allows interaction through any web browser.

There are two classes of user interfaces : *asynchronous* and *synchronous*.

	Asynchronous Interaction	Direction	Time line
#1	*What is the weather in sunnyvale?*	User -> Assistant	Start
#2	Thank you, processing your request	User <- Assistant	Start + 2 [s]
#3	Getting weather forecast in Sunnyvale, CA	User <- Assistant	Start + 3 [s]
#4	Five day forecast in Sunnyvale, CA Today: fair, temperature 60 F Monday : sunny, temperature 70 F	User <- Assistant	Start + 5 [s]

	Synchronous Interaction	Direction	Time line
#1	*What is the weather in sunnyvale?*	User -> Assistant	Start
#2	Thank you, processing your request Getting weather forecast in Sunnyvale, CA Five day forecast in Sunnyvale, CA Today: fair, temperature 60 F Monday : sunny, temperature 70 F	User <- Assistant	Start + 9 [s]

Figure 6.7: Asynchronous vs Synchronous interaction

Asynchronous interfaces can receive information asynchronously, at any time, from our application. The top table of figure 6.7 shows the sequence of messages when using an asynchronous interfaces, where the assistant sends four separate messages as its processing takes place. In our prototype, both the instant messenger and thick SOAP-based client allow for asynchronous messaging.

Synchronous interfaces cannot receive messages asynchronously, they can only receive information as a synchronous response to a request. The lower table of figure 6.7 represents a synchronous interaction, where a single message is delivered as the result of an explicit request. The web-based interface of our system is synchronous.

A note on the last modality offered by our system : email messages. Use of email is asynchronous by nature, it is nevertheless treated as a synchronous modality by our system. It would not be practical to receive four separate email messages as a response to a simple request. Therefore, our system waits until all information is retrieved before delivering the response email message.

The implemented prototype dynamically picks the interface to use when sending messages to the user using the delegation technique described in section 5.3.3 on page 106. The current selection algorithm is very simple: when the system needs to deliver information to a user, it picks the modality used by the user when the last utterance was received.

Language processing

The front line of our application is the Active Ontology in charge of language processing. It has been implemented on the *semantic networks* technique described in section 5.2.3 on page 71. The network created is rather large (about 47 leaf nodes, 20 gather nodes and 5 select nodes) and uses all the features offered by the semantic networks technique (see figure on the following page).

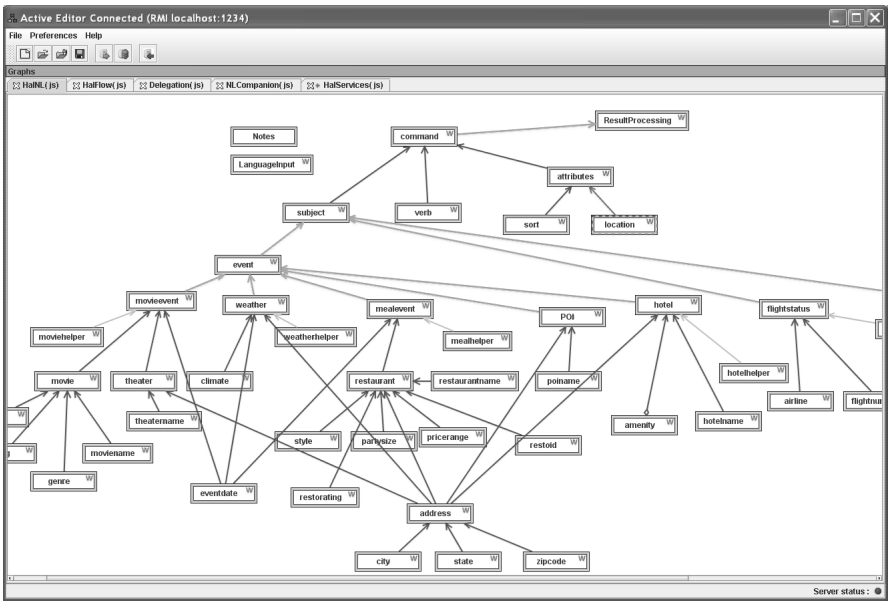


Figure 6.8: Semantic network in the Active Editor

Leaf nodes The prototype leverages all types of leaf nodes. The **actor** leaf uses the prefix *with* to rate incoming words as actors, the **zipcode** node uses regular expressions to detect zipcode and the **state** node uses a small local vocabulary set to claim US states. In addition, the **city name** and **restaurant name** nodes are connected to a database to check against large vocabularies containing major US cities (30'000 entries) and restaurant names (100'000 entries).

Non-leaf nodes The network contains five *select* nodes, used to pick the main topic of interest expressed by the user. Given the large size of our vocabularies, the disambiguation process described on page 82 is helping select nodes make the correct decision. For instance, let us consider the utterance *"find hotels with a fitness center in Mountain View"* triggers the disambiguation process. The word *"fitness center"* is both a point of interest, *"find a fitness center in Palo Alto"* is valid, and a hotel amenity. In this particular example, our system will consider a hotel rather than a point of interest because the hotel branch has three tokens, whereas the point of interest only has two words activated for the utterance.

A set of *gather* nodes are used to create data structures containing the information gathered through user utterances. Two of them request semantic validation before submitting their ratings**.**
First, the **address** gather node, in charge of collection city name, zipcode and state information, asks for validation not only to check if a zipcode matches a city name, but also automatically fills empty information slots. For instance, the utterance *"find Starbucks for 94040"* triggers the zipcode leaf. The **address** node gathers empty values from its city name and state nodes, and asks

for validation before submitting its semantic rating. The validation agent is implemented as a service and uses third party services and local databases to not only fill in missing slots (94040 would resolve to Mountain View, CA), but also provide suggestions. For the utterance "find movies in memphis", as they are multiple Memphis in the US, the validation agent picks the largest one (Memphis, Tennessee), but also suggests names and state of other matching city names.
Secondly, the **flightSearch** node validates airport names. If a three-letter airport code is provided (i.e. SFO or GVA), it requests help for validation. In addition, should a full city name be provided, the validation service converts it into the appropriate airport code (i.e. Geneva would become GVA).

Infrastructure nodes As for any semantic network-based Active Ontology, two generic concepts are in charge of receiving utterances and passing along valid commands for execution. These nodes also take early actions to notify users about the process taking place, which is relevant for asynchronous user interfaces. As an utterance is received, before any processing takes place, the delegation mechanism is used to acknowledge the reception of the utterance. Similarly, as a valid command is generated by the language processing Active Ontology, a message is sent back to the user to notify what was understood by the system and which actions are going to be undertaken. These actions, relevant for asynchronous user interfaces, generate the message #2 of the top table on figure 6.7.

Processes

A second Active Ontology has been designed to model the logic of the application. Based on the technique presented in section 5.4 on page 109, it represents the sequence of actions to undertake to fulfill user requests. The logic of our application is rather simple and figure 6.9 shows the pattern used for most actions.

- First, a *start node* is programmed to trigger whenever a command to execute is submitted by the Active Ontology in charge of language processing.
- A *switch node* examines incoming command to find out which processing branch to activate. There is one branch per processing task (retrieving hotels, restaurants, flights, movies, booking a table, etc...).
- The first element of each processing branch is an *execution node* in charge of preparing all necessary flow instance variables for the task to accomplish. Some variables are extracted from the incoming command. Each leaf of the language processing semantic network is part of the generated command and therefore carries valuable information to be extracted and used by the processing logic. Some variables are created specifically for the task to carry out. For instance, information to send back to users is stored in two flow instance variables. A *message* variable is prepared to notify users about what our assistant is about to undertake and an error message keeps the message to deliver should the processing fail.
- Next, an *invocation node* activates a notification service to deliver a message to the user. Note that there are multiple services registered under the

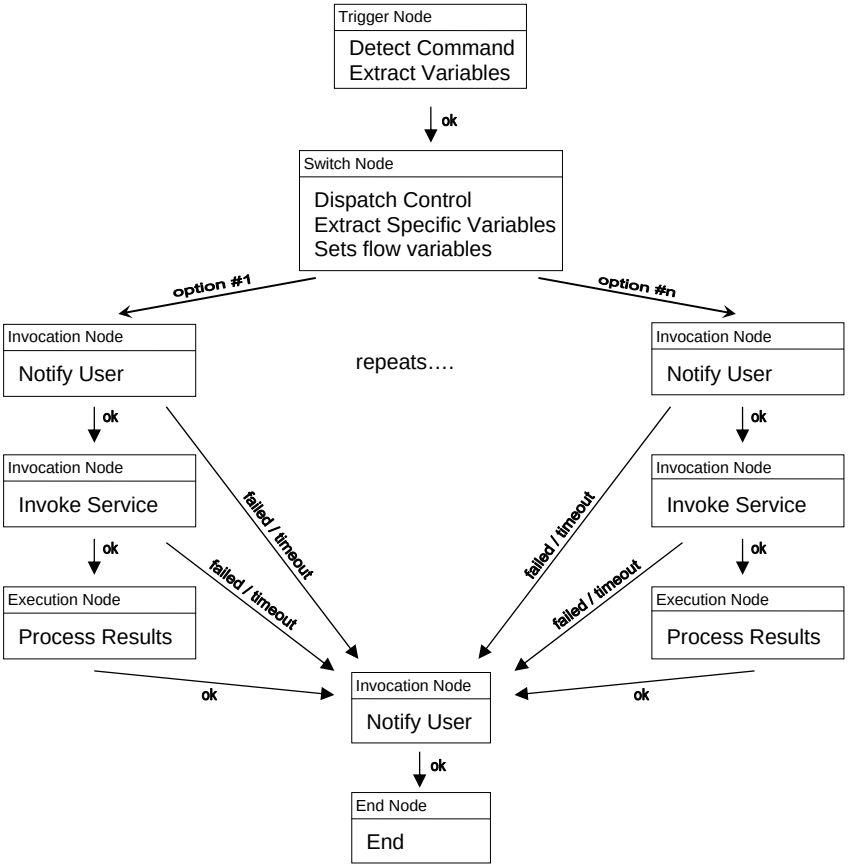


Figure 6.9: Information retrieval execution pattern

notification service category. The dynamic selection technique presented in section 5.3 on page 96 selects the most appropriate one to deliver the message. In our case, the policy is simple: the picked modality is the same the one from which the utterance was received. If the user request came as an incoming email, the response will be an email. At this stage, the delivered message is a confirmation of what was understood and what our system is about to perform. It is the message #3 of the top table on figure 6.7.

- Once the user has been notified of our intentions, a second *invocation node* requests the execution of a service that fulfills the core processing of the execution branch. There, we would invoke services in charge of retrieving movie listings, weather forecasts or flight information.
- If the service invocation succeed, an *execution node* retrieves the results of the invocation to prepare the message to be sent back to the user. Results would be for instance a list of movies, restaurants or hotels. The logic

of this node would iterate through results and format the message to be shown to the user.

- The control then goes to an *invocation node*, calling the notification service to deliver the message.
- Finally, an *end node* performs cleanup tasks to terminate the execution of the flow instance.
- At any point in the process, should any service invocation fail or timeout, a dedicated execution node would prepare the message to inform the user about the problem.

Services

The implementation of the service management part of the application is a three-step process. First, in the Active Ontology in charge of service management, service categories are designed. Categories expose an API that reflects the data types required by our application. For instance, figure 6.10 shows a restaurant

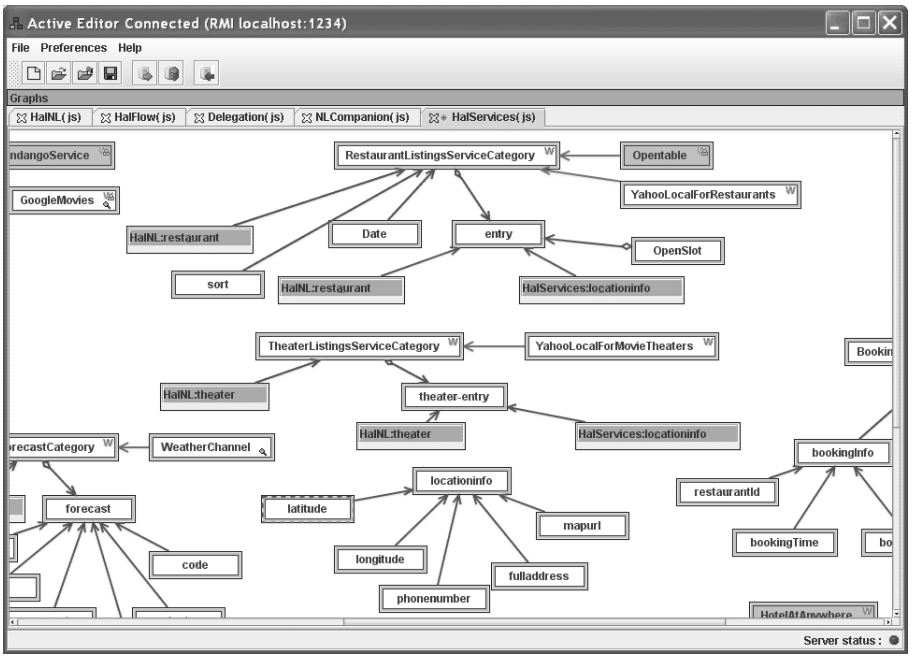


Figure 6.10: Service Category in Active Editor

information retrieval service category. The goal is to get a list of restaurants for a given city, and possibly know when a table is available. Some attributes may be specified about restaurants (i.e. style, price range) a sorting attribute (i.e. distance, rating, price) controls the order of entries in the result set. Therefore, inputs the service takes a restaurant definition, a date and a sorting attribute. As outputs, services of this category return a list of entries where each entry contains a restaurant object, location information i.e. latitude, longitude, phone

number) and a list of openings (date and time of availabilities). Note that the details of the restaurant object are not defined here, there is a reference to the *restaurant* node as defined in another Active Ontology in charge of language processing.

The second step consists of getting hold of a service that provides the required functionality. In the case of this application, we have created two types of SOAP compliant services: *web scrapers, database wrappers.* Web scraping consists of writing a program that automatically extracts information from online sources by simulating the behavior of a web browser. The technique used in this prototype consists of writing web scrapers (typically in Javascript) and expose them as SOAP compliant web services. Some services, such as city or airport information, are implemented on top of large databases. In this case, a thin wrapper (typically written in Java using JDBC) exposes the database content as a SOAP service.

Once a service has been created, the third step consists of registering it under service categories defined during phase one. For SOAP compliant services, the Active Editor provides a wizard that simplifies the service provider registration process. After picking the service category where the service belongs, a set of registration steps including WSDL introspection, graphical interactive mapping between the service-specific API and the canonical API defined by the service category and testing allow users to register services without writing any code. A similar process applied to all services and categories of the application.

6.2.5 System evaluation

The evaluation of the implemented system is a two-phase process. First, we ensure the actual system fits the functional and responsiveness requirements defined at the design phase. Secondly, we compare our system against the best commercially available tool designed for mobile users (Google Mobile) by asking a small population of users to perform similar tasks with both systems. The goal of this is to evaluate whether an assistant-like software is likely to compare favorably to more traditional software.

Compliance with initial requirements

This section explains how our prototype fully complies with its requirements. As discussed earlier (see section 6.1.4 on page 125), the validation of the system is a three-step process.

First, using the application scenarios, a collection of language processing regressions tests have been created to cover as many interactions dialogs as possible. The regression tests have been regularly and successfully executed against the system to verify its compliance with the devised scenarios.

Secondly, beyond language processing, users have extensively used the system by running scenarios. Problems were reported to correct errors in both the Active Ontology in charge of dialog execution and the community of services used as the system backend.

Finally, the overall responsiveness has been measured to ensure that users get a feedback from the system with the time range specified in the application requirements. The average response time to an end-to-end query such as *"find*

movies in Palo Alto" or *"get my all flights out of SFO to Los Angeles"* is below 10 seconds.

Component group	**Response time**
Input gathering	10 [ms] to 10 [s]
Language processing	300 [ms] to 4[s]
Process/dialog execution	200 [ms]
Service Execution	400 [ms] to 20 [s]

Figure 6.11: Contributions to the responsiveness

Following the guidelines provided in section 6.1.4 on page 125, we measured the contribution of four main components of the application (see figure 6.11). It is nevertheless important to note that the response time varies significantly depending on the modality, the utterance size or services to contact.

First, the contribution of the input gathering process depends on the user interface. For instant messenger, thick Java client or web-based interface, the information reached Active within milliseconds. On the other end, for the email modality, messages can take up of 10 seconds to be reported to the Active Server. This is not an issue given the asynchronous nature of emails. The user sends an email and moves on to different activities, expecting the answer to be delivered later in his or her mailbox.

The contribution of the language processing task ranges from 300 milliseconds up to four seconds. Multiple factors impact the time spent processing user inputs. They include the size of the semantic network, the size of the utterance, the semantic validation techniques and the vocabulary size. A detailed analysis of the language processing performance impact is provided in section 7.3.2 on page 182.

The contribution of running the dialog and formatting the messages is negligible. The impact of using external services varies from less than 1 second to 20 seconds. Services in charge of delivering information to user interfaces take between 400 to 600 milliseconds, including the delegation process. On the other end, web-based information retrieval can be rather slow and take up to 20 seconds. Unfortunately, Active programmers usually have not control over these external services. The only solution to improve response time is to create a local cache, for storing static information. For instance, the results such as phone numbers, addresses or locations can be cached. However, dynamic information such as flights status, hotel openings or even restaurants ratings cannot be cached and require the invocation of an external service.

Figure 6.11 shows that the overall response time contribution of the Active platform itself ranges from 500 milliseconds to 4 seconds and averages over long dialogs at 3 seconds. Given that the average response time of the system is about 10 seconds, the Active platform is responsible for 30% of the overall response time. The current implementation of the Active platform was designed as a research tool, without a strong focus on performance and scalability. A complete performance analysis of the Active Server (see section 7.3) presents its strengths and weaknesses. It also provides a list of recommendations to take into account when developing an optimized and scalable version of our systems.

User evaluation

This prototype has been used to demonstrate that, in some domains, intelligent assistant applications perform better that more conventional software. A small population of users was asked to perform a series of travel related online tasks with both our prototype and Google Mobile™. Results show that our Active-base application, code named Active Mobile, performed better both in terms of time to completion and effectiveness that its commercial counterpart. This user evaluation is described in details in section 7.1 on page 161.

6.2.6 Conclusion

This section presents the first end-to-end application built with the Active platform. The application is an assistant that helps users with online activities over dialog-based natural interactions.

This prototype helps us to achieve two goals. First, it validates the Active platform it terms of both implementation and techniques. The actual prototype, implemented on top of the Active framework, is functional and complies with all its initial requirements. The application demonstrates that both the Active tools and methods can be used to build user-centric applications. Tools such as the Active Server and its extensions, the Active Editor and its plugins and the Active Console have been extensively used to design, implement, test and deploy the system. All methods introduced so far, including basic techniques (see section 5.1 on page 57), language processing (see section 5.2 on page 64), process modeling (see section 5.3 on page 96), service dynamic orchestration (see section 5.3 on page 96) and application design (see section 6.1 on page 122) have been used.

The second goal, consisting of demonstrating that a combination of natural language and context-based dialog improves mobile users online activities has also been reached. A small population of twenty users has been asked to perform the same tasks using our system and a commercially available similar tool (Google Mobile). Results show that, for this specific application domain, an intelligent assistant-like system provides better performances in terms of both task completion and required time (see section 7.1 on page 161 for details).

The prototype has also been validated by being positively evaluated by commercial companies. First, a major player in the field of mobile telephony evaluated the impact of such systems and decided to pursue research and development along the tracks suggested by our approach. Secondly, a large database vendor is evaluating our software in the context of building a assistant-oriented system to help users query and navigate large databases. Finally, a commercial system providing an intelligent assistant for mobile users is being designed, following many of the concepts demonstrated by our prototype, and based largely on the value proposition demonstrated by our user evaluation.

6.3 The Intelligent Operating Room

6.3.1 Introduction

Modern operating rooms are equipped with various computer systems, allowing surgeons to perform complex operations and develop new techniques to improve

results, limit the trauma of surgery on patients and shorten hospital stays. However, both the environment and users of this field make the integration and usage of computer systems a challenging task. The operating room has obvious and strict constraints about space and sterilization that prevent the use of classic keyboards and mice. In addition, surgeons and their staff often wear cumbersome outfits including sterile clothing, head lamps and gloves. The patient being the highest priority, surgeons always need to focus on the operating field. Therefore, they cannot afford to switch attention or drop their tools to interact with computer systems. According to surgeons, computers will be more effective and easily accepted if they can be seen as any other member of the team. This implies that computer-human interaction should be as natural and simple as possible. The operating room is an environment where an intelligent assistant approach could allow computers to be more efficiently accepted and used.

The section is organized as follows. First the requirements of the system are gathered to create regression tests and scenarios. This is followed by a detailed description of the system implementation. Next, an analysis of the system performance and the results of a user evaluation help validate the system against its initial requirements. Finally a conclusion summarizes our achievements and the results of the system evaluation.

6.3.2 Prototype goals

This prototype was designed to implement two goals. First, after showing that the Active platform can be used in the field online activities (see section 6.2 on page 127), we would like to demonstrate its flexibility by building a user-centric system in a totally different application field. Secondly, we intend to use the prototype as an evaluation tool to explore if and how intelligent assistants can help surgeons in the operating room.

6.3.3 Requirements definition

Based on the technique proposed in section 6.1 on page 122, the application design starts from creating a requirement list and building a set of scenarios. Based on preliminary discussions with surgeons, we characterized the system along two lines: *feature set* and *interaction modes*. Let us start with the feature list:

- Information retrieval. The system allows surgeons to retrieve and manipulate preoperative data: a set of CT scans and a reconstructed 3D model of the area to operate.
- Live real-time feedback. A video feed coming from an image source (endoscope or microscope) is to be displayed along with vital patient information. Optionally, additional data streams are added to inform about the patient's condition or any relevant information required by surgeons. For instance, neurosurgery can be conducted while monitoring the patient's brain activity through real-time electroencephalograms.
- Robotic component. The video source is mounted on a powered platform, or a robotic arm, to be controlled through our prototype.

- Unified single interface. An integrated user interface needs to gather and render all the information in a single console.
- Intelligent space. The prototype should not only be aware of the user intentions, but also gather information from the operating room as a whole. In our prototype, we provide the assistant with patient related data (heart beat, blood pressure and body temperature) to not only be displayed on the user interface but also to be taken into account in its processing.

In addition to this feature set, interaction modes have also been defined:

- Multiple input modalities. In addition to the conventional mouse-keyboard inputs, the system should support voice and hand gesture (contact-less mouse [23]) recognition.
- Multiple output modalities. Symmetrically, when the system has information to communicate, multiple modalities should be supported. Messages can be delivered as text messages on the user interface or as sounds (beeps or speech synthesis).
- Modality fusion. Users should be able to express commands across modalities. For instance, one may say *"move the endoscope this way"* while gesturing to the left. The system should be able to fuse incoming signals to understand the user's intention and produce the correct behavior: moving the camera to the left.
- Context aware. The system should be aware of the interaction context. In our example, we defined three contexts: *camera*, *preop images* and *3D model*. Once in a given context, short partial commands should be enough for the system to understand the action to perform. For instance, the sequence : *"please move endoscope to the left"* should set the context to *camera*. Therefore, subsequent motion control utterances can simply be *"up"* or *"zoom in"*.
- Provide help. If users issue incomplete commands, the system should be able to provide contextual help. This feature is most important during a training phase where users discover and learn about the system simply by interacting with it.
- Robust parsing. Parsing should be robust and support disfluencies.

To crystallize the constraints defined above and give us a clear goal we defined a set simple scenarios. As an example, figure 6.12 shows a simple interaction sample.

Utterance #1	***ct show first image***
Actions:	Shows the first image of the CT scans. Context set to images.
Utterance #2	***next***
Actions:	Shows the next image of the list
Utterance #3	***camera***
Actions:	Sets the context to camera. Provides help: move or zoom?
Utterance #4	***hughh, move up***
Actions:	Camera moves up.
Utterance #5	***hand motion to the left***
Actions:	Camera moves to the left
Utterance #6	***model show top***
Actions:	Context set to the patient 3D model 3D model rotates to show the top of the organ
Utterance #7	**say: *ct show* and hand gesture: to the right**
Actions:	Context set to images. Shows the next image of the list

Figure 6.12: Evaluation Corpus

Finally, we defined the responsiveness of the system. The nature and user expectations of the operating room assistant differ in many ways from the online activities assistant presented in section 6.2 on page 127. First, all components are local, there is therefore no remote call to potentially slow external services. Secondly, the tasks to be perform by the assistant are far less complex. According to the scenarios they consist of very simple operations such as retrieving images or moving a powered camera. The system to create provides assistance, while the user is still in charge of all operations. It is similar to a modern airplane that helps the pilot fly, but is by no means an autopilot. In addition, user expectations are high. Given the application field, surgeon cannot afford, and will no accept, waiting for their requests to be understood and performed. The response time has therefore to be short, ideally a few hundred milliseconds, but no more than a second.

6.3.4 Implementation

Overall design

Following the Active design approach, the system consists of an Active Server and a community of loosely coupled services (see figure 6.13). The core of the application is based on three Active Ontologies running on the Active Server. They implement the behavior of the intelligent interface: language processing, plan execution and interaction with the environment. A community of loosely coupled services makes up the rest of the application by sensing the environment (speech and gesture recognizers, stereo camera, user interface) and acting (user interface, speech synthesis and optionally a robotic arm).

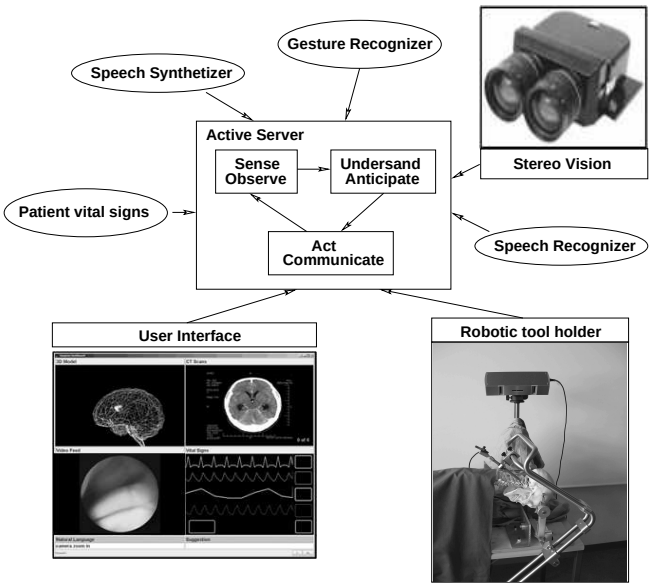


Figure 6.13: Intelligent Operating Room Architecture

When a sensor gathers a piece of information from the environment, it reports it by asserting a fact into the data store of the language parsing Active Ontology. This event triggers the evaluation of running Active Ontologies that will generate the most appropriate action to perform what the user asked. Note that the system is not only aware of the surgeon's activities, but also gathers information about the condition of the patient and the status of various devices running in the operating room. It aggregates this information in its global behavior to, for instance, warn the surgeon when the patients condition changes. As more components get integrated, the Active based surgery assistant has the potential to transform the operating room into a smart intelligent space.

User interface

The main user console is implemented in Java using the Swing toolkit, Java3D and Java Multimedia media libraries for 3D rendering and live video feed display. As shown in figure 6.13, it consists of four main areas: live images delivered by the endoscope, pre-operative images, a 3D model and general information about the current condition of the patient. If the user interface is the only component with which the user directly interacts, it is actually only the tip of the iceberg. The console is just another service in the community of components working for the user. In addition to the console, speech and gesture recognizers, touch screen are also capturing and reporting user generated inputs.

Gesture recognition

Since surgeons cannot use any mouse nor keyboard while operating, we provide them with a virtual mouse pointer by tracking their hands motion. Based on

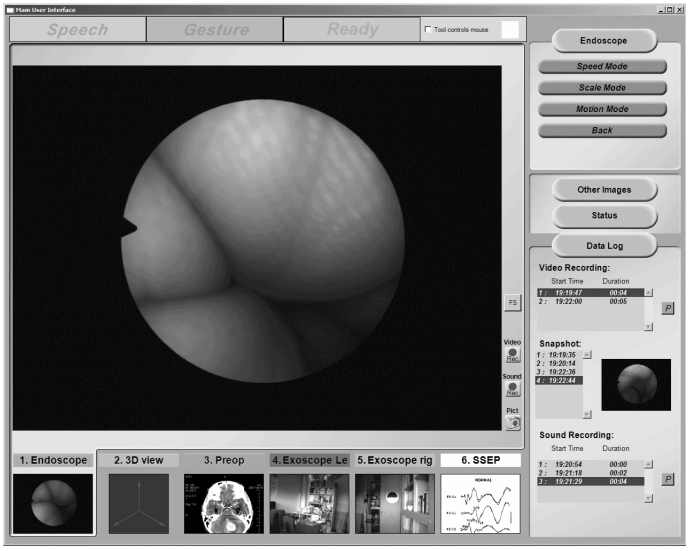


Figure 6.14: Intelligent Operating Room Console

the motion information, surgeons can either use their hand directly as a mouse or perform simple gestures to trigger actions.

Two motion capture techniques have been integrated into the system. First, a stereo camera is used to track the surgeon's hands and feed the gesture recognizer. This technique is non intrusive, easy to install but is rather sensitive to light conditions and its accuracy is limited[23]. Secondly, we used a method where markers are mounted on the surgeons tool and being tracked, using pulsed infrared light, by a base station that computes their location in space[50]. This technique is more intrusive (instruments have to be equipped with markers) but provides a better precision and is less sensitive to light conditions. Thanks to our service oriented approach, both mechanism can be easily replaced and swapped without adjusting any code nor configuration parameters.

For effective and fast gesture recognition, we extended the libstroke 2D recognition technique [83] to work as a 3D gesture recognizer (see figure 6.15). LibStroke takes a stroke (set of captured positions) and converts it into a command by generating its signature. In its classic 2D implementation, the algorithm creates a bounding box around the stroke and divides it into a 3x3 grid where each sub area is uniquely identified (1 to 9). Then, each element of the stroke is visited to find out the subarea of the matrix where it belongs. Identifications of each visited subarea are concatenated to create the signature of the stroke. The signature can then be compared to a vocabulary that binds commands to signatures. Since we are using 3D gesture capture techniques, we extended the libstroke technique to work in 3D. Instead of using a 3x3 matrix, we work with a 3x3x3 matrix consisting of 27 sub areas.

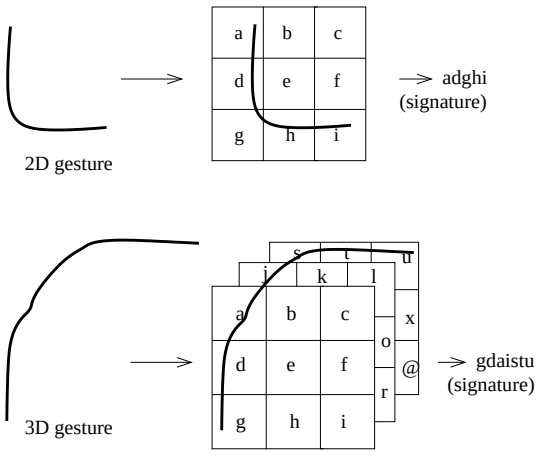


Figure 6.15: 3D Gesture Recognition

Speech and sounds

Speech recognition and speech synthesis are two services implemented in C#, using the Microsoft speech SDK. The speech recognizer has not been trained and uses a generic *male user* model. For best results, we have nevertheless constrained it to a small grammar based on the vocabulary set defined by our application domain.

Recognition triggering

A fundamental issue with both speech and gesture recognition is how to tell recognizers when to start recognizing and when to stop.

The simplest solution is to use a button, or a pedal, to trigger the process. Surgeons are generally against adding any contact device to their work environment. They already have multiple pedals to control the position of the operating table, lights, drills or suction devices.

For speech recognition, our solution is based on a keyword-based technique to let the speech recognizer know when to listen to the surgeon. This was implemented using dynamic grammars, where most of the time the speech recognizer has a very simple grammar, made of a few keywords. Whenever one of them is recognized, a more complex grammar is loaded on the fly and the user can provide a full richer command. A short pause would stop the acquisition and start the speech recognition process.

For gesture recognition, we use to concept of *hot zones*. Whenever the hand of the surgeon (or tracked tool) is positioned in a predefined zone for a few seconds, the gesture recognizer is triggered. The user is then notified (by a beep) so that he or she can start performing a gesture. To specify the end of a gesture, the user stops moving for a predefined period of time (a few seconds) to terminate position acquisition and trigger the gesture recognition process.

6.3.5 System evaluation

The system presented in this section is the second proof of concept showing that the Active platform can be used to implement end-to-end user centric applications. The evaluation of this prototype is organized into two topics. First, we measured performance aspects of the system to compare them with the initial requirements. Secondly, we studied a small population of surgeons asked to evaluate the system by performing a set of simple tasks.

Compliant with initial requirements

The actual behavior of the implemented prototype has been compared with both its functional and response time requirements.

Functional requirements The system fulfills all the functional requirements captured in the design phase of the project. For the language processing aspect of the applications, regression tests simulating user interaction have been successfully run against the application. In addition, usage scenarios have been executed by both non-medical users and surgeons to verify the usability of the assistant. A more complete evaluation from surgeons is given in the following section.

Responsiveness The actual implementation of the intelligent operating room partially complies with its responsiveness requirements. When surgeons express a request, the response time, defined by the amount of time required by the system to provide appropriate actions, ranges from 550 to 1200 milliseconds (see table 6.16).

The execution sequence of each user request can be summarized as follows. Once a user request is sensed either from the graphical user interface, the speech recognizer or the gesture recognizer, it is reported as an incoming event into the Active Server. Then, the Active Ontology in charge of language processing kicks in and produces a result within 350 milliseconds. (Details on the performance of the Active-based language processing technique is provided in section 7.3.2 on page 182). Given that the logic of this application is fairly simple and to optimize performance, a single Active Ontology performs both process modeling and service delegation. This component provides a result within 150 milliseconds. Finally, the action to perform is communicated to the appropriate components through SOAP messages. Since all components of the application are running on a single computer, invoking services is not significantly contributing to the overall response time. Table 6.16 shows the sequence of processing steps and their contributions to the overall response time.

Capture technique	Capture [ms]	Parsing [ms]	Processing and delegation [ms]	Total [ms]
Speech Recognizer	500 to 700	300	150	950 to 1200
Gesture Recognizer	1100	300	150	1450
User Interface (touch screen)	50	300	150	550

Figure 6.16: Contributions to the overall response time

The *capture time* is the amount of time required by sensors to capture and report user inputs. The capture time starts when the user has fully expressed a request to end when the recognized utterance is reported to the Active Server. Let us examine three cases : speech recognition, gesture recognition and the graphical user interface.

First, spoken commands. The speech recognizer used for this prototype uses the Microsoft Speech Engine (version 5.1) exposed as a web service using the .NET framework and written in C#. The speech engine used is grammar based, the smaller the grammar, the faster the results. To provide fast processing time, the grammar is constrained to the application domain and kept as small as possible. The system acquires the voice signal as the user speaks, to launch the recognition process as the user stops talking. In our setup, speech recognition and delivery of the utterance to the Active Server ranges from 500 to 600 milliseconds depending on the length of the utterance.

The second sensor used in the system is the gesture recognizer. The process actually involves two components: the *tracker* and the *recognizer*. The tracker is the device in charge of creating a trajectory out of the three dimensional cursor used to express a gesture (a hand or a tracked tool). Similarly to voice recognition, the capture time starts when the gesture is detected as complete. At this point, the recognizer takes over to get hold of the trajectory and launches the libstroke 3D algorithm to generate a gesture signature and deliver the result to the Active Server. For both tracking techniques, the whole process takes 1100 milliseconds. The delay is mostly due to the technique used to detect the end of the gesture. As described in section on page 147, the system waits for the user to pause for about one second to consider the gesture complete and trigger the recognition process. Experiments showed that the shortest possible pause time was one second to reliably detect the end of a gesture, therefore adding a significant contribution to the capture time. Once triggered, the recognition process and event delivery takes about 100 milliseconds.

The third sensor is the graphical user interface. Touch screens can be sterilized, or wrapped into sterilized plastic covers, and can therefore be used in the operating room. In such cases, surgeons can use their fingers to interact directly with computer programs significantly reducing the capture time.

The total processing time of table 6.16 shows that, for complex input modalities the response time of the system can take up to 1500 milliseconds. Therefore, our initial goal of responding to a user utterance within one second is not met. We note that most of the time is spent on the recognition process (voice and gesture) and can be further improved outside of the scope of the Active framework. However, about 500 milliseconds are spent in various Active-related processing.

This figure needs to be lowered to allow both the application and its processing to become more complex in the future. Section 7.3 on page 175 provides an analysis and suggestions to improve the performance of the Active system.

User evaluation

The system has been implemented and deployed as a prototype setup to be evaluated by surgeons. Given the very small population of our evaluation (three surgeons), results may not be significant. It is however interesting to report their comments and the result of their evaluation.

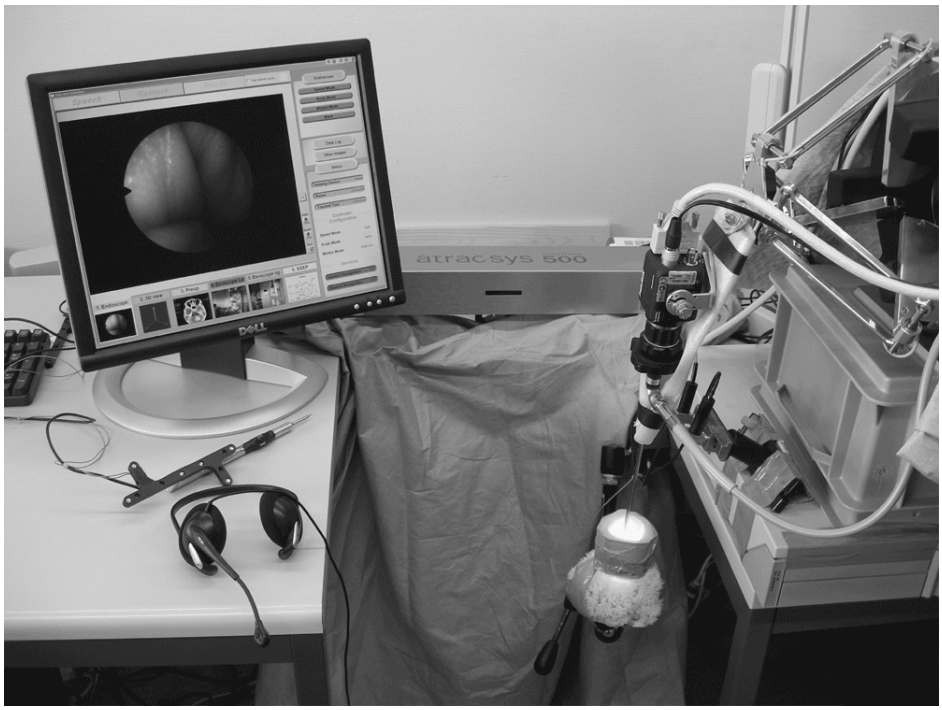


Figure 6.17: Evaluation setup

First, the evaluation took place on a slightly enhanced demonstrator built within the context of a European project. Two major enhancements were added to the system presented so far. First, a surgical instrument has been equipped with a marker to be tracked. The instrument is used to not only perform gesture recognition but also as a 3D joystick to directly control the position of a robotic endoscope holder. The robot was a thee degrees of freedom parallel system, able to move the endoscope within a work area represented by a 30 centimeter cube. Secondly, more functionality has been added to take snapshots and videos out of the video stream generated by the endoscope.

After a brief introduction and basic training on the system, surgeons were given a list of tasks to complete. As a multidisciplinary project, the exercise involved tasks to test and evaluate all the components of the project. Here is the subset of Active related tasks.

1. Image navigation. The user has to bring the third image and the last image of the list of CT scans. Tasks have to performed first using classic user interface, then voice commands and finally gesture recognition.

2. Media capture. The user has to take snapshots and short videos of the video stream. Tasks have to performed first using classic user interface, then voice commands and finally gesture recognition.

3. Robotic control. Surgeons were asked to control the robotic arm to insert the endoscope into a small hole (5 centimeter in diameter). This task is only performed by using a tracked surgery tool as a joystick. However, the motion mode can be controlled by voice commands to decide how fast the tip of the robot should move. Three motion modes were available: *fast*, *normal* and *accurate*. Fast mode is designed for fast motions, to quickly bring the robot around the area to operate. Normal mode is more accurate, the motion ratio between the robot and the joystick is one to one. A one-centimeter motion of the tool forces the robot to move by one centimeter. Finally, the accurate mode is used for fine movements. The motion ratio between the surgeon's tool and the robot is one to ten.

For the Active part of the project the outcome of the evaluation is positive and can be summarized as follows. First, all three surgeons agree with the idea of using assistant-like interfaces to control computer systems brought into the operating room. In addition of increasing usability, it also federates and hides the complexity of all the computer systems they need. Currently, each system comes with its own user interface, forcing surgeons and their staff to get familiar with several different user interfaces and also bring, for each application, a separate set of monitors, keyboard and mouse to the operating room.

Secondly, even if table 6.16 shows that our goal of responding to user requests within one second was not met, no surgeon complained about any lag nor delay in the responsiveness of the system. According to user interface research [60], a one second lag is noticeable and is the limit for users flow of thought to stay uninterrupted. Only systems that respond within 100 milliseconds are perceived as instantaneous. We therefore assume that, our system being a research prototype, surgeons had lower expectations during the evaluation and focused more on the functional side than the response time.

All three subjects expressed concerns with software systems that require heaving training and use complex user interfaces. The contextual help and suggestions brought by our prototype was well received, but only to be activated during training, or first usage of the system. Once the surgeon knows how to utilize a system, such proactive notifications should be disabled.

The combination of hand gesture (tracked by vision or active marker) and simple sounds (to notify that the system is actually tracking) turned out to be promising and was well received. Using a 3D pointer to directly control a device (such as the robotic holder), have a virtual mouse or perform gestures is natural to users. However, the reliability (vision tracking relies on light condition and occlusions), convenience (active markers require cables) and accuracy need to be improved before undertaking more involving clinical tests.

However, the usage of speech, for both input (speech recognition) and output (speech synthesis) was not well rated by users. On the speech recognition side, our log shows a recognition rate of 95% and a 500 millisecond recognition rate.

Despite these positive figures, two out of three surgeons were not complaining about recognition accuracy nor response time, but rather about how and when to trigger recognition. Given their already complex environment, they cannot use any external switch (button or pedal) to trigger the recognition process. We therefore opted for a keyword-based technique to activate the recognition of more complex commands. Once the keyword is recognized, the system listens for twenty seconds to serve one or more user voice commands. One out of three surgeons was accepting this solution arguing that he wears a microphone anyways for teaching purposes, and issuing non critical commands (such as taking a snapshot, or controlling ambient lights) would be best expressed through voice. On the output side, using speech synthesis is considered bothersome and disturbing by all subjects.

Finally, through this evaluation process, our intelligent-space approach received a positive feedback from not only surgeons, but also industrial companies building operating room equipments. As part of our contribution to the project, we were asked to present our work and introduce service oriented architectures to engineering teams in charge of operating room design. If the approach is appealing by its flexibility and potential, two major problems need to be overcome. First, any computer based system used for surgery needs to be extremely reliable to be certified by the authorities. This process may be challenging for components such as a speech recognizers, a hand tracker, a gesture recognizer and the entire Active framework. Secondly, standards need to be designed and agreed on to implement such a service oriented approach.

6.3.6 Conclusion

The prototype presented in this section is the second application built on top of the Active platform. The system is an intelligent assistant aimed at providing help to a surgeon in the operating room.

The application was designed with two goals in mind. First, it shows that the Active framework is generic and flexible enough to build user-centric applications in different application domains. As section 6.2 on page 127 presents an Active-powered system that helps mobile users dealing with online services, the prototype presented here shows a functional application designed to provide help in the medical field. Our operating room assistant complies with all its functional requirements. A batch of regression tests are used to ensure its language processing component complies with its requirements and user-based tests ensure that all functional pieces work according to scenarios. However, the responsiveness requirement of reacting within one second to any user input could not be met. Some requests require up to 1500 milliseconds to be processed and their actions undertaken. If this is a problem to be solved in the future, the user evaluation did not find it blocking and did not make the application unusable.

Secondly, the prototype claims that using an assistant-like interaction style helps surgeons accept and leverage computers in the operating room. To evaluate this claim, the prototype was tested by a small population of surgeons. Our system was positively rated and a valuable list of comments and suggestions was drawn from the test results. Surgeons agree that a more natural interaction scheme with computer would help their integration and use in the operating room. Moreover, such approach federates heterogeneous systems into

one single, easy to use interface that hides their complexity. The outcome of the evaluation also showed that a combination of gestures and simple sounds would not conflict with surgeons activities and be the most appropriate way of communicating simple commands. On the other hand, voice-based command and text synthesis are found invasive and more distracting than useful. Finally, the commercial world expressed interest in this new way of approaching computer systems in the operating, while warning about safety and standards compliance problems.

Finally, the implementation of the system allowed us to design, implement and test innovative solutions to solve some of the problems expressed the application requirements. For instance, the extension of the libstroke algorithm to implement fast 3D gesture recognition had never been done before and proved to be effective for our application. We also experimented with a range of techniques to trigger the process of both speech and gesture recognition.

6.4 Scheduling Assistant

6.4.1 Introduction

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

The section is constructed as follows. First, the goals and reasons behind the development of the system are presented. Next, a scenario presents the requirements of the system. This is followed by a detailed implementation description of the application. After an evaluation of the system, both in terms of functionality and implementation, a brief conclusion summarizes the achievements of this project.

6.4.2 Prototype goals

This application was chosen for three reasons. First, the prototype is another testbed to validate and improve our Active-based approach to build user-centric applications, one that pushes our use of long-running processes as opposed to the request-response style interaction that characterizes our previous prototypes.

Secondly, as meeting organization has been the subject of multiple agent-based implementations[56, 62, 57, 4], it has become a standard test for intelligent applications. It is interesting and relevant to compare our approach with other implementations of the same task.

Finally, based on our experience, we will try to build the whole system in less than a week, to prove that the Active platform is suited for quick application building and prototyping.

6.4.3 Requirements definition

The functional requirements are expressed through the following scenario. The organizer uses an instant messenger client (Yahoo! Instant Messenger^{TM} in our example) to interact with the intelligent assistant. 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 typing *"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.

This scenario is the basis of a regression test harness designed to validate the Active Ontology in charge of language processing. In addition, the scenario is used to model the logic of our application. Although this example proposes a fairly simplistic modeling of the scheduling process, it is sufficient to illustrate natural language interpretation and dialog, running asynchronous processes, web-service delegation, and multimodal user interaction.

As for responsiveness, we apply the rule suggested by Chai[87], where feedback should be given to the user at most ten seconds after the last request was expressed. For operations that take more than thirty seconds to complete, which is the case of the process of organizing the meeting, acknowledge about what was understood and what is being undertaken should be provided within ten seconds.

6.4.4 Implementation

Overall design

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 assistants email account or its instant messenger client

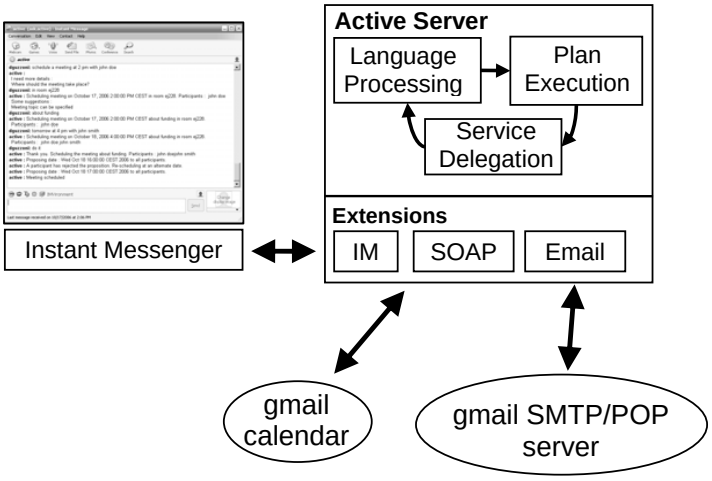


Figure 6.18: Meeting Assistant Architecture

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 6.19) using the process technique defined in section 5.4 on page 109. Following a top down execution, the start node *Start-ScheduleMeeting* triggers the execution of the process. First, the meeting organizer is notified through the execution of the *NotifyOrganizer* node that contains Javascript code to format user messages. Then, the *Get-PossibleDate* 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 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 inter-

actions with these external resources are ran through the delegation mechanism described in section 5.3 on page 96. 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 users preferences, its location or current availability of providers.

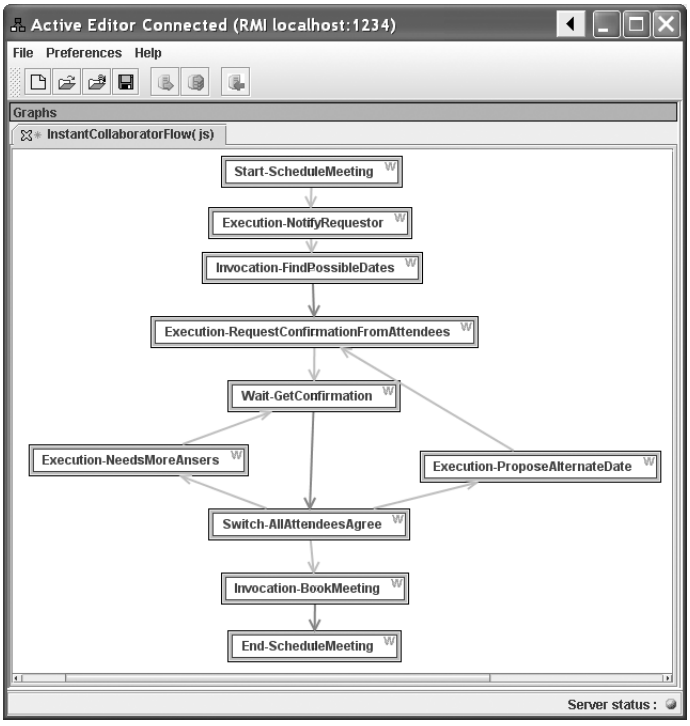


Figure 6.19: Scheduling logic in Active Editor

External services

Following the Active design pattern, our application is made out of a set of Active Ontologies and a community of loosely coupled services (see figure 6.18). In our case, the backend is based on the Google calendar APIs to read and modify the personal calendars of meeting attendees who are supposed to have a Google Calendar^{TM} account. Google public APIs 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 6.19) waits for incoming email messages asserted as Active facts.

6.4.5 System evaluation

The system presented in this section has been evaluated along two lines. First, functional and responsiveness tests were conducted to ensure that the implemented system actually complies with its design requirements. Secondly, it has been compared, in terms of implementation and featureset, with similar research projects.

Compliance with initial requirements

The implemented application fulfills its functional requirements. Regression tests have been run with success against the Active Ontology in charge of language processing. The scenario defined as the main requirement for the application can be successfully executed by test users.

The response time requirements are also met by the implementation. During the initial dialog that defines all the details of the meeting to organize, the assistant responds with two seconds to any user request. As the scheduling process starts, each long lasting action is preceded by a message explaining what is about to be undertaken. (see figure 6.20)

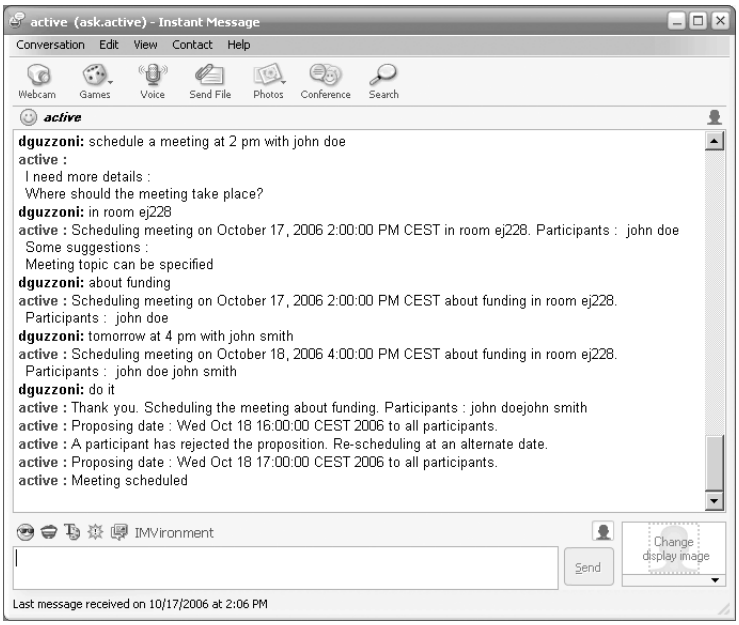


Figure 6.20: Meeting Assistant Interaction

Finally our goal was to implement the whole system is less than a week was achieved. The prototype was designed, implemented, tested and deployed within three height-hour workdays. It demonstrates that trained programmers can build simple end-to-end assistant applications within days. The modular and yet unified Active-based design allows for rapid prototyping of simple applications, to further analyze and improve specific components. For instance, the current scheduling logic is very simple and nicely encapsulated in a specific

Active Ontology. Improving the process would only impact this part of the application, as remaining components would be untouched.

Comparison with existing systems

The domain of meeting scheduling has been a popular application for intelligent assistant applications. This prototype was designed to prove our claim that programmers can more easily and quickly create assistant-like applications using the Active system. We have compared our implementation with two systems designed to help organize meetings: *CMRadar* and *PTime*.

CMRadar[57] is a specialized agent designed for scheduling meetings. It part of the larger RADAR project, whose main functionality is to help a user deal with the "crisis" of coordinating a major conference. Everything predominantly starts with trying to deal with a full inbox of email that needs to be sorted through and acted on. Emails will be of different types (e.g. update a website, reschedule some event, find a place for some event), so as a first step RADAR will classify emails into a hierarchy of possible template types; for instance, a message that says *"I'm a keynote speaker currently on the schedule for Tues, but I won't be arriving until Wednesday"* can be classified as a "Schedule Effect Task", or in a more detailed way down the hierarchy, a "Schedule Modification Task" or even a "Schedule Time Modification Task". Once classification of the task type for an email has happened, information extraction will try to pull out the relevant slots from the message. This will produce a formally structured task item that will show up in the user's task list. As the user validates and "executes" the task, this will add constraints to the space/time scheduling component. The RhaiCAL component of Radar is a calendar-style GUI that displays the current schedule, and a list of violated constraints that grows as you add new requirements. Once you have collected lots of the constraints, hitting "optimize schedule" will ask the scheduling piece to try to find the best solution to the posed problem.

Note, this is quite a different problem than the PTIME (and Active) problems, which are trying to meet the need of getting a schedule based on group consensus (where each participant has their own, perhaps hidden, preferences).

The PTIME[4] (Personalized TIme ManagEr) system is an agent the helps users manage their temporal commitments in an adaptive, mixed-initiative and collaborative fashion. Each user is assigned a PTIME agent in charge of managing his or her calendar. The agent is capable of negotiating on behalf of the user, provide relevant information and learn about preferences. The system integrates with commercial calendars and integrates constraint satisfaction scheduling, goal directed process management and preference learning. Finally, part of the larger CALO project, PTIME leverages common components such as the IRIS[11] extensible user interface application.

	CMRadar	PTime	Active-based
Features set			
Language processing	**Yes (Email only)**	iAnywhere^{TM}	**Yes**
Scheduling	Negotiation and constraint-based	Negotiation and constraint-based	**Naive (extensible)**
User Interface	**Email or RhaiCAL**	Towel Task Manager	**email, instant messenger, thick clients**
Implementation			
Programming languages and tools	**Ozone scheduling, pattern based parser written in Perl for NL**	Process control (spark), Constraint reasoner (Prolog), Preference Learning (Java, SVMLight, C)	**Active Ontologies, Javascript**
Integration	Shared memory, proprietary communication protocol	OAA	**Plugins, extensions, SOAP**
Domain Specific	**Yes**	**Yes**	**No**

Figure 6.21: User Evaluation Results Summary

6.4.6 Conclusion

This sections presents an assistant specialized in helping organize meetings. The system, the third Active-based prototype, interacts with a combination of instant messages and emails.

The project fulfills three goals. First, since the application complies with all its initial requirements, it is the third application that validates our unified approach in designing user-centric applications. Secondly, since multiple systems implement assistants helping in organizing meetings, it is an excellent candidate to compare our approach with other techniques. The comparative analysis shows that our approach allows a programmer to quickly create simple, but complete prototypes that can be easily refined and improved in subsequent phases. Other implementations provide one single excellent specialized component (planner, user interface or dialog) while other elements of the overall

application are more limited.

Chapter 7

System Evaluation

7.1 User evaluation

The goal of the user evaluation is to compare and contrast our approach against what is currently available to mobile users. This comparative study helps us validate our claim stating that in the domain of online information retrieval, an intelligent assistant-like approach can be more effective than more conventional tools.

7.1.1 Evaluation protocol

A limited population of 20 users has been asked to participate in our study. Each user has been asked to perform a list of tasks with both Google Mobile™and our prototype nicknamed Active Mobile. For practical reasons, the test duration is limited to one hour and divided into four segments: *introduction* (2 minutes), *pre-evaluation questionnaire* (5 minutes), *tasks to perform* with both systems (twenty-five minutes each) and a *post-evaluation questionnaire* (2 minutes). All test documents (questionnaires and response sheets) are provided in annex B.

- **Introduction** In this segment, an introduction to the experiment is given to the subjects. First, we explain that the test is anonymous and only the systems are being tested, not the users. The goal is to relieve any stress or pressure users may feel.
  Secondly, whereas many users will know how to use a Google-based search engine, three points are provided to introduce them to the features of an assistant-like system. First, we emphasize that in addition to keywords to express their intent, full rich sentences can also be submitted. Secondly, an assistant will try to anticipate user intentions and provide relevant contextual suggestions. Finally, an assistant maintains a context as the dialog unfolds, and is therefore aware of some details (such as the current location or time) that do not need to be repeated.

- **Pre-evaluation** To establish a profile about subjects, a questionnaire collects information about how experienced users are with traditional search engines.

- **Scenario** A ten-task scenario related to the domain of travel is given to users. They are asked to perform various tasks including fetching information about a specific flight's status, finding a five-day weather forecast, searching for hotels, restaurants, entertainment and services about a city where they have to travel (see appendix for details). A response sheet with blanks has to be filled out within twenty-five minutes. Half of the users were asked to start the quiz with Active Mobile, before performing the same tasks with Google Mobile^{TM}. The other half was asked perform the test in the opposite order.

- **Post-evaluation** Finally, a post-evaluation questionnaire is given to all participants to gather information about how they evaluated the systems and how the systems could be improved.

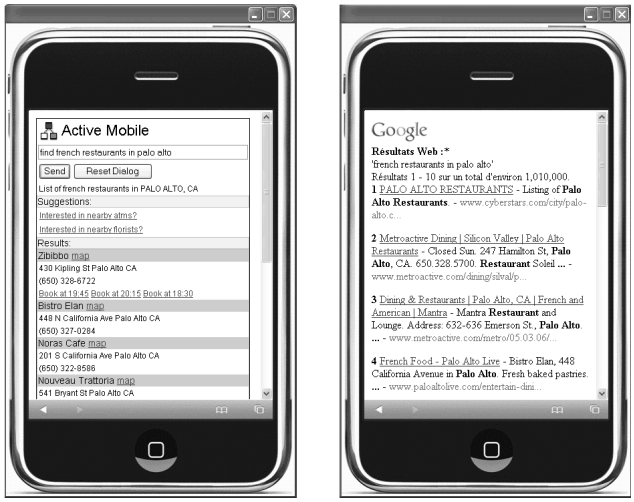


Figure 7.1: Screen shots of the *Active Mobile* and *Google Mobile* systems.

Users were asked to performed the required tasks using a cellular phone emulator running on a desktop computer. The emulator (see figure 7.1) is used to provide realistic screen size and form factor conditions. In addition, the emulator has been instrumented to time and log all requests and responses as users perform the required tasks. Collected logs were used to extract the following measurements:

- For each task:
  - Time to completion. Duration, expressed in minutes, from the initial query until the request information is retrieved.
  - Number of user request. How many consecutive requests were necessary to complete the task.
  - Completion factor. The information to get can be fully or partially retrieved. This measurement is expressed as a percentage (100% for fully retrieved information)

- At the test level (for all tasks):
  - Total time. Time, expressed in minutes, to complete the ten-task scenario. If after twenty five minutes the scenario is not completed, the test is aborted.
  - Completion factor. The total completion factor.
  - Total number of user requests.

7.1.2 Results

This section provides the results gathered from subjects. The results of our evaluation have been grouped into three categories: pre-evaluation questionnaire, task results and post-evaluation questionnaire.

The pre-evaluation questionnaire shed an interesting light on our population. If most of users (95%) use search engines very frequently, only 15% have used them from their mobile phones. When asked why, users mentioned multiple causes including Internet access fees (this may be specific to Switzerland), communication latencies, and mostly frustration due to mobile user interfaces being difficult and slow to use. A very large number of users use Google (95%) from their desktops as their primary search engine and a minority of them (20%) use advanced search features such as operators, quotes, formats or sites constraints.

On the core test, subjects performed significantly better with Active Mobile than Google Mobile^{TM}. The main result of our evaluation consists of analyzing logs generated by users attempting to perform our ten-task scenario. Within the twenty-five minute period allocated to complete the tasks, Active Mobile allowed users to complete 90% of the work, whereas Google Mobile^{TM}allowed for the completion of 54% of the tasks. In addition, Active users needed an average of 13.6 queries to solve the tasks, whereas Google users required an average of 48.6 requests. On the time to completion, most Active users (90%) completed all tasks within the time allocated for the test, whereas only 70% of Google Mobile^{TM}users managed to complete the test. The following discussion elaborates more on the results and attempts to explain them.

The post evaluation questionnaire gave a clear preference to the intelligent assistant approach over the more conventional keyword-based technique. A large portion of subjects (95%) thought the Active-based system performed better than its Google Mobile counter part.

7.1.3 Discussion

The results presented above show that, for a specific domain (travel related queries issued by mobile users in our example), an intelligent assistant-like approach performs better than a generic keyword-based system. It is nevertheless important to provide a more detailed comparative analysis.

Behaviors

As subjects were testing the system, we realized that user behaviors and search techniques had a significant impact on the performances of the keyword-based system. However, most user behaviors led to similar performance with the Active-based assistant.

First, let us describe the *expert behavior*. A small set of users used the Google Mobile™ system in a way that made it almost comparable to the Active-based system in terms of task completion. They nevertheless required about three times as much time and many more queries.
Instead of looking for the information itself, advanced users looked for specific web sites or portals that would be best suited for the task to accomplish. For instance, looking for an actual flight status proved nearly impossible with Google Mobile™ using a keyword-based approach based on airline name, flight number or airports codes. However, some subjects (10% of the population) used Google to navigate towards specialized sites. Instead of using keywords like *"flight united 501 status"* they typed *"flight tracking"*. Through this approach, Google provided links to dedicated sites where specific flight information could be provided and the correct information fetched.
A second class of users has been categorized as *verbose* users. They tend to pack as much information as possible into each query, to make sure the system has enough information. They are trying to maximize their chances to get the correct data while avoiding multiple back and forth utterances to save time and avoid delays. Thanks to its robust parser, the intelligent assistant copes well with long queries and gracefully ignores unnecessary words. On the other hand, too many keywords confuse a Google-like system, which has very little semantic knowledge about incoming tokens.
Finally, we noticed users with a *consistent* behavior. This group used the same keyword-based approach for both a plain search engine and the intelligent assistant.

Context

The ability of the assistant-like application to build a context over the dialog was well received and used. It has been leveraged by 80% of users, who did not repeat location information after getting the first set of data about the city to visit.
It is interesting to note that search engines like Google Mobile™ offer an explicit and limited context management. When starting a session, users are asked how to search, either over the entire Internet or choose to look for localized businesses and services. The vast majority of our population (90%) was not aware of this option and chose to search over the entire Internet. The localized search screen provides two input fields, one for search keywords and a second to specify a location (city name or zipcode). After submitting a request, for instance asking for restaurants, the user is provided with a summary of what was understood and a list of structured matches. In this mode, Google Mobile™ behaves like our Active-based assistant, it returns aggregated and formatted information instead of links to potentially relevant pages. Search constraints being purely based on keywords without semantic understanding, additional information to get the *best*, *most popular* or *cheapest* are either not taken into account or confusing to the system. The bottom of the result page provides two input fields for further searches. Both fields are pre-populated with the content of the previous request, thus building a simple, explicit and useful context management.
Switching back and forth between the two search modes (broad Internet or localized information) is not easy and prevents users from easily look for non localized data (such as a flight status) and local details (such a s hotel), without

starting a new session. The semantic capabilities of an Active-based assistant approach hides this complexity from the user, who can express request about local business or generic information without having to pick which search mode to use.

Search follow up

We have also studied how users express follow up requests after an initial query that did not return the expected information.
When using the search engine-based system, users modify the previous request by adding, replacing or re-ordering keywords. According to the logs gathered, adding is not a good technique, as it tends to confuse the search engine that tries to match as many keywords as possible. Re-ordering of keywords had no impact on the search results. Replacing keywords with new ones allowed users, over fine tuning, to get to the correct information.
On the case of the assistant-like system, users took advantage of the dialog approach by adding one or two words to fine tune their requests and get the appropriate information. For instance in our test scenario, after getting flight information, users were asked to get the five-day weather forecast for the destination city, San Diego. A user typed *"weather at destination"*, which failed because the test prototype does not use a data validation agent that could resolve *destination* to *San Diego*. Instead, the system returned the weather forecast using the current context, set to Palo Alto, California. Noticing it, the user simply typed *"San Diego"* to get the correct information and adjust to dialog context for further queries.

Utterance size

Both systems led to the same average of about four words per utterance.
First, let us discuss these results for Google Mobile™. A large scale research on mobile and desktop Google users search habits[42] shows a average of 2.3 words per query, which is smaller that what we observed. Since our test queries are task oriented asking our subjects to accomplish something, they may be slightly more expressive that if they are looking for online information.
On the Active Mobile side, this rather small number of words per utterance can be explained by two factors. First, users were told that keywords as well as natural language utterances could be used. Using a cellphone emulator for the evaluation, subjects naturally tried to minimize typing and started using the system as they would use any search engine. Note that other modalities, such as a speech recognizer, may lead to longer and fuller utterances. As the dialog unfolded, Active users became aware of context management and did not have to repeat locations or attributes when looking for new pieces of information. For instance, in our test scenario after looking for hotels in San Diego, users where asked to find the best Indian restaurant in town. Since the context already contains San Diego as the location, the three-word utterance *"best indian restaurant"* is enough whereas *"get me the best indian restaurant in town"* would have worked as well.

Information presentation

Information and rendering proved to be a crucial component to effectively get the requested information.
Using Google Mobile™, links resulting from search queries lead to sites that provide relevant information. However, many sites are not suited to mobile devices, and the information cannot be easily found because it is hidden in a cluttered and overloaded HTML page. In our test, when looking for flight status with Google Mobile™, several users ended up on a page that contained the correct information but did not see it.
The assistant-like application provides aggregated and formatted information instead of links to potential information sources. It is both an disadvantage and an plus. The downside is that, in some cases, the system is not able to provide information that is not part of its formatted view of the application domain. This is the reason why, future versions of our system will provide, among the clean and formatted information, direct links to the information source. For instance, for each hotel information, in addition to hotel details and ranking, the assistant should provide a link to hotel site, where users can navigate and try (given that the site is compatible with a portable device) to get more detailed information. On the other hand, there are many advantages in providing formatted information instead of raw links. First, multiple heterogeneous sources can be aggregated and rendered in a consistent manner, thus hiding the complexity of underlying data representation from the user. Items can be sorted and filtered more easily. For instance users looking for Italian restaurants only get a subset of all possible restaurants. When looking for the best place to eat, entries can be sorted by rating. In addition, subjects suggested that providing structured information cleanly laid out helps building trust between the assistant and its users.

Application domain

In some cases, the broad generic search capabilities of the keyword-based approach is an advantage. An assistant-like application uses *à priori* knowledge about a specific domain to deliver the best information. For instance, the application domain and vocabulary are encoded into the assistant and are effectively working as long as users ask questions related to the assistant's domain of expertise.
For instance, in our evaluation, our Active-based system would not have been able to provide the recipe of a porcini risotto. A shallow, but massively broad system like Google Mobile™has the potential of providing such information to the user. There is a trade off between the quality of information and the domain covered by an application. The results of the present evaluation show that a broad system like Google Mobile™is outperformed by an intelligent assistant specialized in a specific domain.

Most common errors

The Active-based application allowed for 90% of required tasks to be completed. It is interesting to understand why 10% of the tasks could not be completed and how the system could be improved to reduce this figure. Through the analysis

of user logs, we identified two main reasons that prevented the system from delivering the requested information.

Vocabulary Most errors occurred because of missing vocabulary and misspellings. For instance, hotel amenities and points of interests where expressed with words that were not part of the vocabulary set initially defined. An improvement would consist of enhancing our semantic network approach with a learning component. The application could have a special *learning mode,* where users are asked to classify unknown words based on the current context and the semantic network. Once the system has been trained, or the misunderstanding rate is low enough, the learning mode could be turned off.

Semantic validation Another source of errors comes from the assumption from the user that the assistant is fully aware of the context. For instance, after getting flights details, a user typed *"weather at destination",* which failed. To resolve these types of errors, more semantic validation could be added for a better leverage of the context.

7.1.4 Conclusion

This section presented a user evaluation of our prototype. A limited population of twenty users was asked to perform a ten-task travel related scenario. Each user evaluation lasted for one hour, with fifty minutes for the test itself and ten minutes for explanation and filling out of pre and post evaluation questionnaires. The evaluation test consisted of performing the same tasks with both an Active-based assistant (Active Mobile) and the leading commercial system designed to help mobile users (Google Mobile^{TM}).

	Active Mobile	Google Mobile
Tasks completion	**90%**	**54%**
Nb requests	**13.6**	**48.6**
Nb words per request	**4.1**	**3.9**
Time spent per task	**1.3 minutes**	**4.33 minutes**

Figure 7.2: User Evaluation Results Summary

Overall, the Active-based system performed significantly better (see figure 7.2). The assistant-like system provided a better completion rate (90% versus 54%), smaller number of requests to complete the ten-tasks (13.6 versus 48.6) and faster task completion (1.3 minutes versus 4.33 minutes). Interestingly, both techniques led to a similar number of about four words per request. Studies have shown that the average number of keywords used for Google queries is 2.3. We explain this difference by the task-oriented nature of our test scenario, which may have forced users to be more expressive. On the assistant side, users leveraged the dialog-style interaction by providing a small number of words at each utterance to adjust and fine tune the dialog context.

When used by expert users, Google Mobile^{TM} leads to similar completion rates as the Active-based system, but require more time, requests and man-

ual information extraction. The intelligent assistant has shown similar results independently of users skills and expertise.

In conclusion, for specific task-oriented domains (travel in our case), an assistant-like system can better help mobile users than keyword-based conventional search engines. However, for unconstrained domain searches, a Google-like system allows mobile users to retrieve virtually anything but requires a significant effort in sorting, rating and extracting information from the returned links.

7.2 Programmer Evaluation

This section presents the results of a programmer evaluation of the Active system. In sections 6.3 and 7.1 on page 161 we have shown that, for specific domains, assistant-like systems can perform better than conventional applications for end users. The present section validates our claim that using the Active approach can greatly accelerate the training and development time required to build AI-based, assistant-like applications.

In this evaluation, a population of programmers is given a simple two-step test protocol. First, a tutorial (see appendix C) is used to train the subjects on the Active system, more specifically about how Active can be used top create a language processing application. Once the training phase is completed, candidates are asked to create a language processing application able to process a small *corpus* of ten utterances. Each created application is then tested and evaluated against the ten-utterance corpus given as specification, but also against a one hundred-utterance corpus collected from real end-users usage of an intelligent assistant application.

The chapter is organized as follows. First, each step of the evaluation protocol is presented in details. Then, a section presents the evaluation results. It explains how Active programs are scored against the test utterances, presents and discusses the actual results. Finally, a discussion and a conclusion summarize our findings.

7.2.1 Test protocol

Each developer was given a three-step protocol consisting of a *training* session, *a building* phase and a *post-evaluation questionnaire*.

Training tutorial

After getting a short presentation on the Active system, we asked participants to go through a tutorial about how to create a language processing application with Active. In a step-by-step process, the tutorial shows how to create, deploy and test a language processing application based on the Active-based *semantic networks* technique (see section 5.2.3 on page 71). The tutorial introduces the most important features of the technique, starting from simple concepts, moving towards more complex constructs. At the end of the tutorial, a simple language processing application capable of parsing requests about classified ads (cars and homes) has been built and tested by the programmer.

System building

Once subjects have completed the training tutorial, the are considered as *experts* and given the specifications of a language processing application to be built from the ground up. The system to create is defined as a *specification corpus*, shown in figure 7.3, whose utterance set has to be processed.

get me the weather forecast in San Diego
whats the weather in Seattle
find restaurants in menlo park
get me the best indian restaurant in Seattle
status flight air france 83
is flight united 510 on time?
united 1233
find thrillers in San Diego
get movies with John Wayne
hotels with a pool and wifi
I want a hotel in san diego with internet connectivity and a fitness center
get me all starbucks in Paris
find florists in San Diego
nearby ATMs in Palo Alto

Figure 7.3: Specification corpus

Most features of the semantic network technique need to be used to effectively create an application able to parse the specification corpus. For leaf nodes, some may be implemented using vocabulary set, for instance to enumerate hotel amenities; some may use regular expressions, e.g. to detect flight numbers; finally, a set of prefixes may be used to detect cities in utterance such as "in Paris" or "in Palo Alto". To create an effective semantic network, programmers will also need to use *helper, gather* and *select* nodes.

Questionnaire

At the end of the exercise, a questionnaire (see appendix D) is given to all participants. First, it asks about their background and programming skills. The goal is to establish a rough profile to assess whether being an accomplished programmer or whether being familiar with language processing techniques influences the quality and nature of the generated Active programs. Secondly, subjects are asked about their overall impression, comments and suggestions about the Active system.

7.2.2 Evaluation

This section explains how the Active-based language processing applications have been evaluated and scored.

Test corpora

Once complete, each application is tested against two corpora: the *specification* corpus and the *end-user* corpus.

First, the system is tested against the ten-utterance *specification corpus*, which was used by the evaluation subjects to create their applications. Since it was the basic specification of the test, the evaluation score against this corpus is expected to be very high.

In order to have a realistic corpus to test against, we need another, more realistic, set of utterances based on actual end-user interactions. Since the system to be created by our subjects is very similar to the Active Mobile application, we could create a substantial *end-user* corpus out of its end-user evaluation . So that our end-user evaluation (see section 7.1 on page 161) will be used to create the *end-user* corpus, that feeds our programmers evaluation.

Using the two corpora, we can evaluate the effectiveness of Active programs created against their specifications, and also how they would perform against a realistic set of utterances.

Scoring

After defining the two test corpora, we need the define how to score and measure the effectiveness of the language processing applications created by our subjects. The effectiveness of the created parsing applications is measured by the amount of relevant information extracted from user utterances. For each utterance of the corpus, the language processing is scored. The overall score of the system is defined as the average of each individual utterance score.

Basic For instance, the utterance *"find movies in palo alto"* contains two pieces of information. A location (*palo alto*) and a subject (movies). A parser able to pick both pieces of information gets the maximum score of *100* (one hundred percent of the information has been extracted), a parser the only gets the subject and misses the location would get a score of *50*. For the sentence *"find a hotel with a pool in Paris"*, parser that misses the amenity (*pool*), but is able to extract the location (*Paris*) and the subject (hotel), get a score of *66*.

Context The Active-based language processing technique naturally supports context management (see section 5.2.3 on page 81). Therefore, to fully evaluate our system, two separate scores are created for each utterance: the *instant score* and the *context score*. Instance score only considers the information contained in the current utterance, measures how much of it was extracted by the parser to create the score. The context score, not only takes into account the amount of information provided by the current utterance, but also what is expected given the current context. For instance, let us consider the following sequence: *"find hotels in palo alto"*, followed by *"on with a pool"*. The first utterance provides a location and a subject, whereas the second utterance contains a single piece of information, an amenity. When computing the *context score*, we expect the parser to provide three pieces of information after the second utterance (location, subject and amenity type), whereas the *instant score* considers

that where is a single piece of information to extract for the second utterance. A parser that successfully extracts the location and subject from the first utterance, but misses the amenity specified by the second utterance would get a *0 instance score* and a *66 context score*.

Lists Finally, the scoring technique take errors and suggestions lists into account. The semantic network technique used in this evaluation has the ability to provide *errors* and *suggestions* list along with the parsing results. If a parser misses a piece of information, but mentions in its error list that it is missing, it gets half the penalty. For instance, if a parser misses the location from the utterance *"find movies in palo alto"*, but specifies in its error list that the location is missing, it get a score of *75*. If the parser is used in an end-to-end application, the user would be notified that a location is missing and be able to correct the error by expressing more accurate utterances.

7.2.3 Results

Population

We tested a small population of ten subjects. Most of them being software engineers (70%), some very creative (see figure 7.4), others more compact (see figure 7.6). Although not programmers,the rest of our population are professionals active in technology companies. Only 20% of our population had experience with AI and were familiar with the concept of language processing.

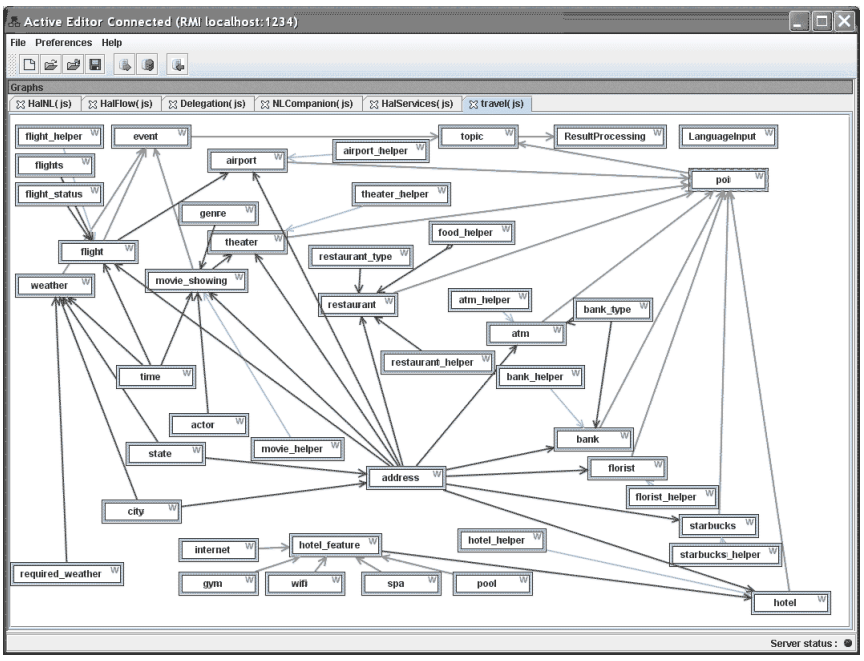


Figure 7.4: Sample Active Onotology from the programmer evaluation.

Timing

All users completed the training part within 90 minutes; some (50%) completed the training tutorial in less than 45 minutes. The second phase of the evaluation, the creation of an language processing application, was also completed in about on hour. Programming time ranges from 50 minutes for the fastest subject, to 1 hour and 20 minutes for the slowest. Most participants (70%) managed to complete the second phase between 55 minutes and 1 hour and 10 minutes.

Effectiveness

Using both *specification* and *end-user* corpora, we evaluated the Active Ontologies created by the participants using both *instant* and *context* scores.

Participant	Instant score	Context score
1	86.9	88.6
2	98.2	98.2
3	98.3	98.2
4	91.1	91.1
5	98.2	98.2
6	91.1	91.1
7	94.6	95.8
Average	**94.1**	**94.5**

Figure 7.5: Scores on the *specification* corpus

Table 7.5 reports scores related to the *specification* corpus for each participants. The numbers show that, after one hour of self-training using a tutorial, most users were able to create a language processing application that fulfills more than 94% of its requirements. We also notice that, since sentences of the specification corpus do not heavily rely on context, *instant* and *context* scores are rather close.

Participant	Instant score	Context score
1	75.9	75.4
2	77.8	79.5
3	78.3	78.6
4	79.9	82.5
5	76.0	76.6
6	76.0	76.4
7	82.2	85.3
Average	**78.0**	**79.2**

Figure 7.7: Scores on the *end-user* corpus

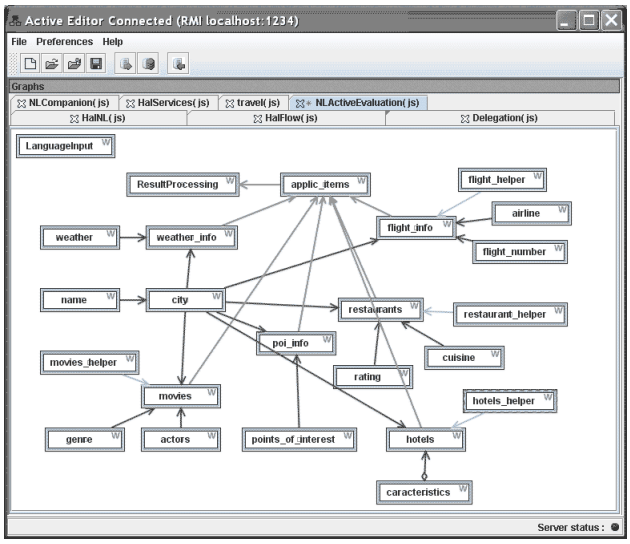


Figure 7.6: Sample Active Onotology from the programmer evaluation.

Table 7.7 reports scores related to the *end-user* corpus for each participants. The scores show that, our population of participants was able, in about two hours, to get trained on both Active and on the concept of language processing. Within this time, subjects successfully created an application capable of extracting nearly 80 percent of the information carried by a corpus, consisting of one hundred utterances, created from actual end-user queries.

These results help us demonstrate our goal that consists of using the Active framework to lower the bar and ease the creation of intelligent assistant applications.

Discussion

Our sample population is rather small – a larger number of subjects would be useful to draw and validate our conclusions. However, some aspects of the results presented in the previous section can be interpreted and discussed.

Uniform results

First, we note that user results, on both types of scores and corpora, are rather uniform. It tends to demonstrate that being a software programmer may not have a significant impact on the effectiveness of the end application. This trend can be explained by the nature of the language processing used by the evaluation subjects. The semantic network technique focuses on abstract domain modeling, where all processing is automatically added by Active Editor wizards. Therefore, Active programmers need only to model the application domain, the structure of the language, with no coding is required.

Active Editor

Undertaking this programmer evaluation showed that a powerful IDE is a crucial component to our system. Many features of the Active Editor were put to the test, showed their limitation, even requiring some improvement and work after initial pre-evaluations from developers. Programmers are used to the power and flexibility of platforms such as Eclipse or Visual Studio and are expecting the same level of maturity from any programming tool. This exercise helped us gather a list of improvements and suggestions. It also confirmed our future plans to port the entire Active Editor to run as a set of Eclipse extensions.

Common problems

While scoring the Active Ontologies created by our test population, we verified for each utterance, why some part of the information carried by incoming utterances were not properly extracted.

Most errors came from leaf nodes semantic rating techniques, not the overall structure of the semantic network. All participants were able to model the upper part of the network without any problems. On leaf nodes, the most common errors can be summarized as follows.

Prefix. The prefixed technique, which consists of considering word following a known prefix to create semantic ratings, is simple and elegant to implement, but poses problems in real world scenarios. In our study, all users but one used the prefix *in* to detect cities. This technique worked in most cases, but many utterances of the *end-user* corpus simply use city names, without prefix. One user, subject #7, used an enumeration of all cities used in the *specification* corpus, thus getting better overall scores.

Vocabularies Another common problem consists of the technique used to check vocabulary members. Our Active-based technique offers multiple ways of testing utterance words with a vocabulary set. By default the technique is the strictest, which is case sensitive, whereas other techniques can be cases insensitive or even fuzzy using the Levenhstein distance (see paragraph 5.2.3 on page 74). Most programmers left the default strict option, thus not detecting relevant words. The same problem happens with plurals. For instance, to detect *comedies*, participants would only enter the singular version of the word as *comedy*. On the other hand, end-users would express a request as *"find comedies"*, which would prevent our node from reacting properly.

7.2.4 Conclusion

This section presents the results of the programmers evaluation of the Active system.

A small population of users has been been given a basic training on the Active system and then asked to create, from scratch, and language processing application. The training lasted for about one hour, and consisted of going over all steps of a tutorial explaining how to create a language processing application with Active. Once trained, subjects were given the specifications of an application domain, as a small set of utterances to parse. It took participants about one hour to create their applications.

The application were then evaluated and scored against two corpora. First, the small corpus given as the specification of the domain to test. Then, their

applications were tested against a larger, one hundred-utterance corpus made out of real user utterances for the application domain. Results show that programs created were able to extract 94% of the information contained in the specification corpus, and nearly 80% of the real actual end-users corpus.

The results help us validate our claim, that a unified tool and associated methodology for creating intelligent assistant application eases the design and implementation of such systems. It lowers the bar to create AI-based systems, by allowing programmers to encapsulate and reuse techniques such as natural language processing.

7.3 Performance Evaluation

We have shown so far that the Active framework allows for the design and implementation of end-to-end user-centric systems. Even if the Active system was designed as a research prototype, it is relevant to measure, analyze and attempt to characterize its performance behavior. This analysis is useful for two reasons. First, we will try to assess how Active-based applications would scale up and therefore know the limitation and operating range of our existing system. Secondly, we will pinpoint bottlenecks and isolate important features to be taken into consideration when designing and Active-like system as product able to scale up and manage large applications.

The section consists of the following parts. First, we provide basic metrics that define performance measurement of the core Active Server. Then, we analyze how the Active-based language processing technique performs. Finally, a conclusion summarizes the result and provides a set of technical recommendations to improve the overall system performance.

7.3.1 The Active Server

The Active Server is the processing core of the Active system. Its performance behavior should ideally approach *linear scalability,* where the system response time grows linearly as its workload augments. Linear scalability, also called *perfect scalability,* ensures that a system provides a constant response time by duplicating processing resources as its workload grows. In the industrial real-world linear scalability is difficult to reach; systems are designed not to scale indefinitely, but to reach near-linear scalability within their operating range.

The Active Server is built around a production rule engine, whose performances mostly depend on how rules are processed. Rule processing can be seen as a two-step process: *rule condition evaluation* and *rule action execution.* The time required by an Active evaluation cycle will directly depends on how many rules are to be evaluated, and among these, how many rules conditions will be validated and lead to the actual execution of the rule action.

This section provides performance trends as well as absolute time measurements. The machine used to characterize our system is a desktop computer, equipped with an Intel Pentium D840 (3.2GHz) processor , 1GB RAM and running Windows XP (SP2). On the software side, all components where running under Java 1.5 (*build 1.5.0_05-b05*) and the database system used is MySql (*ver 14.12*).

Rule evaluation

There are multiple rule evaluation techniques, ranging from the most naive that consists of fully evaluating all rules at each evaluation cycle, to the highly efficient Rete[19, 14] algorithm that keeps intermediate results in memory and only performs partial evaluations for a new cycle. The Active Server implements an intermediate solution, where rule conditions are partially evaluated based on types and relations between the tests to perform against the fact store. The Active Server implements four types of optimizations.

Fact store optimizations The first optimization set focuses on the Active fact store. As introduced in section 4.2.7 on page 47, rule conditions consists of boolean expressions that combine *store-checks*. Since store-checks work by running a unification against the content of the fact store, the overall performance of the evaluation directly depends on two steps: the performance of the read API exposed by the fact store and the number of unifications (expensive operation) to perform.

A naive approach consists of running unifications with all facts, on the entire content of the fact store. This technique would be too slow and limit the response time of Active-based applications. To reduce the number of unifications operations, the fact store uses an indexing mechanism. A read operation on the fact store is a two-step process. First, using indexes a subset of the fact store is retrieved. Then, a series of costly unifications are performed on this subset only. If indexes are carefully chosen, the number of unifications to perform can be substantially reduced. Note that if this technique significantly improves read operations, since indexes have to be generated on the fly, it has a small cost [1]on write operations. For each fact inserted into the fact store, indexes need to be generated and index tables updated. The current implementation of the Active store uses the following indexes:

- The Active Ontology name. Since the fact store is a central repository for all deployed Active Ontologies, this index pre-sorts candidates on an Ontology-basis.
- Fact signature. As facts are asserted into the facts store, a signature made out of the *name* of the stored fact and the Active Ontology where it belongs. The name of a fact depends on its type (see section 4.2.1 on page 41 for a definition of fact types). For *simple facts*, the name is the a string representation of the fact itself. For *complex facts*, the name is the identifier of the top structure. For instance, the name of : employee(name(john), lastname(doe), ssn(1234)) would be employee.
- Fact unique identifier. Upon insertion, each fact is given a unique identifier, that can be used as a key for fast access to a specific fact.
- Creation pass. Finally, each fact is tagged with the id of the evaluation pass during which it was created. This allows for fast retrieval of *events*, or facts that were asserted within the current pass. This set of facts is queried for *check-event* tokens of rule conditions.

[1] About 20 microseconds per write operation on our setup

To characterize the behavior of the Active Server, a series of measurements where taken on the Active Server.

First, we created a simple Active Ontology with a single rule whose condition is a single store-check. The condition of the test rule is of the form : Store.checkEvent(test(simple))

To trigger the evaluation of the test rule, a test fact of the form test(simple) is asserted. Over multiple tests, the test rule has been replicated to increase the number of check-facts to be evaluated. Graph 7.8 shows that rule condition

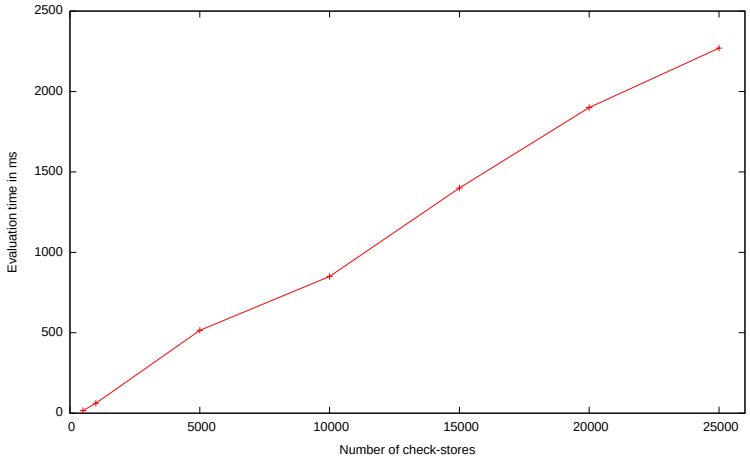


Figure 7.8: Evaluation time vs Number of check-facts

evaluation time grows linearly as the number of rules expands. In addition to the linear trend shown on the graph, it is also interesting to provide absolute values. On our test machine, evaluating a simple rule whose condition has a single check-fact takes about 45 microseconds. This value will be used to assess how performances evolve as the rule conditions get more complex and the size of the fact store grows.

To verify that our indexing technique behaves as expected, we pre-populated the fact store with up to one million facts of the form value($V) where $V ranges from 0 to one million. Using a check-fact of the form value($V) the Active Server performed up to one million read accesses to the fact store. The *clean store* chart on figure 7.9 on the next page shows how the measuring time evolves in a quasi linear way as the number of facts goes from 10 to one million. Next, we added 10'000 *parasite facts* of the form value2($V) where $V ranges from 0 to 10'000. If the indexing technique works as designed, adding these facts should have a minimal impact on our tests because facts of the form value2($V) should be filtered out by the indexing mechanism early in the read process. The *full store* chart on figure 7.9 on the following page shows that the indexing mechanism allows for constant performances as more facts are added to the store. There are nevertheless two interesting facts to mention. First, the impact of additional parasite facts grows as the overall number of facts grows (1% for 100'000 fact to 4% for 1 million facts). Secondly, as more facts are to be read, the response time per check-fact grows linearly from 40 microseconds fro 10'000 facts to 55 microseconds for one million facts.

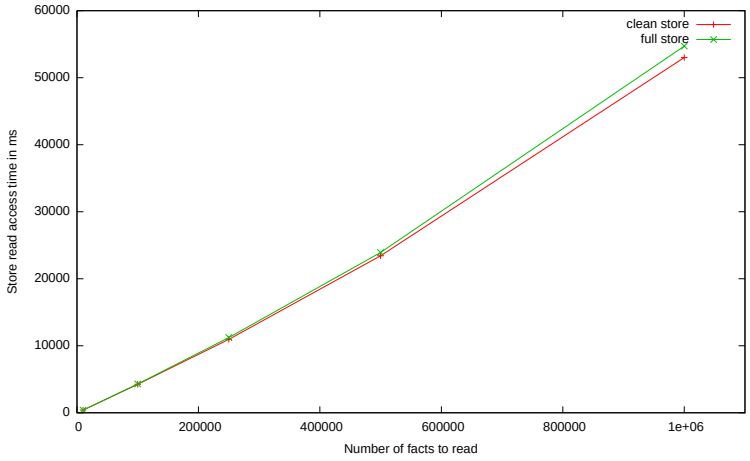


Figure 7.9: Fact store read access

Condition guards The second optimization consists of leveraging guarded conditions. As introduced in section 4.2.7 on page 47, rule conditions consists of boolean expressions that combine *store-checks*. Store-checks are patterns designed to query the fact store using unification. Two types of store-checks can be used when creating complex rule conditions: *check-events* and *check-facts*. Check-events look only at the small set of new facts that actually triggered the current evaluation cycle, whereas check-facts look to unify with all facts of the store, regardless of when they were asserted. Since they query a reduced set of facts, check-events evaluate more quickly than check-facts. When evaluating rules, the Active Server isolates and evaluates check-events first. If a check event is used to guard the rule condition, its negative evaluation aborts the rest of the rule evaluation, thus significantly optimizing the overall response time.

Definition 11 Guarded rule condition

```
Store.checkEvent(GUARD_PATTERN) AND
( ... Store.checkFact(F1_PATTERN) .... Store.checkFact(F2_PATTERN) ... )
```

Definition 11 shows how such expression is constructed by using a check-event that conjunctively, using an *and* operator, conditions the evaluation of the rule condition. This optimization is effective because a very small set of facts (typically less than 10) usually triggers an evaluation cycle, therefore it is a good practice to use a guarded technique where ever possible to ensures better response times.

This technique is used to create *virtual memory spaces*, designed to limit the number of solutions when evaluating rule conditions. For instance, in the *semantic network* language processing technique described in section 5.2.3 on page 71, multiple users dialogs can be simultaneously managed and represented as facts. Each user is assigned a unique session identifier used as a component of all the facts used to store her dialog. When processing events for a specific user, the bound session identifier is used to select and constrain the number of matches during rule evaluations. Similarly, processes modeled with Active

(see section 5.4 on page 109) have a unique process instance identifier, allowing Active to efficiently manage simultaneous process instances. Finally, the *invocation* and *delegation* of services presented in section 5.3 on page 96 guards rule conditions with unique invocation identifiers.

Bound variables in compound expression. As a third optimization, condition variables are used to further optimize the evaluation of complex rule conditions. Before performing an evaluation, store-checks are ordered so that variables get bound early in the evaluation process to restrict and lower the number of solutions. If an empty set of solutions is detected before the full evaluation of a condition, the process aborts, thus saving processing time. To illustrate this technique, let us consider a rule designed to join a person with its employee data, where a person is represented by facts of the form person(firstname, lastname, employee_id) and additional information of the form employee(employee_id, department, title).

Definition 12 Variables binding

```
a) Store.checkFact(person($firstname, $lastname, $PERSON_ID)) AND
Store.checkFact(employee($EMPLOYEE_ID, $department, title))
AND ($PERSON_ID == $EMPLOYEE_ID)
b) Store.checkFact(person($firstname, $lastname, $ID)) AND
Store.checkFact(employee($ID, $department, title))
```

Definition 12 shows two ways of expression the rule condition. First, a naive approach, effectively performing a Cartesian product of all possibilities, followed by a comparison. Assuming the company has n employees, the algorithm would perform in the order $O(n^2)$. The second approach, leverages a bound variable to restrict the solution space of the evaluation. In the example, as facts matching person($firstname, $lastname, $ID) are found, the value of $ID is bound and immediately used to evaluate the second part of the expression employee($ID, $department, title), hence constraining the number of solutions on the fly as the evaluation process unfolds. Much more effective, this technique performs in linearly, in the order of $O(n)$. Graph 7.10 shows measures taken on the Active Server running an Active Ontology exposing rules using conditions shown on definition 12. As the possible matches grows, the number of employees in our example, the response time of the naive approach rapidly degrades, whereas the variable binding technique shows much better response time and exhibits a quasi-linear behavior.

Caching Finally, as a fourth optimization technique, the Active Server caches and stores pre-compiled rule conditions. During the very first evaluation pass of an Active Ontology, rule conditions are parsed and stored as in-memory structures later used for the actual evaluation. The evaluation rule is a two-step process. First, all store-checks are validated and converted into boolean values. For instance, the rule shown in definition 12 after store-checks are replaced with boolean values, could lead to expressions such as:

```
true AND (false OR true )
true AND (true OR true )
```

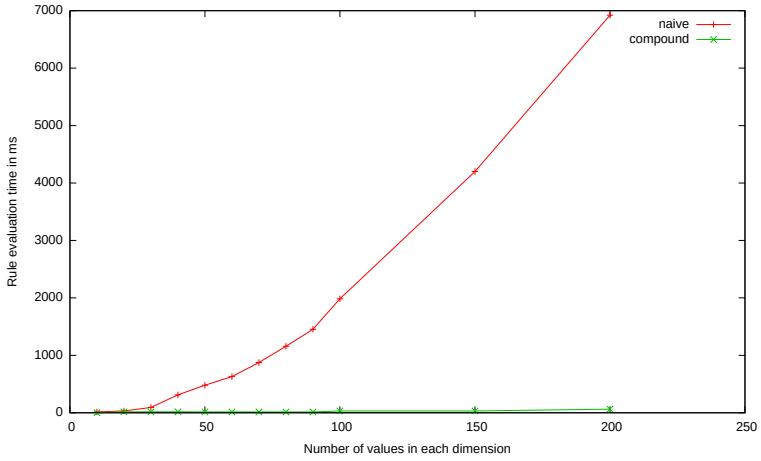


Figure 7.10: Evaluation time with and without variable binding

Once created, such expressions are passed to an interpreter for final evaluation. Since these expressions do not have any variable parts, the result of the evaluation can be cached. Therefore to speed-up the evaluation process, the interpreter first looks into its cache to quickly respond to expressions it has already evaluated.

Event if we elaborate more on full applications later in this chapter, it is interesting to put these results in perspective with existing Active-based applications. Our largest application prototype is the information retrieval assistant presented in section 6.2 on page 127. At each evaluation pass, the system evaluates about 300 rules, requiring the evaluation of approximately 400 check-stores. For a single user, the application creates about 700 facts in the store, adding a 100 more for each concurrent user.

Rule execution

Once the condition of a rule has been evaluated positively, the rule fires and the associated action is executed. Rule actions are code snippets in charge performing tasks, often using the values of bound variables, to be undertaken by a rule.

When it comes to performances, the cost of executing a rule action consists of two parts : the *infrastructure cost* and the action *code execution cost*.

The infrastructure cost is defined as all the preliminary processing, starting after the positive evaluation of the rule, that prepares the execution of the code snippet. To measure and qualify this cost, we have created a rule with an empty code snippet and triggered it multiple times. The chart labeled *do nothing* on figure 7.11 represents the infrastructure cost as the number of executions of the action grows. First, the chart grows linearly and as the number of executions per evaluation cycle goes up, the cost per execution remains constant at 2.3 ms.

This processing time consists of three parts. First, the code snippet is automatically enhanced with system variables such as the name of the ontology and all bound variables. Then, a pre-processing if applied to resolve references

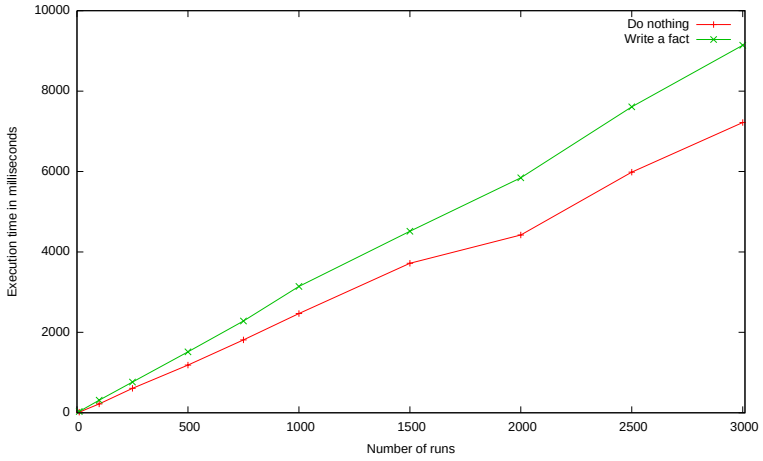


Figure 7.11: Action execution time

to packages defined in Active Server extensions. Finally, since the default programming language offered by the Active platform is Javascript, most of the time is consumed by the parsing and interpretation of the code snippet.

In addition to the infrastructure cost described above, executing actions involves the cost of running the actions described by the programmer. This section is out of the control of the Active platform, there are nevertheless guidelines and best practices to follow when programming with Active.

- Avoid heavy synchronous processing. Active programmers should avoid infinite or long processing loops from happening synchronously with an action code snippet. The current Active Server implementation allocates one thread per Active Ontology, therefore if an action code enters an infinite loop, it suspends the evaluation process of the entire Active Ontology. Intensive processing loops can be performed, but asynchronously through calls to external services using the Active invocation or delegation techniques.
- Use built-in Active Server extensions. The Active Server supports an extension mechanism where pre-packaged java-written libraries of function can be registered and made available to Javascript code snippets. The Active Server comes with built-in extensions to manipulate Active facts, read and write to the fact store, use the invocation technique, call SOAP compliant web services and perform basic logging.
- Create custom Active Server extensions. The Active Server provides an SDKallowing programmers to write and register their own Active Server extensions to encapsulate, register and expose Java-based function to be called from Javascript as a rule action is executed.

Active Ontologies execution and management

Ontology execution management is a weak part of our existing implementation. We use a simplistic algorithm that performs a sequential evaluation of Active

Ontologies. The engine cycles through the list of deployed Active Ontologies to conditionally trigger their evaluation cycles. For each Active Ontology, the server first checks if any relevant fact has been modified since the last execution before triggering an evaluation cycle. Note that testing if relevant facts for a given Active Ontology have changed is very fast (a few microseconds). The fact store keeps a flag for each Active Ontology to signal any change as write, delete or update operations take place on its API.

It would be a major improvement to redesign this part of the Active Server to give one independent thread to each Active Ontology. I would allow our system to be more responsive and, most of all, scale up nicely as the number of Active Ontologies to host becomes large. As the following sections will show, this weakness has a limited impact on our Active-based applications. The functional design of our prototypes implies a sequential execution anyways, as results produced by first tier Active Ontologies (i.e. language processing) has to be completed to sequentially trigger further processing through second tier Active Ontologies (i.e. dialog modeled as a process). It is however a serious limitation as multiple users connect to the system and many evaluations need to be executed in parallel to serve different sessions at various stages of processing.

Conclusion

This section presented a performance evaluation of the Active Server. First, we looked at the basic processing elements involved in the evaluation cycle of an Active Ontology: rule evaluation and rule execution. Then, a detailed description of the implementation and optimizations of the server described the overall performance behavior of our system. Finally, a series a measurement were taken on the server to validate and verify that the implementation of the server actually meets its design requirements.

The Active platform is a research prototype, designed as a flexible and simple tool to experiment our new software technique. It is written in Java, implements a production-rule paradigm and uses an interpreted language as its execution arm. Therefore, even if it proved to be sufficient to run our prototypes, the simple design of the Active Server shows weaknesses and could be further optimized for speed and scalability. Even if the Active Server shows near linear scalability in its operational range, its absolute response time is rather slow. Figure 7.11 shows that our Server requires an average of 1.2 seconds to evaluate 500 simple rules. If this response time is enough for simple user-driven systems, it will not be sufficient for large number of rules that consist of complex conditions. Also, the server has not been thoroughly tested outside of this limited application domain, where linear scalability may not be verified.

7.3.2 Active Language Processing

This section presents the performance analysis the semantic network language parsing presented in section 5.2.3 on page 71. As the most processing intensive part of an Active-based application, the performance analysis of this technique is the subject of a dedicated section.

Reminder of the technique

Language processing with semantic networks is based on a tree-like structure that defines the semantic domain of an application. As user utterances are captured by the environment, they are tokenized into words and injected into the semantic network from its bottom leaf nodes. Leaf nodes analyze incoming words, rate them and report their results to their parent nodes. Parent nodes analyze the messages coming from their children, to come up with their own ratings to be reported up the tree. Through this bottom-up processing flow, a command percolates to the top of the structure. As processing unfolds, information such as user suggestions and error messages are created to provide details and rationales about the decisions made by various nodes along the parsing the process.

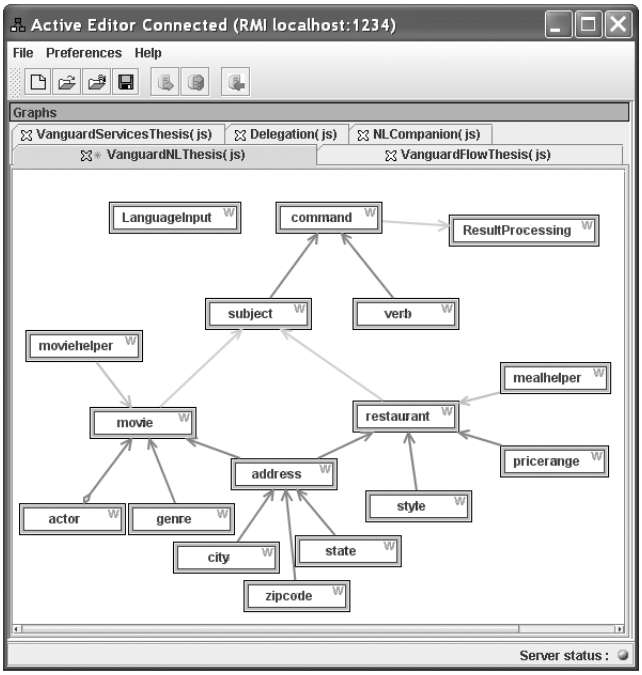


Figure 7.12: Sample Semantic Domain in Active Editor

Measurements and tests reported in this section have been performed on the semantic domain shown in figure 7.12. The domain represents a simple application able to parse queries to retrieve information about movies and restaurants. Sample utterances would be: *'find restaurants in palo alto tonight'*, *'get me only italian ones'*, *'good, now find me nearby movies'*. The domain features all possible leaf node types: *actor* uses a prefix, *genre* has a local list of movie genres, *zipcode* uses a regular expression and *city* is connected to a database containing 30'000 major cities in the US.

Many factors influence the performance and response time of the semantic-network language processing approach. The following sections present the most relevant ones.

Number of rules

As described in the Active Server evaluation section, the performance of an Active-based application depends on the number of rules to be examined at each evaluation pass. In the case of language processing applications, the number of rules will directly depend on the number and types of nodes that make up the semantic network. A semantic network consists of various node types (see section 5.2.3 on page 74for details) :

- *leaf nodes*: at the front line of processing, they get tokens sensed by the environment
- *gather nodes*: create and rate structures out of their children ratings
- *select nodes*: select the single best candidate among their children
- *infrastructure nodes*: two specialized nodes are in charge of providing the parsing infrastructure. First, the *context* node, processes incoming utterances by tokenizing them into words to be injected to the network. It also manages user sessions. Secondly, the *root* node processes values generated by the processing tree.

Node type	# Checks	# Rules	# Rules/token	Processing [ms]
Leaf (reg. exp.)	13	5	3	25
Leaf (local list/DB)	13	7	2	15
Gather	13	5	1	35
Select	12	5	1	35
Language Input	10	5	2	40
Result Processing	14	5	2	40

Figure 7.13: Performance analysis of processing node types

An analysis and a series of measurements are summarized in table 7.13. First, to estimate the evaluation time required by nodes, we list how many rules and check-stores are required for each node type. Next, to assess the execution time, we indicate how many rules actually fire when a new token is submitted to the system. Finally, we provide the total processing time required by each node type during processing. Table 7.13 can be used as a reference to estimate the processing time required by an Active-based language processing application.

Vocabulary size

For leaf nodes that base their semantic ratings on vocabulary sets, two factors are important to consider for good performances: *vocabulary size* and *utterance size*.
First, when using a vocabulary set, the processing time depends of vocabulary size. Initially, the words of a given vocabulary can be stored as facts into the Active Server fact store. Secondly, the number of words to consider directly impacts performances. For instance, let us consider the *state* leaf node, instrumented with a rule that checks incoming tokens with a list of states. To detect

two-word state names (i.e. *rhode island*, *new mexico*) as well as one-word state names (i.e. *colorado*, *california*), the *state* node checks any single token and consecutive pairs of tokens against its vocabulary.

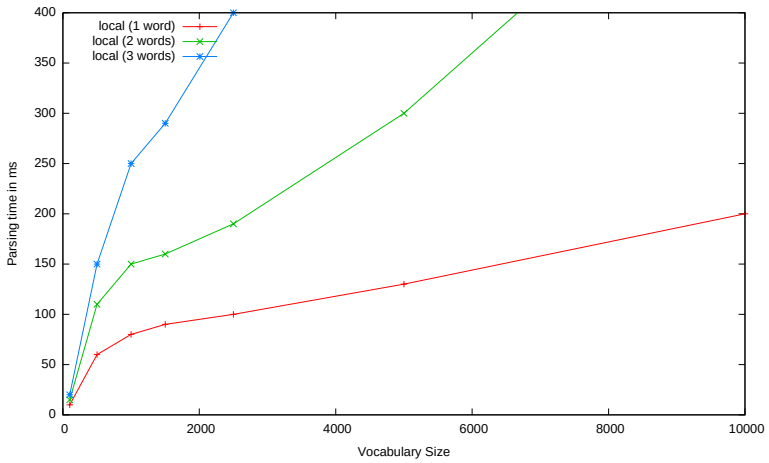


Figure 7.14: Locally hosted vocabulary sets

Figure 7.14 illustrates how the parsing time evolves as these two parameters change. First, let us consider simple one-word utterances. After a sharp response time increase to reach a 1000-word vocabulary, performances degrade linearly as the vocabulary size grows to 10'000 words. Then, for 2-word utterances, the same behavior is measured. However, the response time increase for larger vocabularies is much sharper. Finally, let us look at three-word utterances. The degradation in response time is so dramatic, that considering three-word groups would prevent us from using large vocabularies. Since we intend to support rich user requests consisting of up to 10-word utterances, and possible large vocabularies of hundred of thousand words, we need to find a way of reaching better performances.

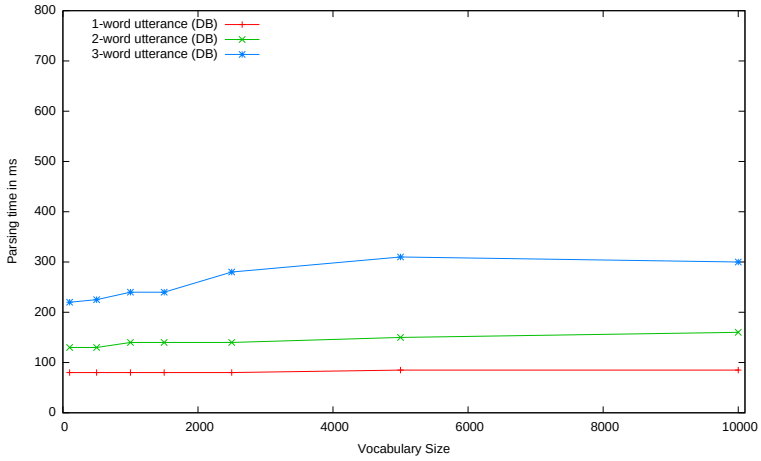


Figure 7.15: Local hosted vocabulary sets

Therefore, for large vocabularies we developed a JDBC connector so that words and synonyms can be stored in database servers. This technique offloads most of the processing from the Active Server to the database server through SQL queries. Figure 7.15 shows how performance is dramatically improved when vocabularies are stored in a database. For 1-word, 2-word and 3-word utterances, the response time remains almost constant, respectively 80 ms, 110 ms and 250 ms, as the vocabulary grows from 10 to 100'000 words.

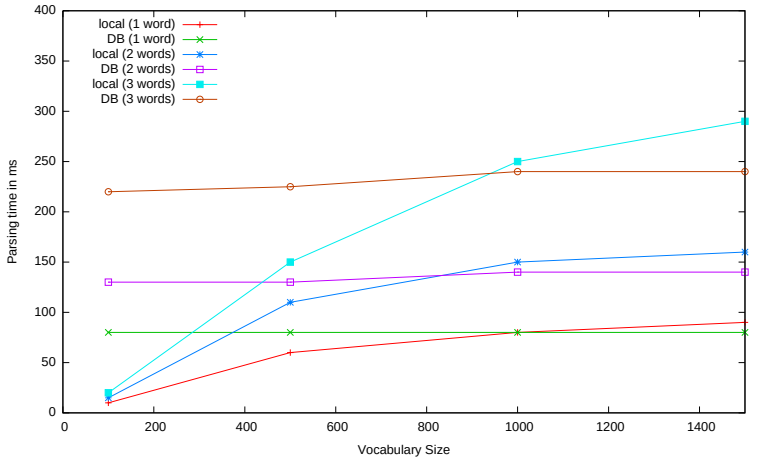


Figure 7.16: Local vs DB hosted vocabulary sets

However, connecting to a database has a fixed cost, about 80 milliseconds, whereas looking up words locally in the Active Server can be very fast for small vocabularies. Figure 7.16 shows that using local vocabularies is faster if the number of words is smaller than 1000. Above that, it is much more effective to use a database system.

Tree depth

The parsing process being a bottom up operation, starting with leaf-node ratings being reported and processed up the tree by *selection* and *gathering* nodes, depends heavily on the depth of the semantic model of the application domain. Processing of each level of the tree (see figure 7.12) requires an Active Ontology evaluation pass. During the first pass, leaf nodes get tokens, generate semantic ratings and report them to their parents' nodes. At the second pass, parents directly connected to the leaves process their children inputs to generate and report ratings to the own parent, who will react at the next evaluation pass. Therefore, the technique requires as many evaluation passes as there are level in the semantic model of the domain. Evaluation passes being run every 100 ms, parsing a ten-level tree would take at least one second.

The current implementation of the Active Server provides solutions to limit the impact of tree depth. As explained in section 4.2.10 on page 51, concepts can be given an execution priority, where high priority concepts get executed first in a given evaluation pass. In the context of our parsing technique, setting these priorities based on the depth of concepts (nodes and leaves) tree ensures that deeper nodes are evaluated and produce ratings first. This way, the parsing tree generates results in one single evaluation pass.

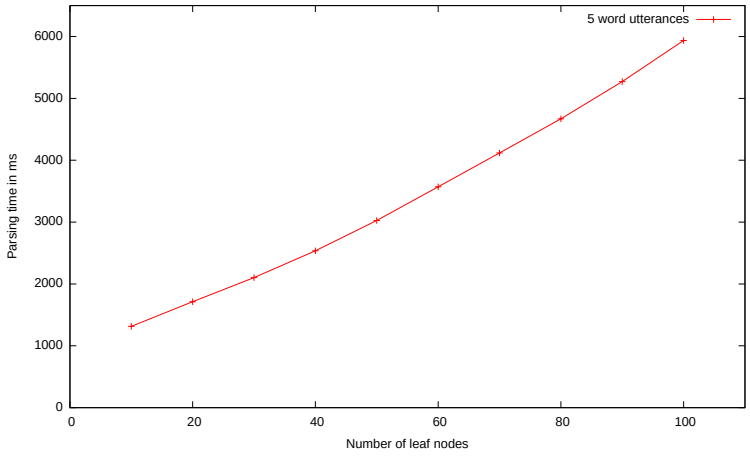


Figure 7.17: Parsing time vs number of leaf nodes

More generally, the number of leaf nodes will directly impact the response time of our language parsing technique. Figure 7.17 shows that the response time augments linearly with the number of leaf nodes. This is explained by the underlying infrastructure of the Active Server that also shows linear response time as the number of rules to process augments.

Number of events to process

Large utterances lead to longer processing time. The larger the utterance, the more tokens will be injected to the system and the greater the parsing time becomes. The processing load is different whether the user expresses a query

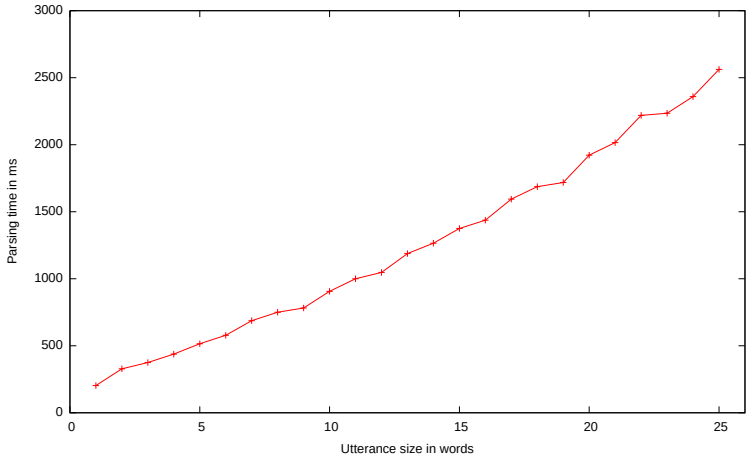


Figure 7.18: Parsing time vs utterance size

such as *"find me a good an italian restaurant tonight in palo alto, California"* or *"italian restaurants palo alto"*. Figure 7.18 shows how the Active-based language processing time increases as utterances get longer. First, it is interesting to notice that as the number of words grows, the response time of the parser augments linearly. In our experiment, the number of words per utterance ranges from 1 to 25, where a one word utterance requires about 200 milliseconds to parse, a twenty-five-word utterance takes up 2.5 seconds for parsing.

These numbers show that the current implementation of our system fulfills our needs. First, our system being designed for user-centric applications and not as a system for information extraction or classification of large texts, user utterances to process are expected to be short. Secondly, we need to define what we intend by *'short'*. Literature shows that Google search queries currently average to around 2.5 words per request [42], on both mobile and desktop situations. However, the goal of intelligent assistants such as Active applications is to make users more comfortable in delegating complex tasks through longer utterances. This is the reason why we are testing our system with longer utterances, to ensure a good experience as users get more expressive using longer utterance. Section 6.2 on page 125provides more details on the size of user utterances expressed when using our prototypes.

Reference resolution and disambiguation

Section 5.2.3 on page 86, has shown how Active includes semantic validation of values as part of the parsing process. Asking third party providers for help in the middle of the parsing process impacts the responsiveness of the system. For clean design purposes, helpers are implemented as rules and concepts deployed as part of a *service management* Active Ontology, invoked through the invocation mechanism (see section 1.2 on page 12). Therefore, using reference resolution and disambiguation suspends the execution of the language processing until helpers have provided results. This requires at least two full evaluations passes, and our evaluation policy being sequential, requires the engine to cycle

twice over all deployed Active Ontologies. As shown earlier, cycle through all know Active Ontologies has a limited performance impact because ontologies whose fact store have not been modified are skipped. On our test setup, using one or more helpers adds an overall cost of about 400 milliseconds the parsing time.

To improve user experience, the first evaluation pass of the parsing tree notifies the user that it is waiting for helpers to provide additional information. An immediate notification message, such as *"Thank you processing your request"*, keeps the user attention and provides helpful feedback.

Number of concurrent user sessions

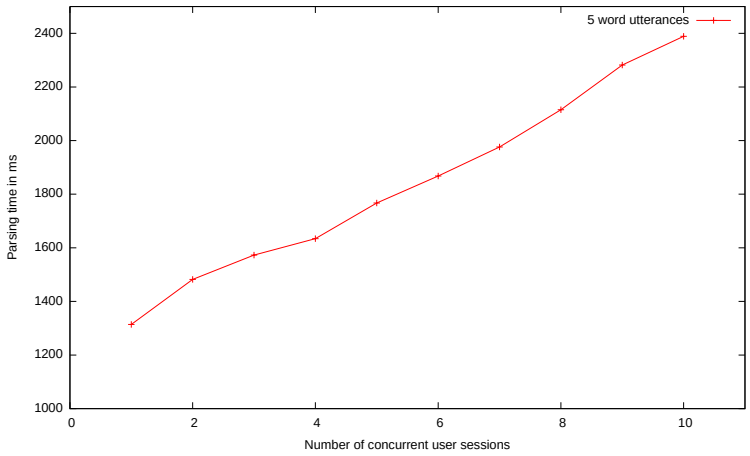


Figure 7.19: Parsing time vs number of sessions

In this section we examine how the processing time varies as the number of concurrent user sessions grows. The Active-based language processing technique holds a context on a user basis. It allows for follow-up questions and simple dialogs where users do not have to specify the entire information at each utterance. On the design side, this is implemented using a set of facts tagged with a unique user session identifier, so that a single instance of the language processing Active Ontology can concurrently manage multiple user sessions. Figure 7.19 shows that the response time of the system grows linearly as more user sessions are created. This is a positive fact, ensuring that scalability can be reached by replicating servers and dispatching sessions to different machines using a load balancing technique.

Best practice

There is series of Active usage best practices to optimize language parsing performances. First, rule evaluation optimizations such as *guarded conditions* and *variables binding* improve performances. Secondly, heavy processing should not be encoded in local Javascript interpreted code snippets, but rather implemented in Java and packaged as Active Server extensions.

Conclusion

This section presented a performance evaluation of the language processing technique based on semantic networks. We first characterized the performance of a semantic network by measuring the impact of each type of node used to build the application domain. This gives us a reference chart used to approximate the response time of a language processing based on the number and types of nodes.

We then analyzed how vocabulary sizes and the number of words to test affect performances. If using a database shows good performances for large vocabularies, it is nevertheless faster to used the Active fact store for application requiring vocabularies smaller than one thousand words.

We then show how using rule priorities of execution limits the impact of application domains with large depths. If not built-in and optimized, algorithms used for reference resolution and disambiguation can have a significant impact on the response time of the system. The semantic network technique supports multiple user session simultaneously, however, as the number of users grows, performances are affected.

7.3.3 Discussion

This section presented the performance evaluation of two components of the Active system. First, a performance evaluation of the Active server shows that its current implementation is powerful enough to run all our prototypes. However, the evaluation draws the reader's attention to bottle necks and design implementation to be taken into consideration when building a commercial product. Secondly, as the most processing intensive Active method, the performance behavior of the language processing technique is analyzed and measured. Similarly, even if the technique fulfilled both functional and performance requirements or our prototypes, it shows weaknesses as the application scales.

Given the interest shown by the commercial world to our research, this discussion summarizes our performance evaluation and leverages our experience to list important features a commercial product inspired by the Active system should provide.

Performance In addition to the existing optimizations implemented by the Active system, well known optimized techniques[19], faster storage systems and a more parallel application design need to be used. As a core processing element, unification algorithms and fact data representations need to be careful designed and implemented.

Scalability In terms of scalability, multiple components need to be carefully designed. First, as the amount of information grows, the fact store needs to provide good performance for both reading and writing fact-based information. On the deployment side, a cluster-based approach needs to be considered so that workloads can be distributed over multiple instances of servers.

Security Since the information stored and managed by intelligent assistants can be sensitive, a full AAA security policy needs to be part of the implementation.

Robustness To be trusted and be able to perform actual transactions, intelligent assistants need to provide a high level of robustness. For instance, the need to recover from catastrophic failures, support transactional processes and provide flawless detection and processing of exceptional situations.

Programming and Administration tools The IDE used by programmers to program and debug applications on an Active-like system should be based on a robust platform, such as popular Eclipse or NetBeans frameworks. Administrative tools, should be web-based and open to be integrated with well established management consoles such as Tivoli or OpenView, using integration standards such as SNMP traps.

7.4 Conclusion

This chapter presented evaluations about three aspects of the Active platform.

First, it described an end-user evaluation of the system. A population of users were asked to perform a set of ten on-line tasks related to travel, including getting a weather forecast, finding information about hotels, restaurants, flight status and various points of interest. Subjects had to perform the test with both an Active-based assistant program and Google Mobile^{TM}, the leading commercially available product for mobile search. Our intelligent assistant application performed better than its commercial counterpart. The assistant-like system provided a better completion rate (90% versus 54%), required a smaller number of requests to complete the ten-tasks (13.6 versus 48.6) and produced faster average task completion (1.3 minutes versus 4.33 minutes). This validates our claim, stating that in some cases, intelligent assistant applications better server users than more conventional software systems.

Secondly, an evaluation of Active as a programming language was conducted. A population of programmers, mostly unfamiliar with AI-based systems, trained used the Active system and then were asked to create a language processing application. Once trained, they were able to successfully create an application, able to understand nearly 80% of a one hundred-utterance corpus created from actual user utterances (from the first experiment that performed better than Google). The whole process, from training to evaluation, took less that two hours per participant. This results supports our claim, stating that a unified tool and associated technique eases the development of intelligent assistants and their AI-based technologies..

Finally, a performance evaluation of the core Active Server and its language processing technique is provided. The current implementation of the Active Server has shown enough processing power to successfully run our evaluation prototypes. It has nevertheless not been designed as a commercial product capable of scaling up to large amount of data and high levels of transactions. The evaluation pinpoints weaknesses and bottlenecks to provide suggestions about implementing a production ready system inspired by out work.

Chapter 8

Conclusion

The work described in this document presents and evaluates an innovative approach to ease the development of AI-based intelligent assistant software. This concluding chapter starts with a summary of our research domain and a description the claims that motivated our work. This is followed by a list of achievements that helped us support and validate our claims. Finally, future directions and potential use of our work in both academic and commercial fields are provided.

8.1 Contributions and results

Application context

Computer systems keep growing in complexity, processing power and web connectivity. To better leverage this rich environment and to assist users, we believe a new type of assistant-like software will emerge, that goes a step beyond conventional click-and-do paradigm by allowing users to delegate through natural interaction some of their activities. Although progress towards these goals has been made in the last decade, recent research has shown that building intelligent assistants is a difficult and complex task that requires expertise in many AI and engineering related fields. We believe that in order to accelerate widespread adoption of this type of software, a breakthrough is required that simplifies and acclerates intelligent agent development. In this context, we have developed a new methodology and toolset based on the notion of Active Ontologies that aids in the creation of end-to-end intelligent software integrating technologies such as natural language understanding, dialog, process modeling, and service orchestration.

Contributions

Our work consisted of validating two claims at the source of our research:

- Advantages of intelligent assistants. We have shown, through the implementation of prototypes and comparative evaluations, that intelligent assistants can be more effective in some domains than conventional systems.

- Unified approach. We have also demonstrated that a coherent and unified approach can speed the design, implementation, testing and deployment of intelligent assistant applications.

Achievements

To support and validate our claims, our work can be summarized through the following achievements.

Active Ontologies and associated methods The first achievement has been to devise the concept of Active Ontologies and associated techniques. Active Ontologies unify several AI techniques, including ontologies, production rule engines, agent technologies and neuroscience-inspired systems.

Based on this original approach, a set of programming techniques has been designed. First, basic programming schemes such as message passing over communication channels and asynchronous invocation have been drafted. Then, a collection of high level methods to implement language processing, process modeling and dynamic service selection and invocation have been elaborated.

Active framework implementation To validate the techniques described in the previous paragraph, we created a suite of tools to program, test and deploy Active Ontologies. The current implementation of Active software suite consists of four components: the *Active Editor*, the *Active Server,* the *Active Console* and the *NL Test Tool*. Additionally, a set of programmatic extensions have been developed to support programmers develop AI systems.

The Active Editor is a design environment used by developers to model, deploy and test Active Ontologies. Within the Active Editor, developers can graphically create and relate concept nodes. Manually write code or select *Wizards* that automatically generate rule sets within a concept to perform complex pre-packaged operations.

The Active Server is a runtime engine that hosts and executes one or more Active Ontologies. It can either be run as a standalone application or deployed on a J2EE compliant application server. The Active Server exposes SOAP and RMI protocols to allow external sensors component to report their results by remotely inserting information into fact stores, thus triggering the evaluation of concept rules within the deployed Active Ontologies.

The Active Console permits observation and maintenance of a running Active Server. The console can be used as a test tool by injecting events into an Active Server to trigger processing and monitoring the results by inspecting the content of the fact store. In addition, the Active Console offers tools to remotely manage the set of Active Ontologies deployed on an Active Server.

The NL Test Tool allows a programmer to test and debug natural language applications using an interactive approach as well as a batch-oriented regression test system.

Finally, a collection of Active Extensions has been developed to help programmers build Active-based systems. The Active framework implementation is a Java-based software suite designed to be extensible and open. For both the Active Editor and Active Server, plug-in mechanisms enable researchers to

package AI functionality allowing developers to share, apply and combine 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. In addition, utility extensions have been implemented to allow Active programmers to use email capabilities (both sending and receiving), connect to the Instant Messenger network and leverage a simple REST API to build web-based applications.

Application design Based on the implementation of the Active platform and its associated methods, we elaborated a simple set of guidelines to design and implement Active-based applications.

The overall application design consists of a collection of specialized Active Ontologies (language processing, process logic and service orchestration) and a community of loosely coupled services.

In addition to this overall architecture design, we devised a technique to extract Active Ontologies out of initial application requirements. The responsiveness of the application is also taken into account to ensure users comfort and acceptance of the system.

Prototypes To evaluate and validate our initial claims, concepts, tools and techniques presented above, a set of three Active-based prototypes have been implemented and evaluated.

Mobile user assistant An assistant designed to help mobile users with online activities was designed and implemented. The idea is to provide mobile users with an interaction mode based on natural dialog over multiple utterances to access online data and services.

For instance, when looking for an affordable French restaurant in Miami, instead of accessing the Internet on an embedded limited browser, users can send messages in plain English, such as *"Find me affordable French restaurants in Miami"*. The answer is a message providing the list of relevant restaurants. In addition to retrieving information, the system is able to undertake transactions on behalf of the user, such as making reservations. For instance, if *"Chez Paul"* is in the list of returned restaurants, an actual booking can be expressed as: *"good, book me a table for two tonight at chez paul"*. Further requests leverage dialog context for follow up questions. For instance, once a specific restaurant has been picked, looking for nearby florists can be expressed with a follow up query as: *"get me nearby florists"*. The system covers multiple domains such as restaurants, movies, hotels, points of interest (florists, ATMs, fitness center, etc...), real time flights information and weather forecasts.

This prototype helps us achieve two goals. First, as the first Active-based prototype, it validates our implementation and design approaches by fulfilling all the requirements imposed its initial definition. Secondly, the prototype was used to demonstrate that an intelligent assistant using a combination of natural language and dialog, is more effective than a more conventional keyword based approach to retrieve online information. The evaluation consisted of asking a population of twenty test users to accomplish a series of tasks using both our system and the leading commercial application designed for mobile users (Google Mobile^{TM}). The results show that using an assistant-like system in a

specific domain can be more effective than a keyword-based search engine. In our test, 90% of required tasks could be completed by subjects using the Active-based system, whereas only 50% could be achieved with Google Mobile^{TM}. In addition, the assistant-like system required 14 queries to complete all ten tasks, whereas the search engine-based system needed 49 queries to complete the test.

Finally, our prototype received a positive feedback from the commercial world. The backend of our system has been used as a fully functional demonstrator to successfully fund and launch a company whose goal is to better help users on the move. Also, a large database system vendor is evaluating how an Active-based approach could help users query over large datasets using natural language and dialog.

Operating room assistant The second prototype built on top of Active provides help to surgeons in the operating room. The application is implemented as a multimodal system allowing surgeons to manipulate pre-operative data, visualize live images coming from an endoscope and control a robotic arm. Surgeons and their staff interact with the system by a combination of hand gesture using a contact-less mouse and voice recognition.

This prototype helped us validate and improve our work along two axes. First, as the second Active-powered application that fulfills all its initial requirements, it further validates our tools and techniques. This second prototype bears multiple differences with our first application dealing with online services. Both prototypes are deployed in very different application domains, and yet are built with the same tools and methods. It demonstrates the flexibility and versatility of our approach. This specific prototype focused on multimodal fusion and on incorporating real-world effectors.

The prototype was evaluated by a small population of surgeons to determine if an assistant-like approach is suitable for the operating room. Surgeons agree that the approach is relevant for two main reasons. First, it allows them keep their attention and focus on the patient while interacting with computer-based systems. Secondly, the assistant federates all computer systems in one single intuitive and coherent interface that hides some of the complexity of the underlying system. Results also showed that a combination of gestures and simple sounds emerges as a promising interaction mode between computers and surgeons. However, on the implementation side, two major constraints need to be overcome for a clinical use. All system components have to be certified to be used in the operating room and vendors of surgical equipments need to agree on integration standards.

Meeting organizer assistant The third prototype implemented consists of an assistant designed to help organize meetings. The assistant interacts in a natural way (plain English) with the meeting organizer and its attendees through instant messages and email.

Similarly to the prototypes presented above, creating this application and ensuring that is complies with all initial requirements allowed to further improve and validate our platform and its associated techniques. This prototype focused on implementing and assistant who can manage long-running transactions, as opposed to the request-response interaction characteristic of our other prototypes.

A second goal behind the system was to evaluate how the Active platform could be used for rapid prototyping. The aim was to design, implement and validate the application in less than a week. A simple, but fully functional, version of the system was completed in three eight-hour work days.

Finally, this application domain being popular for intelligent assistant systems, our implementation has been compared with existing similar meeting scheduler assistants. As a general tool for AI-based systems, Active produced fully functional system that includes language processing, a simple scheduling logic and uses a community of services to communicate, reason and act. Other applications do not offer the full spectrum of components, but are much more advanced and effective on some specialized components, for instance scheduling policies. The conclusion is that Active can be used to rapidly create an end-to-end system that can be further improved in specific components. Our approach isolates components as either specialized Active Ontologies or external services, allowing programmers and researchers to gradually enhance the system by swapping in new experimental components.

Evaluation In addition of evaluating each of the three prototypes for functionality and user studies, the programming tools and the core components of the Active system have also been evaluated.

First, a performance evaluation of the current implementation of the Active Server shows its strength and weaknesses. The evaluation shows that most components of the Active Server scale linearly as the required processing load augments. Scalability problems could therefore be solved by replicating multiple instances of the Active Server. The analysis also mentions processing optimizations and shows their positive impact on response time. Programming best practices have also been introduced to leverage these optimizations. Finally, evaluations show that the performance of the current Active Server implementation was sufficient to implement three research prototypes. However, the performance analysis found weaknesses in terms of running multiple Active Ontologies simultaneously, robustness to catastrophic failures and rule evaluation techniques. Our implementation is therefore not suited *as-is* for large systems, but is powerful enough for research and prototyping purposes.

The programming tools and methods developed for Active programmers have also been evaluated. After an introduction to the Active platform and a short training, a population of programmers was asked to create an Active-based application. They were asked to create a language processing application able to parse a reference corpus of a dozen utterances representing travel related queries. In average, it took each participant about one hour to their applications, which were able to extract 95% of the information contained in the reference corpus. In addition, each application was tested against a realistic corpus, made out one hundred travel-related utterances collected from actual end-users when running the evaluation of our *mobile user assistant*. Results show that nearly 80% of the information contained in the real end-user corpus was successfully parsed and extracted by the applications created by the programmer evaluation subjects. These results help us validate our claim, that a unified tool and associated methodology for creating intelligent assistant application eases the design and implementation of such systems.

Finally, as the Active platform was developed, the software industry showed

interest in our research and evaluated the system. A leading cellular phone provider has positively evaluated our assistant for mobile users. The positive outcome started a knowledge transfer process, aimed at creating a commercial version of an intelligent assistant whose overall design is inspired by our research. In an other industry, a major player in the database business is also evaluating our system. The goal is to provide access to large databases through natural language and dialog.

Documentation As part of our work, a set of documents related to our research has been written.

In an effort to expose and confront our ideas to the academic world, five peer-reviewed publications have been submitted, accepted and presented at international conferences[29, 27, 30, 28, 26]. As a separate publication, the concept of Active Ontologies and associated techniques is the subject of a patent filed in the United States. (see appendix A)

To introduce our work to potential developers, a one-hundred page *Active Developer's Guide* has been written. The document provides an introduction to Active, step-by-step tutorials for building language processing applications and a description of the *Active SDK*. The Active SDK is a set of classes to be used when writing extensions and plugins that encapsulate AI techniques or processing libraries.

Finally, a web site exposes the Active philosophy and some of our achievements. (*http://imtsg14.epfl.ch/active*)

8.2 Future Work

The result of our project being an actual tool and associated techniques, it opens the door to multiple exciting challenges multiple domains.

Research

On the research side, additional AI-based techniques can be encapsulated and expressed through the Active prism.

BDI systems A first domain would be BDI-based techniques. Reactive planning has proved [85] to be an effective technique to model intelligent assistants. Our existing process modeling method and its extensions would be a good base to implement BDI planners.

Learning The Active system makes a limited use of learning techniques. Providing a learning modules, for instance a classifier, could be used in many applications. For instance, when building a natural language processing Active Ontology, learning could be used as another disambiguation technique.

Activity recognition We developed and tested the semantic networks technique in the domain of language processing to determine the meaning of incoming utterances. In a broader view, semantic networks could model high level activities to help intelligent spaces understand user intentions. Smart spaces could then provide relevant help, monitor and even take over some tasks.

New application domains We identified two fields where the Active platform could contribute as the software base for research systems. First, the field of transient computing or intelligent spaces. Research in this domain consists of instrumenting a space, typically a room, with multiple sensors. The space becomes aware of the local activities to react and provide relevant contextual help. Examples are intelligent meeting rooms able to automatically create meeting notes and facilitate communications, or intelligent homes providing help to their residents, especially for senior citizens.

Another domain where Active could contribute is robot-human interaction. An Active Server could be installed on a mobile robot to manage all interactions the robot may have with human users. In addition, Active can model interactive user interfaces to delegate complex tasks to robots.

Software suite

If the Active tools are have been tested and proved effective when designing our collection of prototypes, there are many areas where they can be improved and further developed.

Active Editor The current Active Editor is a standalone Java-based application that uses Swing for its user interface components and a specialized package for graph rendering and manipulation. Re-writing the Active Editor as an Eclipse plugin would make it more robust, easy to use by leveraging many existing functionalities of the Eclipse platform and, most of all, popular and easy to distribute. Once integrated into Eclipse, a community of programmers could create a collection of specialized addons. For instance, some functionalities of the Active Console (reading the content or installing triggers on the fact store) could be used directly from the IDE, making the processes of testing Active systems faster and easier.

Active Server If all three Active-powered prototypes implemented so far have shown enough responsiveness, our performance analysis of the Active Server showed weaknesses and suggests potential improvements. In particular, the fact store should not be an in memory only store, but provide persistence to recover from catastrophic failures. In addition, the production rule engine could be re-engineered to implement highly optimized evaluation techniques for better performance and scalability. Finally, our implementation does not include any security features. Since personal intelligent assistant can hold and manage personal information, it is crucial to secure and control the access to the system data and services.

Bibliography

[1] S. Abney. Partial parsing via finite-state cascades, 1996.

[2] J. Anderson. *Rules of the Mind*. Lawrence Erlbaum Associates Inc, New Jersey, July 1993. Gibts in Golm unter 'CP 4000 AND'.

[3] R. Bastide, D. Navarre, P. Palanque, A. Schyn, and P. Dragicevic. A model-based approach for real-time embedded multimodal systems in military aircrafts. In *ICMI '04: Proceedings of the 6th international conference on Multimodal interfaces*, pages 243–250, New York, NY, USA, 2004. ACM Press.

[4] P. Berry, K. Conley, M. Gervasio, B. Peintner, T. Uribe, and N. Yorke-Smith. Deploying a personalized time management agent. In *Proceedings of the Fifth International Joint Conference on Autonomous Agents and Multi Agent Systems (AAMAS-06) Industrial Track*, Hakodate, Japan, may 2006.

[5] P. Berry, K. Myers, T. Uribe, and N. Yorke-Smith. Constraint solving experience with the calo project. In *Proceedings of CP Workshop on Constraint Solving under Change and Uncertainty*, pages 4–8, Sitges, Spain, oct 2005.

[6] A. Broder. A taxonomy of web search. *SIGIR Forum*, 36(2):3–10, 2002.

[7] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-oriented software architecture: a system of patterns*. John Wiley & Sons, Inc., New York, NY, USA, 1996.

[8] P. Busetta, R. Ronnquist, A. Hodgson, and A. Lucas. Jack intelligent agents - components for intelligent agents in java, 1999.

[9] J. Cassell. Embodied conversational interface agents. *Commun. ACM*, 43(4):70–78, 2000.

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

[11] A. Cheyer, J. Park, and R. Giuli. Iris: Integrate. relate. infer. share. *1st Workshop on The Semantic Desktop. 4th International Semantic Web Conference*, page 15, nov 2005.

[12] M. Cilia and A. Buchmann. An active functionality service for e-business applications, 2002.

[13] F Curbera, M. Duftler, R. Khalaf, W. Nagy, N. Mukhi, and S. Weerawarana. Unraveling the web services web: An introduction to soap, wsdl, and uddi. *IEEE Internet Computing*, 6(2):86–93, 2002.

[14] R.B. Doorenbos. *Production matching for large learning systems.* PhD thesis, Pittsburgh, PA, USA, 1995.

[15] G. Dorais and K. Nicewarner. Adjustably Autonomous Multi-agent Plan Execution with an Internal Spacecraft Free-Flying Robot Prototype. In *Proceedings of ICAPS'03 Workshop on Plan Execution*, Trento, Italy, June 2003.

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

[17] G. Ferguson and J. Allen. TRIPS: An integrated intelligent problem-solving assistant. In *AAAI/IAAI*, pages 567–572, 1998.

[18] D.M.R. Ferreira and J.J. Pinto Ferreira. Developing a reusable workflow engine. *J. Syst. Archit.*, 50(6):309–324, 2004.

[19] C. Forgy. Rete: a fast algorithm for the many pattern/many object pattern match problem. pages 324–341, 1990.

[20] Ernest Friedman-Hill. *Jess in Action : Java Rule-Based Systems (In Action series).* Manning Publications, December 2002.

[21] D. Georgakopoulos, M.F. Hornick, and A.P. Sheth. An overview of workflow management: From process modeling to workflow automation infrastructure. *Distributed and Parallel Databases*, 3(2):119–153, 1995.

[22] J. Giarratano and G. Riley. *Expert Systems: Principles and Programming.* Brooks/Cole Publishing Co., Pacific Grove, CA, USA, 1989.

[23] C. Graetzel, S. Grange, T. Fong, and C. Baur. A noncontact mouse for surgeon-computer interaction, 2003.

[24] R. Grimm and B. Bershad. Future directions: System support for pervasive applications, 2002.

[25] R. Grimm, J. Davis, E. Lemar, A. MacBeth, S. Swanson, S. Gribble, T. Anderson, B. Bershad, G. Borriello, and D. Wetherall. Programming for pervasive computing environments, 2001.

[26] D. Guzzoni, A. Cheyer, and C. Baur. Active : A unified platform for building intelligent web interaction assistants. In *Web Intelligence, WI-2006*, pages pp.417–420, 2006.

[27] D. Guzzoni, A. Cheyer, and C. Baur. Active, a platform for building intelligent software. In *Computational Intelligence, CI-2006*, pages 121–125, 2006.

[28] D. Guzzoni, A. Cheyer, and C. Baur. Active, a platform for building intelligent operating room. In *Proceedings of Surgetica07*, number 1, 2007.

[29] D. Guzzoni, A. Cheyer, and C. Baur. Active, a tool for building intelligent user interfaces. In *Proceedings of the 11th IASTED International Conference on Artificial Intelligence and Soft Computing*, number 1, 2007.

[30] D. Guzzoni, A. Cheyer, and C. Baur. Modeling Human-Agent Interaction with Active Ontologies. In *AAAI Spring Symposium, Interaction Challenges for Intelligent Assistants*, volume 1, pages 52–59, 2007.

[31] J. Hammerton, M. Osborne, S. Armstrong, and W. Daelemans. Introduction to special issue on machine learning approaches to shallow parsing. *J. Mach. Learn. Res.*, 2:551–558, 2002.

[32] J Hawkins and D. George. Hierarchical temporal memory. concepts, theory, and terminology. Technical report, Numenta Inc., Menlo Park, California, March 27 2007.

[33] S. Helal, W. Mann, H. El-Zabadani, J. King, Y. Kaddoura, and E. Jansen. The gator tech smart house: A programmable pervasive space. *Computer*, 38(3):50–60, 2005.

[34] G. Herzog, H. Kirchmann, S. Merten, A. Ndiaye, and P. Poller. Multiplatform testbed: An integration platform for multimodal dialog systems, 2003.

[35] B. Hodjat and M. Amamiya. Applying the adaptive agent oriented software architecture to the parsing of context sensitive grammars, 2000.

[36] B. Hodjat and M. Amamiya. Introducing the adaptive agent oriented software architecture and its application in natural language user interfaces. In *First international workshop, AOSE 2000 on Agent-oriented software engineering*, pages 285–306, Secaucus, NJ, USA, 2001. Springer-Verlag New York, Inc.

[37] K. Höök. Steps to take before intelligent user interfaces become real. *Interacting with Computers*, 12(4):409–426, 2000.

[38] DiCesare C. Hoxmeier, J.A. System response time and user satisfaction: an experimental study of browser-based applications. In *Proceedings of the Association of Information Systems Americas Conference*, 2000.

[39] M. Huber. Jam: a bdi-theoretic mobile agent architecture. In *AGENTS '99: Proceedings of the third annual conference on Autonomous Agents*, pages 236–243, New York, NY, USA, 1999. ACM Press.

[40] G.W. Johnson and R. Jennings. *LabVIEW Graphical Programming*. McGraw-Hill Professional, 2001.

[41] M. Kahn and C. Della Torre Cicalese. Coabs grid scalability experiments. *Autonomous Agents and Multi-Agent Systems*, 7(1-2):171–178, 2003.

[42] M. Kamvar and S. Baluja. A large scale study of wireless search behavior: Google mobile search. In *CHI '06: Proceedings of the SIGCHI conference on Human Factors in computing systems*, pages 701–709, New York, NY, USA, 2006. ACM Press.

[43] G. Kappel, S. Rausch-Schott, and W. Retschitzegger. A framework for workflow management systems based on objects, rules and roles. *ACM Comput. Surv.*, 32(1es):27, 2000.

[44] K. Konolige, K. Myers, E. Ruspini, and A. Saffiotti. The Saphira architecture: A design for autonomy. *Journal of experimental & theoretical artificial intelligence: JETAI*, 9(1):215–235, 1997.

[45] J. Laird, A. Newell, and P. Rosenbloom. Soar: an architecture for general intelligence. *Artif. Intell.*, 33(1):1–64, 1987.

[46] P. Langley, K. McKusick, J. Allen, W. Iba, and K. Thompson. A design for the icarus architecture. *SIGART Bull.*, 2(4):104–109, 1991.

[47] V.I. Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady*, 10:707–+, February 1966.

[48] Selker T. Lieberman, H. Agents for the user interface. 2002.

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

[50] G. Marti, V. Bettschart, J-S Billiard, and C. Baur. Hybrid method for both calibration and registration of an endoscope with an active optical tracker. In *CARS*, pages 159–164, 2004.

[51] D. McCarthy and U. Dayal. The architecture of an active database management system. In *SIGMOD '89: Proceedings of the 1989 ACM SIGMOD international conference on Management of data*, pages 215–224, New York, NY, USA, 1989. ACM Press.

[52] M. McTear. Intelligent interface technology: from theory to reality? *Interacting with Computers*, 12(4):323–336, 2000.

[53] S. Middleton. Interface agents: A review of the field. *ArXiv Computer Science e-prints*, March 2002.

[54] N. Milanovic and M. Malek. Current solutions for web service composition. *IEEE Internet Computing*, 8(6):51–59, 2004.

[55] M. Minsky. *Society of Mind.* Simon & Schuster, March 1988.

[56] T.M. Mitchell, R. Caruana, D. Freitag, J. McDermott, and D. Zabowski. Experience with a learning personal assistant. *Commun. ACM*, 37(7):80–91, 1994.

[57] P. Modi, M. Veloso, S. Smith, and J. Oh. Cmradar: A personal assistant agent for calendar management, 2004.

[58] D. Morley and K Myers. The spark agent framework. In *Proc. of the Third Int. Joint Conf. on Autonomous Agents and Multi Agent Systems (AAMAS-04)*, pages 712–719, New York, NY, July 2004.

[59] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.

[60] F.F-H. Nah. A study on tolerable waiting time: how long are web users willing to wait? *Behaviour and Information Technology*, 23:153–163(11), May-June 2004.

[61] A. Newell. *Unified theories of cognition*. Harvard University Press, Cambridge, MA, USA, 1990.

[62] J. Niekrasz, M. Purver, J. Dowding, and S. Peters. Ontology-based discourse understanding for a persistent meeting assistant, 2005.

[63] G. Papamarkos, A. Poulovassilis, and P. Wood. Event-condition-action rule languages for the semantic web, 2003.

[64] A. Rao and M. Georgeff. Modeling rational agents with a bdi-architecture. pages 317–328, 1998.

[65] C. Rich and C. Sidner. Collagen: A collaboration manager for software interface agents. *User Modeling and User-Adapted Interaction*, 8(3-4):315–350, 1998.

[66] F. Rosenberg and S. Dustdar. Business Rule Integration in BPEL – A Service-Oriented Approach. In *Proceedings of the 7th International IEEE Conference on E-Commerce Technology (CEC 2005)*, 2005.

[67] V. Roto and A. Oulasvirta. Need for non-visual feedback with long response times in mobile hci. In *WWW '05: Special interest tracks and posters of the 14th international conference on World Wide Web*, pages 775–781, New York, NY, USA, 2005. ACM Press.

[68] L. Rudolph. Project oxygen: Pervasive, human-centric computing - an initial experience. In *CAiSE '01: Proceedings of the 13th International Conference on Advanced Information Systems Engineering*, pages 1–12, London, UK, 2001. Springer-Verlag.

[69] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Englewood Cliffs, NJ, 2nd edition edition, 2003.

[70] D. Saha and A. Mukherjee. Pervasive computing: A paradigm for the 21st century. *Computer*, 36(3):25–31, 2003.

[71] M. Satyanarayanan and et al. Pervasive computing: Vision and challenges, 2001.

[72] B. Shneiderman and P. Maes. Direct manipulation vs. interface agents. *interactions*, 4(6):42–61, 1997.

[73] D. H. Stefanov, Z. Bien, and W.-C. Bang. The smart house for older persons and persons with physical disabilities: Structure, technology arrangements, and perspectives. *IEEE TRANSACTIONS ON NEURAL SYSTEMS AND REHABILITATION ENGINEERING*, 12(2):228–250, 2004.

[74] K. Sycara, M. Paolucci, M. Van Velsen, and J. Giampapa. The retsina mas infrastructure. *Autonomous Agents and Multi-Agent Systems*, 7(1-2):29–48, 2003.

[75] A. Tafat, M. Courant, and B. Hirsbrunner. Implicit environment-based coordination in pervasive computing. In *SAC '05: Proceedings of the 2005 ACM symposium on Applied computing*, pages 457–461, New York, NY, USA, 2005. ACM Press.

[76] W.M.P. Van der Aalst. The Application of Petri Nets to Workflow Management. *The Journal of Circuits, Systems and Computers*, 8(1):21–66, 1998.

[77] W.M.P. Van Der Aalst, A.H.M. Ter Hofstede, B. Kiepuszewski, and A.P. Barros. Workflow patterns. *Distrib. Parallel Databases*, 14(1):5–51, 2003.

[78] S. Vinoski. CORBA: integrating diverse applications within distributed heterogeneous environments. *IEEE Communications Magazine*, 14(2), 1997.

[79] W. Wahlster, N. Reithinger, and A. Blocher. Smartkom: multimodal communication with a life-like character. *In EUROSPEECH-2001*, pages 1547–1550, 2001.

[80] N. Ward and W. Tsukahara. A study in responsiveness in spoken dialog. *Int. J. Hum.-Comput. Stud.*, 59(5):603–630, 2003.

[81] J. Weizenbaum. Eliza, a computer program for the study of natural language communication between man and machine. *Commun. ACM*, 9(1):36–45, 1966.

[82] D. Wilkins and K. Myers. Asynchronous dynamic replanning in a multiagent planning architecture. In A. Tate, editor, *Advanced Planning Technology: Technological Achievements of the ARPA/Rome Laboratory Planning Initiative*, pages 267–274, AAAI Press, Menlo Park, CA, 1996.

[83] M. Willey. Design and implementation of a stroke interface library, 1997.

[84] M. Winikoff, L. Padgham, and J. Harland. Simplifying the development of intelligent agents. In *AI '01: Proceedings of the 14th Australian Joint Conference on Artificial Intelligence*, pages 557–568, London, UK, 2001. Springer-Verlag.

[85] W. Wobcke, V. Ho, A. Nguyen, and A. Krzywicki. A bdi agent architecture for dialogue modelling and coordination in a smart personal assistant. In *MMUI '05: Proceedings of the 2005 NICTA-HCSNet Multimodal User Interaction Workshop*, pages 61–66, Darlinghurst, Australia, Australia, 2006. Australian Computer Society, Inc.

[86] J. Yang, W. Yang, M. Denecke, and A. Waibel. Smart sight: A tourist assistant system. In *ISWC '99: Proceedings of the 3rd IEEE International Symposium on Wearable Computers*, page 73, Washington, DC, USA, 1999. IEEE Computer Society.

[87] J. Yue Chai, M. Budzikowska, V. Horvath, N. Nicolov, N. Kambhatla, and W. Zadrozny. Natural language sales assistant - a web-based dialog system for online sales. In *Proceedings of the Thirteenth Conference on Innovative Applications of Artificial Intelligence Conference*, pages 19–26. AAAI Press, 2001.

Glossary

AAA Security AAA is a framework providing three independent security functions: authentication, authorization and accounting.

Active Extension Server-side pre-compiled functionalities that can be called from Active rule code snippets.

AI Artificial Intelligence

API Application Programming Interface

POP Post Office Protocol. An application-layer Internet standard protocol, to retrieve e-mail from a remote server.

SOA Service Oriented Architecture

Corpus Set of reference utterances used to validate linguistic rules

CT Computer Tomographic imaging. In medical imaging, technique to construct 3D information out of 2D section images.

DARPA Defense Advanced Research Projects Agency

Dynamic programming Programming tehcnique designed to solve a problem by caching subproblem solutions rather than recomputing them.

EBNF Extended Backus Naur Form. A notation to express context-free grammars.

EBNF Extended Backus Naur Form. A syntax used to express formal grammars.

EJB Enterprise Java Bean. Managed server side Java-based component.

GPS Global Positioning System

HCI Human Computer Interaction

HCI Human Computer Interaction

IDE Integrated Development Environment

JDBC JDBC is an API for the Java programming language that defines how to access a database.

PDA Personal Digital Assistant, an electronic device which can include some of the functionality of a computer, a cellphone, a music player and a camera

RDBMS Relational Database Management System. Refers to a database management system that offers a reliable persistence machnism.

REST Representational State Transfer. Lightweight HTTP-based communication protocol. Requests are expressed through url attributes and results as XML documents.

RPC Remote Procedure Call

SDK Software Developer Kit

SDK Software Development Kit

snippets Snippet is a programming term for a small region of re-usable source code or text.

SOAP Simple Object Access Protocol

Utterance An utterance is one or more words, making complete unit in a given spoken language.

Web Scraper Web scraping is the action of extracting content from a website

WSDL Web Service Description Language. Provies all necessary information to invoke a SOAP compliant web service.

XML-RPC XML-RPC is a remote procedure call protocol which uses XML to encode its calls and HTTP as a transport mechanism.

XSLT Extensible Stylesheet Language Transformations. An XML-based language used for the transformation of XML documents into other XML-documents.

Appendix A : United States Patent Application

US 20070100790A1

(19) **United States**

(12) **Patent Application Publication** (10) **Pub. No.: US 2007/0100790 A1**

Cheyer et al. (43) **Pub. Date: May 3, 2007**

(54) **METHOD AND APPARATUS FOR BUILDING AN INTELLIGENT AUTOMATED ASSISTANT**

(76) Inventors: **Adam Cheyer**, Oakland, CA (US); **Didier Guzzoni**, St-Prex (CH)

Correspondence Address:
PATTERSON & SHERIDAN, LLP
SRI INTERNATIONAL
595 SHREWSBURY AVENUE
SUITE 100
SHREWSBURY, NJ 07702 (US)

(21) Appl. No.: **11/518,292**

(22) Filed: **Sep. 8, 2006**

Related U.S. Application Data

(60) Provisional application No. 60/715,324, filed on Sep. 8, 2005.

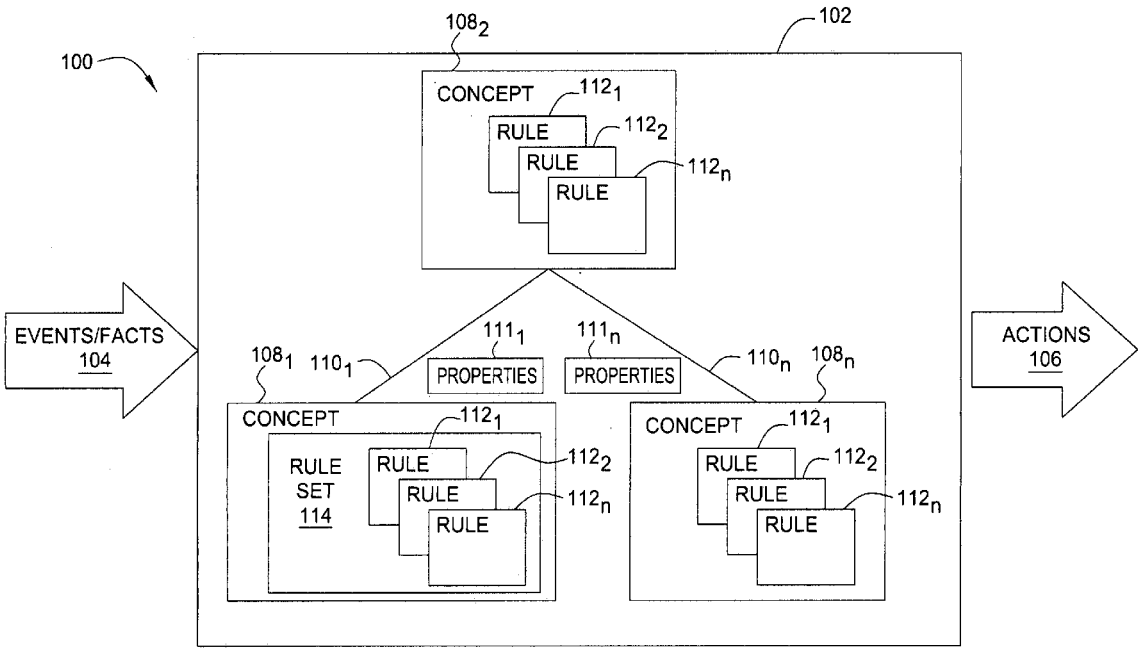
Publication Classification

(51) **Int. Cl.**
G06F 17/30 (2006.01)

(52) **U.S. Cl.** **707/1**

(57) **ABSTRACT**

A method and apparatus are provided for building an intelligent automated assistant. Embodiments of the present invention rely on the concept of "active ontologies" (e.g., execution environments constructed in an ontology-like manner) to build and run applications for use by intelligent automated assistants. In one specific embodiment, a method for building an automated assistant includes interfacing a service-oriented architecture that includes a plurality of remote services to an active ontology, where the active ontology includes at least one active processing element that models a domain. At least one of the remote services is then registered for use in the domain.



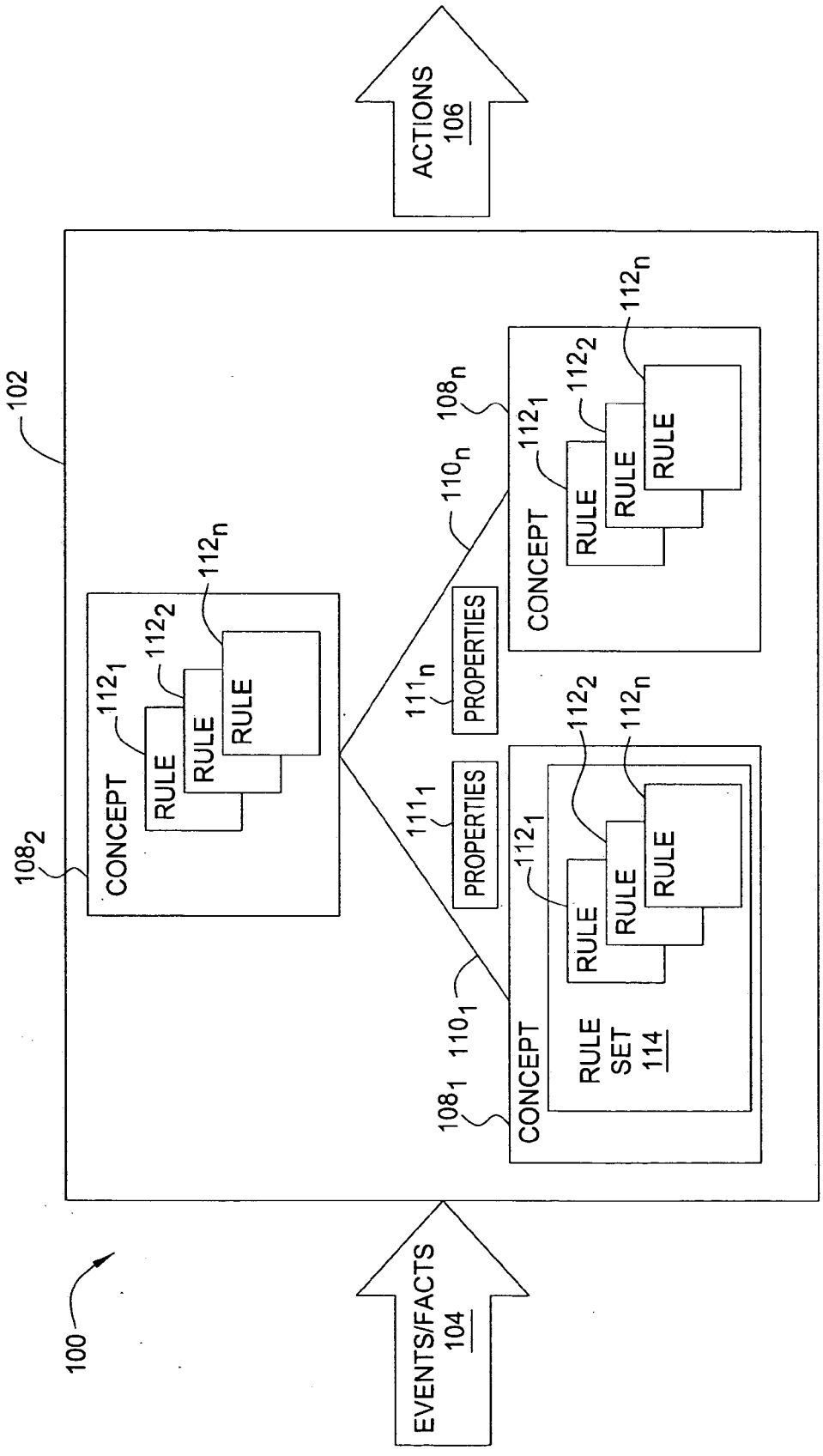

ACTIONS
106
102
CONCEPT
RULE
RULE
RULE
112_1
112_2
112_n
108_2
111_n
PROPERTIES
110_n
108_n
CONCEPT
RULE
RULE
RULE
112_1
112_2
112_n
111_1
PROPERTIES
110_1
108_1
CONCEPT
RULE
RULE
RULE
RULE SET
114
112_1
112_2
112_n
100
EVENTS/FACTS
104

FIG. 1

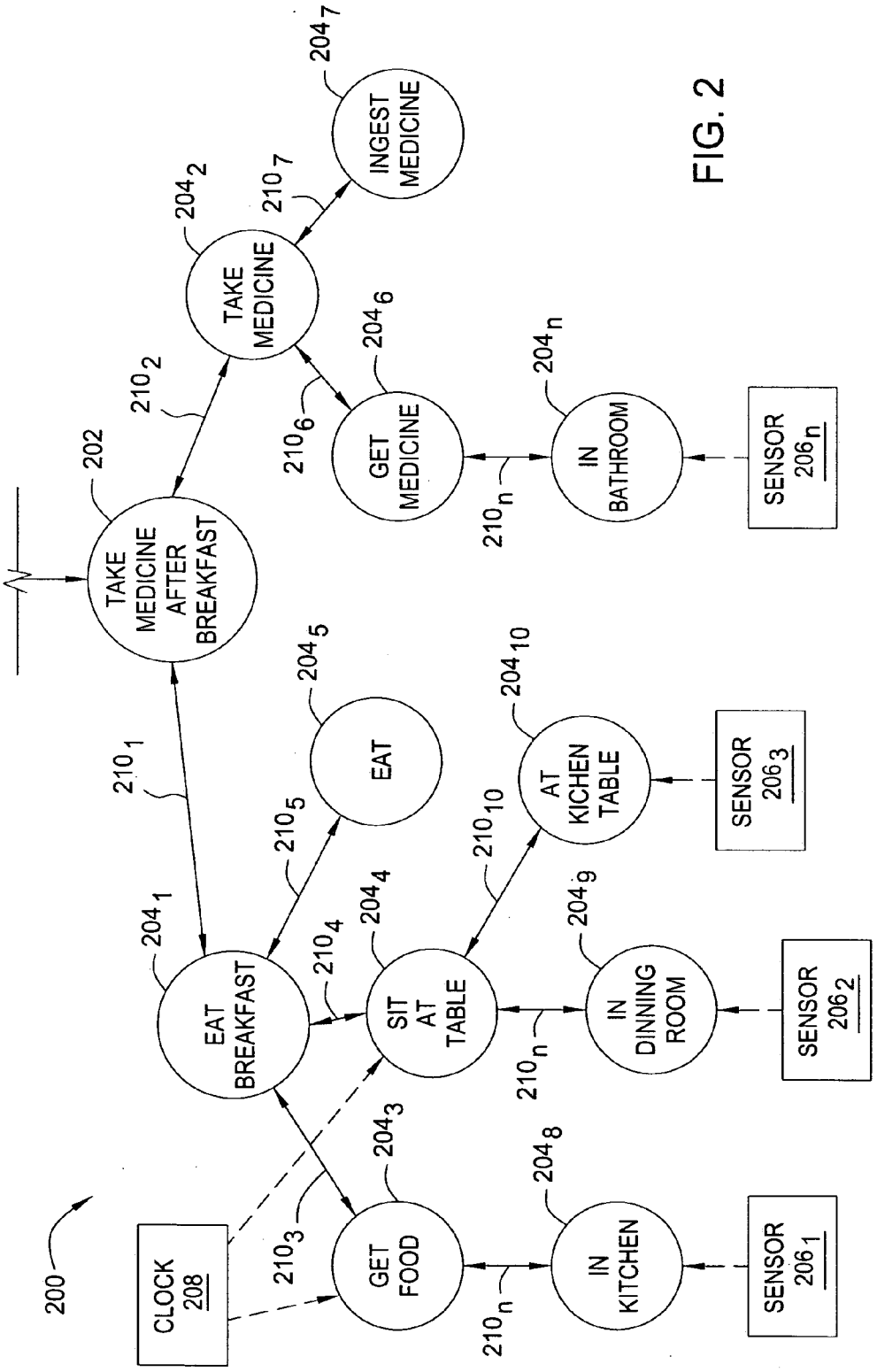

200
CLOCK 208
TAKE MEDICINE AFTER BREAKFAST
202
EAT BREAKFAST
204_1
SIT AT TABLE
204_4
GET FOOD
204_3
EAT
204_5
TAKE MEDICINE
204_2
GET MEDICINE
204_6
INGEST MEDICINE
204_7
IN KITCHEN
204_8
IN DINNING ROOM
204_9
AT KICHEN TABLE
204_{10}
IN BATHROOM
204_n
210_1
210_2
210_3
210_4
210_5
210_6
210_7
210_{10}
210_n
SENSOR 206_1
SENSOR 206_2
SENSOR 206_3
SENSOR 206_n

FIG. 2

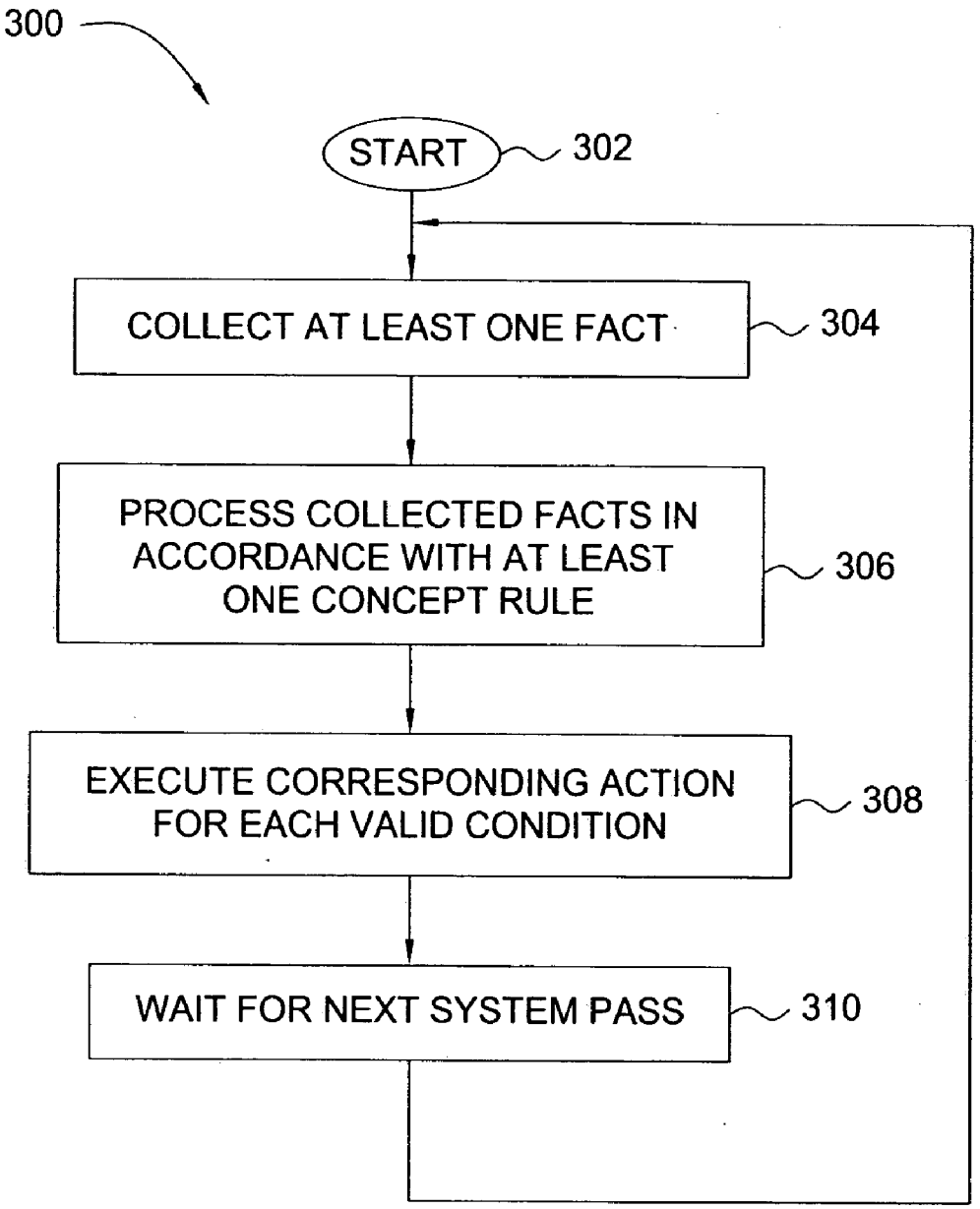

300
START
302
COLLECT AT LEAST ONE FACT
304
PROCESS COLLECTED FACTS IN ACCORDANCE WITH AT LEAST ONE CONCEPT RULE
306
EXECUTE CORRESPONDING ACTION FOR EACH VALID CONDITION
308
WAIT FOR NEXT SYSTEM PASS
310

FIG. 3

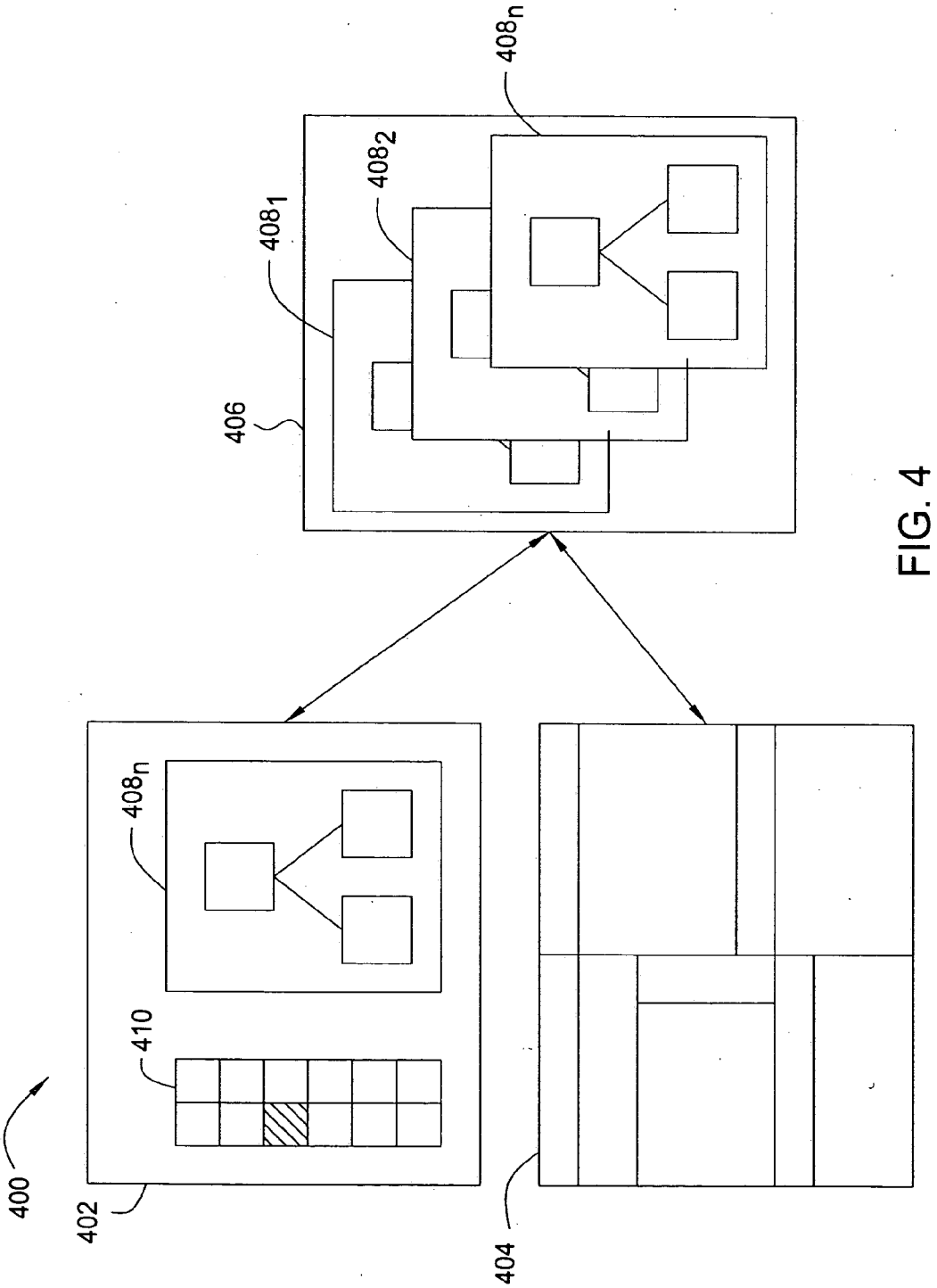


FIG. 4

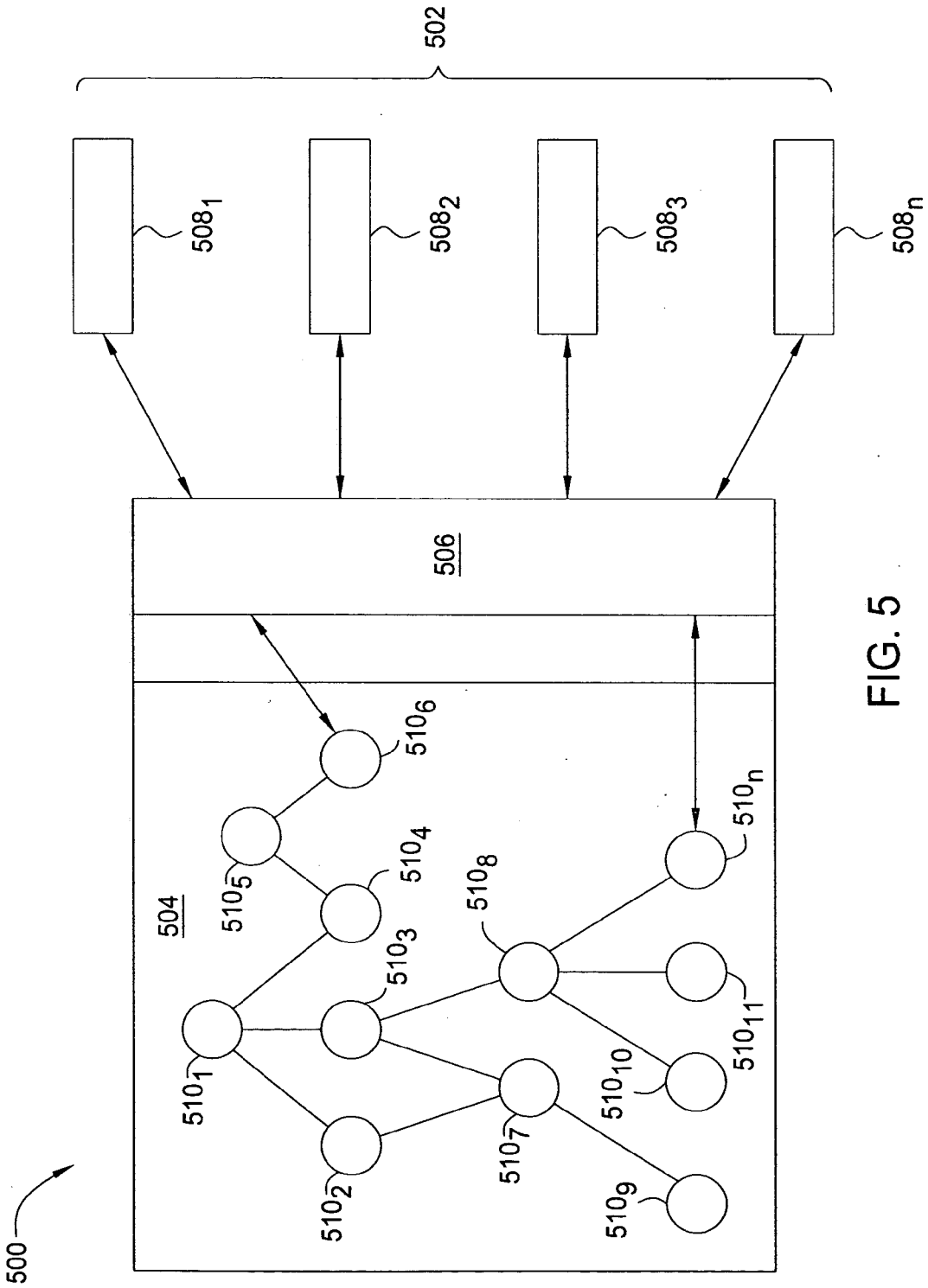

502
508_1
508_2
508_3
508_n
506
504
510_6
510_5
510_4
510_3
510_8
510_n
510_1
510_2
510_7
510_{10}
510_{11}
510_9
500

FIG. 5

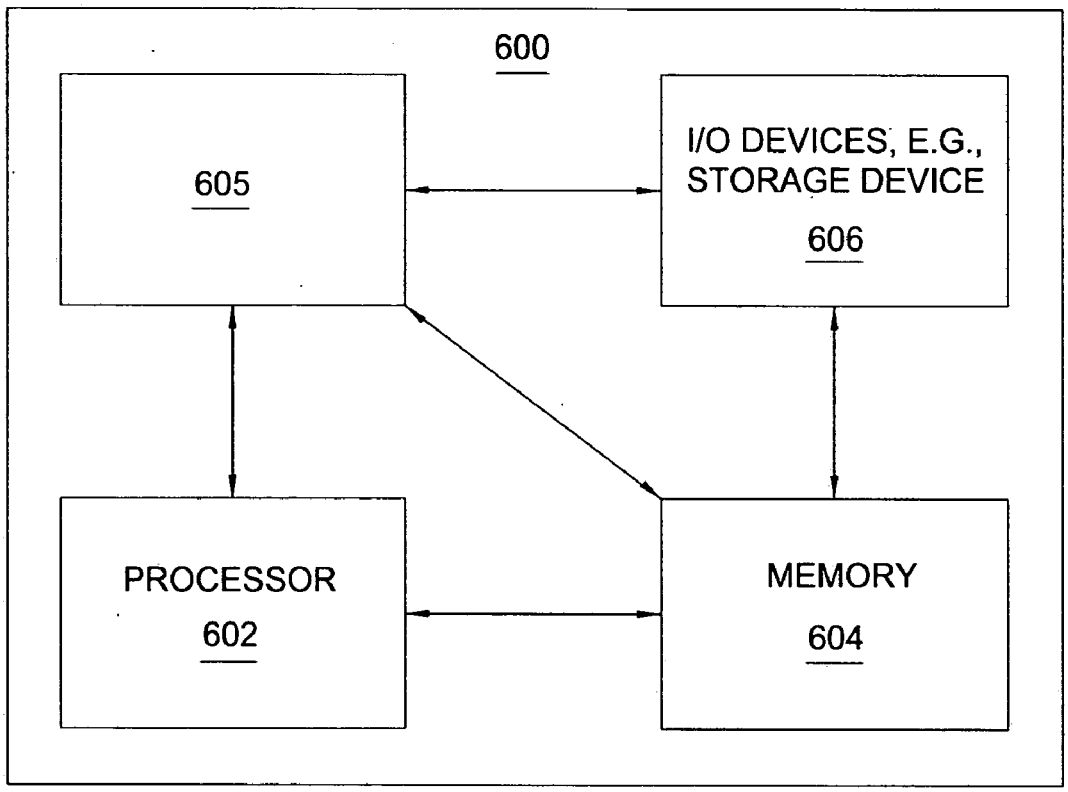

600
605
I/O DEVICES, E.G., STORAGE DEVICE
606
PROCESSOR
602
MEMORY
604

FIG. 6

METHOD AND APPARATUS FOR BUILDING AN INTELLIGENT AUTOMATED ASSISTANT

CROSS REFERENCE TO RELATED APPLICATIONS

[0001] This application claims the benefit of U.S. Provisional Patent Application Ser. No. 60/715,324, filed Sep. 8, 2005, which is herein incorporated by reference in its entirety.

FIELD OF THE INVENTION

[0002] The invention relates generally to intelligent systems and relates more specifically to tools for building classes of applications for intelligent automated assistants.

BACKGROUND OF THE DISCLOSURE

[0003] Intelligent systems, such as intelligent automated assistants, that are capable of interacting with humans (e.g., by observing user behavior, communicating with users, understanding observed situations, anticipating what a user may need and acting to produce useful behavior) are valuable in a variety of situations. For example, such systems may assist individuals who are impaired in some way (e.g., visually, auditorially, physically, cognitively, etc.), including elderly people (who may be stricken by one or more ailments), surgeons (whose eyes, hands and brains are constantly busy when performing operations) and business executives (who may have numerous tasks to accomplish), among others.

[0004] To accomplish all of these objectives, intelligent automated assistants integrate a variety of capabilities provided by different software components (e.g., for supporting natural language recognition, multimodal input, managing distributed services, etc.). Development of a system incorporating these different software components typically requires knowledge of numerous different programming languages and artificial intelligence-based methods. Thus, the development of an intelligent automated assistant is a complex task that typically requires contribution from a plurality of highly skilled individuals each having expertise in different aspects of programming; it is nearly impossible for a lone software developer to build an intelligent automated assistant due to the breadth and variety of expertise that is required to build the system. The typical development process for building intelligent automated assistants is therefore relatively inefficient in terms of time, cost and manpower.

[0005] Thus, there is a need in the art for a method and apparatus for building an intelligent automated assistant.

SUMMARY OF THE INVENTION

[0006] A method and apparatus are provided for building an intelligent automated assistant. Embodiments of the present invention rely on the concept of "active ontologies" (e.g., execution environments constructed in an ontology-like manner) to build and run applications for use by intelligent automated assistants. In one specific embodiment, a method for building an automated assistant includes interfacing a service-oriented architecture that includes a plurality of remote services to an active ontology, where the active ontology includes at least one active processing element that models a domain. At least one of the remote services is then registered for use in the domain.

BRIEF DESCRIPTION OF THE DRAWINGS

[0007] FIG. **1** is a schematic diagram illustrating one embodiment of an active ontology execution environment according to the present invention;

[0008] FIG. **2** is a schematic diagram illustrating one embodiment of an exemplary active ontology that is configured as an autominder for reminding a user to take medicine after meals (e.g., configured for activity and/or time recognition);

[0009] FIG. **3** is a flow diagram illustrating one embodiment of a method for processing facts in accordance with an active ontology (e.g., configured in a manner similar to the active ontology of FIG. **1**) according to the present invention;

[0010] FIG. **4** is a schematic diagram illustrating one embodiment of an open standard-based system for developing and managing an intelligent system using active ontologies;

[0011] FIG. **5** is a schematic diagram illustrating one embodiment of a framework for dynamically registering and coordinating distributed services using active ontologies; and

[0012] FIG. **6** is a high level block diagram of the present intelligent system building method that is implemented using a general purpose computing device.

DETAILED DESCRIPTION

[0013] In one embodiment, the present invention is a method an apparatus for building an intelligent automated assistant. Embodiments of the present invention rely on a developer-friendly unified framework, referred to as an "active ontology", which integrates multiple system-building capabilities in a single tool. An "ontology", generally, is a passive data structure that represents domain knowledge, where distinct classes, attributes and relations among classes are defined. A separate engine may operate or reason on this data structure to produce certain results. Within the context of the present invention, an "active ontology" may be thought of as an execution environment in which distinct processing elements are arranged in an ontology-like manner (e.g., having distinct attributes and relations with other processing elements). These processing elements carry out at least some of the typical tasks of an intelligent automated assistant. Although described within the context of an intelligent automated assistant, it will be understood that the concepts of the present invention may be implemented in accordance with any application that involves interaction with software.

[0014] FIG. **1** is a schematic diagram illustrating one embodiment of an active ontology execution environment **100** according to the present invention. The execution environment comprises an active ontology **102** that is adapted to receive one or more input facts or events **104** and process these inputs **104** to produce useful actions **106**. In one embodiment, the active ontology **102** is tailored to a specific context. For example, the active ontology **102** may be adapted to remind a user to take medication after meals,

Appendix B : User Evaluation Sheet

Subject ID: Date:

ACTIVE USER STUDY INSTRUCTIONS

1 Introduction

This user study evaluates and compares three tools that facilitate online access to information and services for mobile users.

- *Google Mobile* – market leading mobile search tool
- *ASK Mobile* – mobile search tool with some "natural language" capabilities
- *ACTIVE Mobile* – A mobile search assistant

You will be asked to perform a sequence of tasks, with several of the tools being tested. For each tool, we will measure the time to completion, number of queries, accuracy of results and overall user satisfaction.

We'd also like you to answer two brief questionnaires, one before the exercise and one after.

This session is expected to take about 1 hour total.

2 ACTIVE Search Assistant

Typically, most search engines accept keyword inputs, such as "tom cruise", and they return links to web pages that contain those terms. Using the ACTIVE search assistant, you can use keywords if you like, but the system can also understand queries naturally expressed in English. For example, you could enter "give me a list of all flights from San Francisco to Boston", and ACTIVE will return a specific answer to your question, rather than providing links to other web pages that might contain the answer to your question. ACTIVE also maintains context, so if you ask for restaurants in Boston, you can ask for nearby flower shops and it will interpret this request as being relevant to Boston.

3 Scenario

The tasks you will be asked to do will take place within the context of the following scenario:

"You get a call from your boss asking you to fly to <a city> for an important customer meeting. A flight has been booked for you, but you still need to find a hotel and gather information about the trip."

Subject ID: Date:

Pre-evaluation questionnaire

1. How often do you use a search-engine?

Web	Never	1	2	3	4	5	Very often
Mobile	Never	1	2	3	4	5	Very often

Comments:

2. How would you rate the effectiveness of the search tools you use?

Web	not effective	1	2	3	4	5	very effective
Mobile	not effective	1	2	3	4	5	very effective

Comments:

3. What type of search-engine user are you?

In addition to keywords, advanced users use attributes such as site, format, AND/OR operators, quotes, etc…

Web	Naïve user	1	2	3	4	5	Advanced
Mobile	Naïve user	1	2	3	4	5	Advanced

Comments:

4. Which search tools do you use?

Web _____

Mobile _____

Subject ID: Date:

GOOGLE MOBILE TASKS (page 1)

Answer as many of these questions as you can in 25 minutes. You will likely not finish in this time frame (that's OK).

You will be traveling to San Diego on flight United 501. Please find the following information about your flight:

1. Flight status (e.g. delayed, on time, in the air): _____

2. Actual departure time: _____

Worried about what to pack, you need to find out about weather conditions in San Diego for the coming five days. Find the climate (e.g. sunny) and low temperature for each day.

3. Today's climate: _____ Low Temp: _____
   Tomorrow: _____ Low Temp: _____
   Next: _____ Low Temp: _____
   Next: _____ Low Temp: _____
   Next: _____ Low Temp: _____

Next, you need to find hotels. You like to exercise every morning so you'd like one with a fitness center, and you want internet connectivity in your room. Find two candidate hotels that meet both of these criteria

4. Hotel Name: _____
   Hotel Address: _____
   Room price: _____

5. Hotel Name: _____
   Hotel Address: _____
   Room price: _____

Being an Indian food fan, you are interested in trying out the *best* indian restaurant in town.

6. Indian Restaurant: _____
   Phone number: _____
   Evidence that it's good: _____

Subject ID: Date:

GOOGLE MOBILE TASKS (page 2)

You want to find a few things to do in the evenings while you're there. You like to laugh, so find:

7. Funniest comedy club: _____

8. Comedy movie to watch: _____
   Theater where it's playing: _____

You don't have much money in your pocket so you decide to find the closest ATM to where you are (downtown Menlo Park).

9. Name: _____
   Address: _____
   How far to downtown? _____

Finally, you know that your significant other is going to be unhappy with this unexpected trip. You decide to grab some flowers on your way to the airport to appease him or her.

10. Florist: _____
    Address: _____

Subject ID: Date:

ACTIVE MOBILE TASKS (page 1)

Answer as many of these questions as you can in 25 minutes. You will likely not finish in this time frame (that's OK).

You are traveling to San Diego on United flight 501. Please find the following information about your flight:

1. Flight status (e.g. delayed, on time, in the air): _____
2. Actual departure time: _____

Worried about what to pack, you need to find out about weather conditions in San Diego for the coming five days. Find the climate (e.g. sunny) and low temperature for each day.

3. Today's climate: _____ Low Temp: _____
   Tomorrow: _____ Low Temp: _____
   Next: _____ Low Temp: _____
   Next: _____ Low Temp: _____
   Next: _____ Low Temp: _____

Next, you need to find hotels. You like to exercise every morning so you'd like one with a fitness center, and you want internet connectivity in your room. Find two candidate hotels that meet both of these criteria

4. Hotel Name: _____
   Hotel Address: _____
   Room price: _____

5. Hotel Name: _____
   Hotel Address: _____
   Room price: _____

Being an Indian food fan, you are interested in trying out the *best* indian restaurant in town.

6. Indian Restaurant: _____
   Phone number: _____
   Evidence that it's good: _____

Subject ID: Date:

ACTIVE MOBILE TASKS (page 2)

You want to find a few things to do in the evenings while you're there. You like to laugh, so find:

7. Funniest comedy club: _____

8. Comedy movie to watch: _____
   Theater where it's playing: _____

You don't have much money in your pocket so you decide to find the closest ATM to where you are (downtown Menlo Park).

9. Name: _____
   Address: _____
   How far to downtown? _____

Finally, you know that your significant other is going to be unhappy with this unexpected trip. You decide to grab some flowers on your way to the airport to appease him or her.

10. Florist: _____
    Address: _____

Subject ID: Date:

Post-evaluation questionnaire

1. Overall, how would you rate the GOOGLE-based mobile search tool?

very bad	**1**	**2**	**3**	**4**	**5**	**very good**

Comments:

2. Overall, how would you rate the ASK-based mobile search tool?

very bad	**1**	**2**	**3**	**4**	**5**	**very good**

Comments:

3. Overall, how would you rate the ACTIVE-based mobile search tool?

very bad	**1**	**2**	**3**	**4**	**5**	**very good**

Comments:

4. Do you have additional suggestions for how we might improve ACTIVE?

Thank you for completing the ACTIVE evaluation

Appendix C : Programmer Evaluation Tutorial

NLClassifiedsTutorial

From VRAI Group

Contents

[hide]

[edit]

Introduction

This tutorial is a practical guide to show how to use the Active software suite to create a language processing application. As an example, we create a simple system able to parse utterances related to classifieds (cars and real estate). Starting from an empty Active Ontology, each step of the tutorial introduces new concepts and how to use Active to implement, deploy and test them.

[edit]

Install and run required Active Components

The first step is to get and run basic Active software components: Active Server and Active Editor. This step is described here. To verify that both the Active Server and the Active Editor are running, make sure the connection indicator located on the lower right corner of the Active Editor window is green. In addition, the name of the Active Server with which the connection is established is shown on the title bar of the Active Editor. If the indicator is red, the trouble shooting section provides more information.

To help build language processing applications with Active, a set of basic system Active Ontologies need to be deployed on your Active Server. To do so, please execute the following steps:

- From the Active Editor, load and deploy the following Active Ontologies :
    - `ACTIVE_HOME/ontologies/tutorial/javascript/Delegation.xml`
    - `ACTIVE_HOME/ontologies/tutorial/javascript/NLCompanion.xml`
- Close both Active Ontologies tabs

[edit]

Create a new Active Ontology

In this step, we will create the Active Ontology in charge of language processing.

[edit]

Ontology creation

The following steps describe how to create our Active Ontology: (See Hello world! tutorial for details)

- Create a new blank Active Ontology
- Name the ontology : **Classifieds**
- Save it under : `classifieds.xml`

[edit]

Insert basic language processing concepts

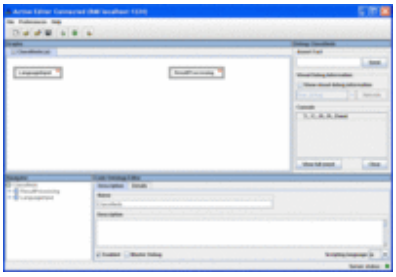


Figure 1

Before starting modeling our application, we need to insert the basic elements to allow our Active Ontology to perform language processing. Two generic processing nodes need to be inserted.

- Right click anywhere on the *Graphs* pane. Select : *Insert Wizard -> Language Processing -> Insert NL Context*
- Leave all fields untouched, click on **Finish**
- Right click anywhere on the *Graphs* pane. Select : *Insert Wizard -> Language Processing -> Insert NL Root*
- Leave all fields untouched, click on **Finish**

Two nodes (or Concepts) have now appeared in the graphs area of the Active Editor: *ResultProcessing* and *LanguageInput*. These two concepts provide processing rules to be inserted to any Active Ontology aimed at providing language processing. (*LanguageInput* provides basic rules to process incoming utterances and *ResultProcessing* contains the logic to manage parsing results). At the end of this step, your Active Ontology should look like **figure 1**. Concepts can be graphically arranged using drag-on-drop mouse operations.

[edit]

Start modeling the application domain

In this step we create the first basic structure of our language processing application. To model the domain of our application, we are going to graphically construct the structures and relationships that map natural language utterances into structured commands to be executed by the system. Our applications model is an upside-down tree-like structure made out of connected nodes. Terminal nodes on the bottom of the structure are *leaf* nodes, non terminal nodes are either *gather* or *choose* nodes. When utterances are submitted to the system for processing, each word is injected to the tree-like structure from its bottom terminal leaf nodes. When presented with a word, leaf nodes use some logic (the value of the word, its position in the sentence, etc…) to produce and communicate reports to their parent nodes. In the context of our tutorial, we are going to parse utterances about cars and real estate. A leaf node representing a car model will react to words such as "honda" or "ford", whereas a leaf node in charge of detecting the home types would trigger with "single family home" or "apartment". Typically, *leaf* nodes are connected to *gather* nodes, in charge of collecting reports coming from their children to create an aggregated message to be reported to their own parents. Through this bottom-up sequence, when the top of the tree is reached, a structured object mapping the natural language request is created.

Let\'s illustrate these concepts through our example. Since we are going to express queries about cars, we need to create a *car* object with two simple attributes: *model* and *color*.

[edit]

Create the *car* gather node

First, we need to create the car parent node:

- Right click anywhere on the *Graphs* pane. Select : *Insert Wizard -> Language Processing -> Insert NL Node*
- In the **name** field, enter `car`
- From the **type** combo box, choose **gather**
- Click **Next**, then click **Finish**

[edit]

Create the *model* leaf node

Figure 2

Next, we need to create a leaf node in charge of detecting car models. The leaf will use a vocabulary list to create its reports. To add the leaf node to our ontology, execute the following steps:

- Right click anywhere on the *Graphs* pane. Select : *Insert Wizard -> Language Processing -> Insert NL Leaf*
- In the **name** field, enter `model`
- Click **Next**
- In the **Values** section, click **Add**. Enter `accura` in the popup dialog.
- To add a synonym, click the **+** button in the word editor section. Enter `honda accura` in the popup dialog.
- Repeat the previous operation with any known user. (`prius`, `impreza`)
- Click **Finish**

[edit]

Create the *color* leaf node

We need to create a second leaf node in charge of detecting colors. Similarly to the **model** leaf node, we will use the **NL Leaf** Active Editor wizard.

- Right click anywhere on the *Graphs* pane. Select : *Insert Wizard -> Language Processing -> Insert NL Leaf*
- In the **name** field, enter `color`
- Click **Next**
- In the **Values** section, click **Add**. Enter `blue` in the popup dialog.
- Repeat the previous operation with `red`, `silver` and `green`.
- Click **Finish**

Note : This simple example shows how to locally store (in the Active Server fact store) the vocabulary attached to a leaf. Optionally, the vocabulary can be stored externally in a database or a flat file. Click here to get instructions about external vocabulary definition.

[edit]

Create structural relationships

We now have to define the relationships among these nodes. The leaf node *model* is a child of the structure defined by the *car* node. To connect *model* to *car*:

- Move your mouse to the center of the *model* node until the pointer becomes a hand.
- Left-click and hold. As you move your mouse, a line starting from the *car* node follows.
- While holding the button down, navigate over the *car* node until the pointer becomes a hand. Release the button to create the relationship.
- An arrow pointing from the *person* node to the *car* node appears.
- Proceed similarly to connect *color* to *car*.

Relationships have attributes that can be configured from the Active Editor. We are presently modeling a relation of type *is member of* between a child leaf node and the parent structure where is belongs. To configure our relationship:

- Left-click on the arrow to select a specific relationship. The arrow gets highlighted and the *code pane* area (bottom right) of the Active Editor indicates the attributes of the relationship to edit.
- In the **type** combobox select **is member of**
- The arrow turns green and the *code pane* show the specific attributes of this type of relationship.
- Proceed similarly to the relationship between *color* and *car*.

[edit]

Create root processing relationship

Figure 3

The last operation is to connect top node of our domain model (the *car* node) to the specialized *ResultProcessing* node inserted at step 1. This will allow our application to notify the user about the result of the processing. To do so, you need to connect the *car* node to the *ResultProcessing* concept using an *is a* (yellow) relationship.

[edit]

Deploy the sample Active Ontology

To reflect changes to Active Ontologies to the Active Server the new modified version needs to be re-deployed.

- To deploy an Active Ontology into the Active server, you need to click on the deploy toolbar icon  . This operation automatically re-generates the underlying rules, saves the Active Ontology to the local file system and deploys the new version to the Active Server. At the end of this step, your active ontology should look like **figure 3**.

[edit]

How to test the system

This step describes how to test our system by creating input utterances, monitoring the state of the parsing tree and analyze the final processing result.

[edit]

Run the NL-SwingConsole

Figure 4 : NL Swing console

The NL Swing console is a stand alone application used to create and send input utterances to an Active Ontology in charge of language processing. Here are the steps required to use it:

- Run the application : `ACTIVE_HOME/release/bin/nl-swing-console.bat`
- In the **Action** combobox, select **parse**

- In the **Target** combobox, select **Classifieds** (or the name you defined for your NL Active Ontology)
- The input section of the NL Swing console (at bottom of the main window), is where to type your utterance. For instance type `find a prius` followed by the **enter** key.

If all goes well, the following events should happen (see **figure 4**):

- The **session** field (top right part of the console) should be filled with a large number. This is the unique id of your session (dialogue) with the language processing Active Ontology you just created. We will come back to the notion of session later in the tutorial.
- The final result of the processing should have appeared in the main section of the console.

The parsed result shown in the console is made out of three parts. The parsed structure shown in the NL-SwingConsole shows:

```
RESULT
 PARSED
   car
     color(not_specified)
     model(prius)
 ERRORS([])
 SUGGESTIONS([color])
```

Under **PARSED** is the structured command produced by the Active Ontology. Under **ERRORS** and **SUGGESTIONS** two lists provide additional information about the parsing process. We will come back to these lists later in the tutorial. In our example, the resulting structure reflects our query describing a car that is a prius model. Note that there are no errors, and a suggestion informs that the color can optionally be specified.

[edit]

Monitor the parsing tree

Figure 5

From the Active Editor it is possible to monitor the current state of the parsing tree. To do so execute the following steps:

- In the Active Editor, check the "Show visual debug information" of the *debug panel* (top right section).
- From the combobox located just below, select "nl_debug"
- Click the "refresh" button

If all goes well, nodes should become highlighted with colors. (see figure 5) All the nodes of the parse tree are colored based on the confidence of their reports. For high scores, the node gets more and more green. For low scores, nodes get more red. In our case, the model node is green because it claims "honda", and its parent node (*car*) is also green because it reported positive report information from one of its children. On the other end, the color node is red because it could not detect anything relevant for its rules.

[edit]

Second utterance

Figure 6

To follow up on what the suggestion list contains, we can now specify price information:

- From the NL-SwingConsole enter : `red`
- Refresh the parsing tree status in the Active Editor

The parsed structure shown in the NL-SwingConsole shows:

```
RESULT
 PARSED
   car
     color(red)
     model(prius)
 ERRORS([])
 SUGGESTIONS([])
```

Now both slots *model* and *color* are filled and the suggestion list is empty.

[edit]

Populate the *car* node

In this section we will see how the **car** node can be augmented more children: **mileage** and **year**.

[edit]

Create the *mileage* leaf nodes

Figure 7

Next, we need to create a leaf node in charge of detecting the mileage of the car. Unlike potential colors, the mileage could be anything and cannot restricted to a finite vocabulary set. The strategy used by the leaf to create ratings uses a suffix. When the word *miles* is detected, the node assumes that preceding words represent the car mileage. For instance, the utterance "100K miles" will trigger a report from the *mileage* leaf stating that "100K" is a mileage. To add the mileage leaf node to our ontology, execute the following steps:

- Right click anywhere on the *Graphs* pane. Select : *Insert Wizard -> Language Processing -> Insert NL Leaf*
- In the **name** field, enter **mileage**
- In the **Check neighbours** section (bottom), check **Enable**
- Select **Check after (suffix)**
- In the text area enter `miles`
- Click **Finish**

Figure 7 shows how to create a suffix based policy.

[edit]

Create the *year* leaf node

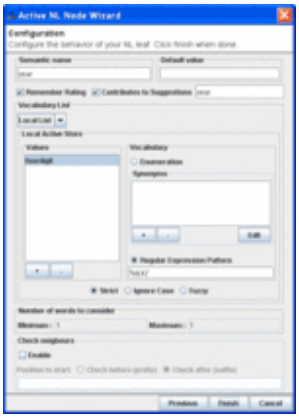


Figure 8

The leaf node in charge to detecting the manufacture year of the car is a four-digit number and cannot be expressed either by a simple vocabulary set nor a suffix-based technique. To detect numbers, (or any structured string) we will use a regular expression. For instance, our year leaf might define a word with value “long” and a Regular Expression Pattern that detects four-digit numbers : \\d{4}. When recognized, the Matching Expression is returned as the value for the parse expression, not the Word Value. To add the year leaf node to our ontology, execute the following steps:

- Right click anywhere on the *Graphs* pane. Select : *Insert Wizard -> Language Processing -> Insert NL Leaf*
- In the **name** field, enter `year`
- Click **Next**
- In the **Values** section, click **Add**. Enter `fourdigit` in the popup dialog.
- In the **Vocabulary** section, select **Regular Expression Pattern**.
- In the text field, enter `\\d{4}`.
- Click **Finish**

Figure 7 shows how to create a regular expression-based policy.

[edit]

Create children relationships

Similarly to previous leaf nodes, children need to be connected to their parent node. To do so, create the following relationships:

- From *mileage* to *car*, add a relationship of type **is member of**
- From *year* to *car*, add a relationship of type **is member of**

[edit]

Test the system

This step show how our application reacts to a sequence of input utterances about cars. (Details about interacting with our application can be found here)

[edit]

First utterance

Figure 9

To start, let's create an utterance about a car:

- To start a new dialog, from the NL-SwingConsole, click on **Reset Session**
- Enter : `find a 1997 red accura`
- Refresh the parsing tree status in the Active Editor

The parsed structure shown in the NL-SwingConsole shows:

```
RESULT
 PARSED
   car
     color(red)
     mileage(not_specified)
     model(accura)
     year(1997)
 ERRORS([])
 SUGGESTIONS([mileage])
```

The resulting structure reflects our query describing the car to look for.

[edit]

Second utterance

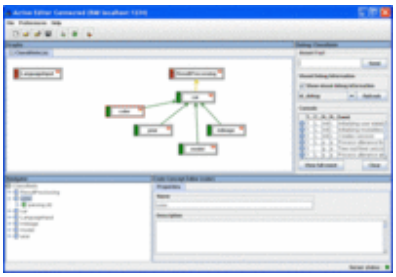


Figure 10

To follow the element of suggestion list, we could now specify price information:

- Enter : `about a 100K miles`
- Refresh the parsing tree status in the Active Editor

The parsed structure shown in the NL-SwingConsole shows:

```
RESULT
 PARSED
   car
     color(red)
     mileage(100K)
     model(accura)
     year(1997)
 ERRORS([])
 SUGGESTIONS([])
```

Note how only the mileage contribution to the parsed structure has changed.

[edit]

Add a second competing branch to the parsing tree

In this step, we will add a new branch in charge of parsing requests about real estate and see how it contributes to a more complex parsing tree. To model a domain about homes, we will create a structure similar to our existing car node. To keep the tutorial simple, we use a limited home definition of three parameters: a city, a list of amenities and a type.

[edit]

Create the *home* gather node

First, we need to create the *home* gather node:

- Right click anywhere on the *Graphs* pane. Select : *Insert Wizard -> Language Processing -> Insert NL Node*

- In the **name** field, enter `home`
- From the **type** combo box, choose **gather**
- Click **Next**, then click **Finish**

[edit]

Create the type node

Next, we need to create a leaf node in charge of detecting the type of home, using a small vocabulary set. To add the **home** leaf node to our ontology, execute the following steps:

- Right click anywhere on the *Graphs* pane. Select : *Insert Wizard -> Language Processing -> Insert NL Leaf*
- In the **name** field, enter `type`
- Click **Next**
- In the **Values** section, click **Add**. Enter `house` in the popup dialog. Next, we will add synonyms to the word house.
- To add a synonym, click the **+** button in the **word editor** section. Enter `"single family home"` in the popup dialog.
- Repeat the previous operation with `"apartment"`, `"studio"` and `"townhouse"`.
- Click **Finish**

[edit]

Create the amenities node

Next, we create a leaf node in charge specifying amenities to look for. The list of amenities to support for out examples are : harwood floor, garage, fireplace and pool. To add the amenities leaf node to our ontology, execute the following steps:

- Right click anywhere on the *Graphs* pane. Select : *Insert Wizard -> Language Processing -> Insert NL Leaf*
- In the **name** field, enter `amenities`
- Click **Next**
- In the **Values** section, click **Add**. Enter `hardwoodfloor` in the popup dialog. Next, we will add synonyms to detect the option.
- To add a synonym, click the **+** button in the **word editor** section. Enter `"hardwood floor"` in the popup dialog.
- Repeat the previous operation with `"garage"`, `"fireplace"` and `"pool"`.
- Click **Finish**

[edit]

Create the city node

A prefix-based leaf node to detect the city where to search for a home. For this tutorial, we will focus on cities located in the south of the San Francisco bay area. To add the city leaf node to our ontology, execute the following steps:

- Right click anywhere on the *Graphs* pane. Select : *Insert Wizard -> Language Processing -> Insert NL Leaf*
- In the **name** field, enter `city`
- Click **Next**
- In the **Values** section, click **Add**. Enter `palo alto` in the popup dialog.
- Repeat the previous operation with `"sunnyvale"`, `"menlo park"`, `"sunnyvale"` or any city you may be interested in.

the popup dialog.

- Click **Finish**

[edit]

Create children relationships

Similarly to the *car* structure, *home* related nodes need to be connected. To do so, create the following relationships:

- From *city* to *home*, add a relationship of type **is member of**
- From *type* to *home*, add a relationship of type **is member of**
- From *amenities* to *home*, add a relationship of type **is member of**

[edit]

Specify child cardinality

So far, we have used leaf nodes that produce a single value. For instance, a car has a unique color and a single make. Amenities are different, one may want to search for a home that has a pool *and* a garage. The Active-based language processor supports this by configuring **is member of** relations. To specify that we may want to create a list of possible amenities, execute the following steps:

- Right click on the **is member of** that connects *amenities* to *home*.
- In the *Code Relationship Editor* (lower right corner of the Active Editor) uncheck the **Is Single** option.

[edit]

Specify child contribution

Some child nodes may contribute relevant information to the structure created by the parents to which they are connected without contributing to their overall confidence weight during the selection ("chooser") process. For instance, the amenities node reacts to words such as *garage* or *pool* bringing details about the hotel to describe. However, if someone is looking for a swimming pool or a garage, it would be unfortunate to have the hotel node chosen because some of its amenities happen to match other points of interest. To prevent this situation, children can be configured not to contribute to their parent's selection rating, while providing structural information. This feature is also useful when a child leaf-node is shared by multiple parents. Since it will equally contribute to all its parents, it can be configured not to influence their selection rating, leaving more relevance to its siblings. To specify this option on *amenities*, execute the following steps:

- Right click on the **is member of** relation that connects *amenities* to *home*.
- In the *Code Relationship Editor* (lower right corner of the Active Editor) uncheck the **Contributes to weight** option.

In summary, if a leaf strongly contributes to the *selection* of its parent, "Contributes to weight" should be selected. If a leaf connects to more than one parent, or does not contain words that strongly influence whether their parent is selected or not, "Contributes to weight" should be unchecked.

[edit]

Create the *item* chooser node

Users are going to express requests to look for cars or homes. To choose among these two possible items, reports coming from the *car* node will be compared with reports coming out of the *home* node. This is performed by *chooser* nodes, whose task is to select the best option by using their children reports and relationship types. To add the *chooser* node, we create a new node named *item*, connected with its two children: the *car* and the *home* nodes. To add the *chooser* node:

- Right click anywhere on the *Graphs* pane. Select : *Insert Wizard -> Language Processing -> Insert NL Node*
- In the **name** field, enter `item`
- From the **type** combo box, choose **choose**
- Click **Next**, then click **Finish**

[edit]

Update the parsing tree

Figure 11

The *car* node is not the root of our parsing tree anymore, it needs to be disconnected from the top *ResultProcessing* node.

- Right-click over the **is a** relationship that connects the *car* node to the *ResultProcessing* node.
- Select the **delete** option.
- The relationship should disappear.

Next, we need to use relationships to create our new parsing tree and connect its new root to the top *ResultProcessing* node.

- From *car* to *item*, add a relationship of type **is a**
- From *home* to *item*, add a relationship of type **is a**
- From *item* to *ResultProcessing*, add a relationship of type **is a**

Finally, save and deploy your Active Ontology. At the end of this step, it should look like **figure 11**.

[edit]

Test the system

This step show how our application reacts to a sequence of input utterances about cars and homes. (Details about interacting with our application can be found here)

[edit]

First utterance

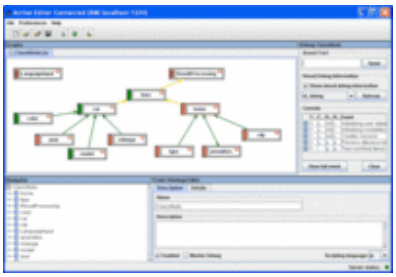


Figure 12

To start, let’s create an utterance about cars:

- To start a new dialog, from the NL-SwingConsole, click on **Reset Session**
- Enter : `blue accura`
- Refresh the parsing tree status in the Active Editor

The parsed structure shown in the NL-SwingConsole shows:

```
RESULT
 PARSED
   item
     car
       color(blue)
       mileage(not_specified)
       model(accura)
       year(not_specified)
 ERRORS([])
 SUGGESTIONS([mileage,year])
```

The resulting structure reflects our query describing information about a blue accura. Note that there are no errors, and a suggestion informs that the mileage and the year can be specified. On the Active Editor side, a green path starting from the car node shows which nodes have produced positive reports.

[edit]

Second utterance

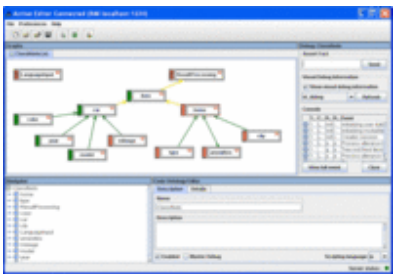


Figure 13

To follow the element of suggestion list, we could now specify year when the car was manufactured:

- Enter : `1999`
- Refresh the parsing tree status in the Active Editor

The parsed structure shown in the NL-SwingConsole shows:

```
RESULT
 PARSED
   item
     car
       color(blue)
       mileage(not_specified)
       model(accura)
       year(1999)
 ERRORS([])
 SUGGESTIONS([mileage])
```

Now slots *color*, *model* and *year* are filled and the suggestion list only mentions mileage.

[edit]

Third utterance

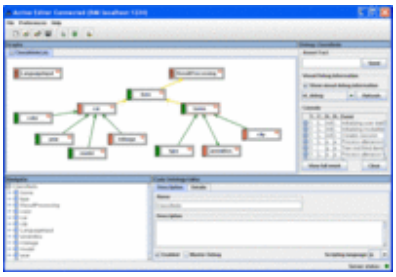


Figure 14

To see how the *item* chooser node works, let's enter an utterance related to a home:

- Enter : `single family home`
- Refresh the parsing tree status in the Active Editor

The parsed structure shown in the NL-SwingConsole is:

```
RESULT
 PARSED
   item
     home
       []
       city(not_specified)
       type('house')
 ERRORS([])
 SUGGESTIONS([city])
```

The parsed result now indicates a home-related activity, where the *type* slot is filled. The item node did its job: Since the user mentioned a word related to real estate, the home side of the parsing tree was given the advantage. To make a decision a chooser node (*item* in our case) uses two factors. As explained before, is uses the confidence of the reports coming out of its children. In addition, chooser nodes use the age of children reports. Each report is time-stamped with the time of its creation. When a chooser node has to select its best children, it lowers to score of incoming reports based on their timestamps. Older reports are penalized more than recent ones. This technique favors the side of the tree the users talked about most recently. In our case both children (*car* and *home*) have produced positive reports, their confidence may be equal. Since the report coming out of the *home* child is more recent than the one produced by the *car* child, *home* wins.

[edit]

Fourth utterance

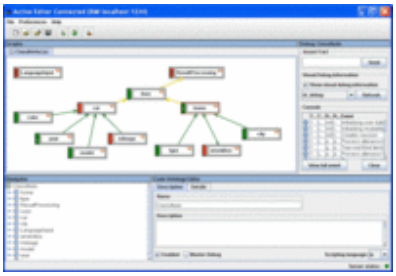


Figure 15

To show more of the aging process of competing nodes, let's add one more utterance related to the home side of our parsing tree:

- Enter : `in palo alto`
- Refresh the parsing tree status in the Active Editor

The parsed structure shown in the NL-SwingConsole is now:

```
RESULT
 PARSED
```

```
   item
     home
       []
       city('palo alto')
       type('house')
 ERRORS([])
 SUGGESTIONS([])
```

[edit]

Fifth utterance

Image:New-s10a-new.gif Now, let us specify amenities:

- Enter : `with a garage`
- Refresh the parsing tree status in the Active Editor

The parsed structure shown in the NL-SwingConsole is now:

```
RESULT
 PARSED
   item
     home
       [amenities(garage)]
       city('palo alto')
       type('house')
 ERRORS([])
 SUGGESTIONS([])
```

[edit]

Sixth utterance

To illustrate the the *amenities* node has a multiple cardinality, let us specify a second amenity:

- Enter : `and a pool`
- Refresh the parsing tree status in the Active Editor

The parsed structure shown in the NL-SwingConsole is now:

```
RESULT
 PARSED
   item
     home

[amenities(garage),amenities('pool')]
       city('palo alto')
       type('house')
 ERRORS([])
 SUGGESTIONS([])
```

Note that the whole process could have been expressed as a single utterance : *find a single family home in palo alto with a garage and a swimming pool*

[edit]

Add auxiliary information

This step introduces the notion of leaf nodes providing *auxiliary information*. In some cases, words expressed by users provide information about which part of the parsing tree should be given more weight, without contributing to the data structure to be created. For instance, an utterance containing the word "car" indicates the topic the user is talking about, without providing any specific information such as the year nor the model name. The word "car" should only bring more weight to the *car* node without being part of the generated data structure produced by the node. To implement this behavior, any leaf can be connected to a gather node using a *provide auxiliary information* relationship. Children connected as *provide auxiliary information* will contribute to the overall confidence and age index of their parent *gather* node in a similar way as their siblings connect as *is member* of children. The only difference between the two type of children is that *provide auxiliary information* leaf nodes will not be part of the structure created for the report of the parent *gather* node. To illustrate this concept, let's add two new leaf nodes to our sample ontology. They will be in charge of detecting the words "homes" and "cars".

[edit]

Create the helper leaf nodes

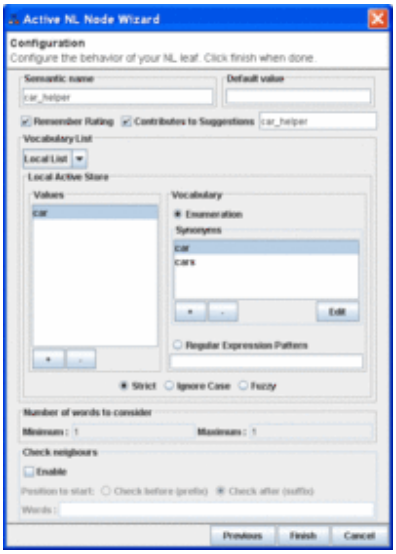


Figure 16

First, let's create a leaf node in charge of detecting "cars".

- Right click anywhere on the *Graphs* pane. Select : *Insert Wizard -> Language Processing -> Insert NL Leaf*
- In the **name** field, enter **car_helper**
- Click **Next**
- In the **Values** section, click **Add**. Enter `cars` in the popup dialog.
- To add a synonym, click the **+** button in the **word editor** section. Enter `car` in the popup dialog.
- Click **Finish**

Similarly to the "home_helper" leaf node, we will use the **NL Leaf** Active Editor wizard to create a leaf in charge of detecting the word "homes" or "home".

- Right click anywhere on the *Graphs* pane. Select : *Insert Wizard -> Language Processing -> Insert NL Leaf*
- In the **name** field, enter `home_helper`
- Click **Next**
- In the **Values** section, click **Add**. Enter `homes` in the popup dialog.
- To add a synonym, click the **+** button in the **word editor** section. Enter `home` in the popup dialog.
- Click **Finish**

[edit]

Update the parsing tree

Figure 17

The helper nodes providing auxiliary information need to be connected to the parent.

- From *car_helper* to *car*, add a relationship of type "provides auxiliary information"
- From *home_helper* to *home*, add a relationship of type "provides auxiliary information"

Finally, save and deploy your Active Ontology. At the end of this step, it should look like **figure 17**.

[edit]

Test the system

This step show how our application reacts to a sequence of input utterances about cars and appartments. (Details about interacting with our application can be found here)

[edit]

First utterance

Figure 18

To start, let's create an utterance about homes:

- To start a new dialog, from the NL-SwingConsole, click on **Reset Session**
- Enter : `find homes`
- Refresh the parsing tree status in the Active Editor

The parsed structure shown in the NL-SwingConsole shows:

```
RESULT
 PARSED
   item
     home
       []
       city(not_specified)
       type(not_specified)
 ERRORS([])
 SUGGESTIONS([city,type])
```

Even if no specific information about a home appears in the result created by the parsing tree, the home option has been picked.

[edit]

Second utterance

Figure 19

- Enter : `a car`
- Refresh the parsing tree status in the Active Editor

The parsed structure shown in the NL-SwingConsole shows:

```
RESULT
 PARSED
   item
     car
       color(not_specified)
       mileage(not_specified)
       model(not_specified)
       year(not_specified)
 ERRORS([])

SUGGESTIONS([color,model,mileage,year])
```

Similarly, even if no specific car related information was provided, the car side of the tree was picked because the *car_helper* node contributed to the overall rating of the *car* node.

[edit]

Third utterance

Figure 20

To swing back on the home side, let us provide rich utterance about real estate:

- Enter : `find an apartment with a fireplace in sunnyvale`

- Refresh the parsing tree status in the Active Editor

The parsed structure shown in the NL-SwingConsole shows:

```
RESULT
 PARSED
   item
     home
       [amenities(fireplace)]
       city(sunnyvale)
       type(apartment)
 ERRORS([])
 SUGGESTIONS([])
```

Back to main Active Wiki

Appendix D : Programmer Evaluation Sheet

Subject ID: Date:

ACTIVE PROGRAMMER STUDY INSTRUCTIONS

1 Introduction

This user study evaluates how programmers can, after a short training, model a language processing application using the Active framework and its extensions.

2 Goal of the evaluation

As you have experienced when going through our tutorial, a language processing application converts samples of human language (utterances) into more formal representations that are easier for computer programs to manipulate.

In this evaluation, you will be asked to construct a simple application in charge of parsing a set utterances. Utterances to support are related to the travel domain and are expressed in the following section.

3 Utterances to support

3.1 About weather

The user can express queries to get a weather forecast in any given city.

- *"get me the weather forecast in San Diego"*
- *"what's the weather in Seattle"*

3.2 About restaurants

The goal is to describe restaurants by their location and type of cuisine. A location is any city name and types of cuisine could be "french", "italian", "chinese", "japanese" or "indian".
Examples:

- "*find restaurants in Paris*"
- "*get me the best indian restaurant in Seattle*"

3.3 About flights

The goal is to inquire about the status of flights. Airlines and flight numbers can be specified. Airlines can be expressed either by their full name (i.e Air France) or code (i.e. AF). Possible airlines should include "Air France" ("AF"), "Lufthansa" ("LH"), "United" ("UA" or "United Airlines").
Flight numbers can be any two-digit, three-digit or four-digit flight number can be specified.
Examples:

- *"status flight air france 83"*
- *"is flight united 510 on time?"*

- *"united 1233"*

3.4 About movies

Users should also express utterances about movies, modeled by a genre ("comedies", "thrillers" or "action"), a location (city name) and a list of any actors.

- *"find thrillers in San Diego"*
- *"get movies with John Wayne"*

3.5 About hotels

Allow users to search for hotels in a city by one or more amenities:

- *"hotels with a pool and wifi"*
- *"I want a hotel in san diego with internet connectivity and a fitness center"*

3.6 Points of interest

Finally, users should express utterances about points of interest, simply defined by a name ("florist", "starbucks" or "dentist") and a location (city information).

- *"get me all starbucks in Paris"*
- *"find florists in San Diego"*
- *"nearby ATMs in Palo Alto"*

Subject ID: Date:

4 Post evaluation questionnaire

4.1 What are your software programming skills ?

weak	**1**	**2**	**3**	**4**	**5**	**strong**

Comments:

_____
_____
_____

4.2 Have you ever worked with/on a language processing software?

☐Yes ☐No

Comments:

_____
_____
_____

4.3 How would you rate the effectiveness of the system for creating language processing ?

not effective	**1**	**2**	**3**	**4**	**5**	**very effective**

Comments:

_____
_____
_____

Subject ID: Date:

4.4 Overall, how would you rate the Active system in general?

very bad	1	2	3	4	5	very good

Comments:

_____
_____
_____

4.5 Please suggest some improvements to the application

_____

_____

_____

_____

Thank you for completing the Active evaluation

Appendix E : Candidate CV

Didier Guzzoni
Address: 5 Chemin des Vergers, 1162 St-Prex, Switzerland
Tel: 021 8062758
Email : didier@gmail.com

Technical Skills

Industrial and academic experience in both Silicon Valley and Switzerland. J2EE specialist and early adopter of web service technologies. Strong interest and experience in software design, implementation, distributed architectures, web based applications, artificial intelligence and HCI.

Technical expertise: Java, J2EE, SOAP, WSDL, Web2 (AJAX/Javascript), C#, IIS, Corba, Swing, OpenGL

Work Experience

October 2003 – Present EPFL / Lausanne	**EPFL, Research assistant & Ph.D candidate**

Based on eight years of research and industry experience, entered a Ph.D program to focus on research in applied computer science. The goal is to ease the development and improve the performance of intelligent assistant software. The solution provides a unified approach for rapidly developing applications incorporating natural language interpretation, dialog management, multimodal fusion, adaptable presentation generation, reactive execution and dynamic brokering of web services.

May 2002 - April 2004 Cupertino/California	**Confluent Software (Oracle since March 2005)** **Senior Software Engineer**

Confluent Software is a leading provider of web services management solutions. Worked as lead engineer on core components of Confluent Software product line. As an early employee, made critical technical decisions that led the company to successfully design, implement and deploy J2EE based enterprise quality software to major financial and manufacturing firms. Confluent Software was acquired by Oracle in October 2005.

June 2000 - May 2002 Palo Alto/California	**VerticalNet Inc. / Advanced Products Group** **Research Engineer**

Significant contributor to the OSM® distributed software architecture for dynamic web services orchestration. OSM® was the first commercially available service oriented (SOAP) platform for electronic commerce applications.

March 98 - June 2000 Menlo Park/California	**SRI International / Artificial Intelligence Center** **Software Engineer**

Major contributor to the OAA® (www.ai.sri.com/~oaa) distributed software architecture, utilized in more than twenty research and industrial projects. Primary responsibilities consisted of maintaining and improving core OAA components (mostly Java, C, C++ and Visual Basic libraries), supporting users, and presenting the OAA® philosophy at international conferences.

August 96 - March 98 EPFL / Lausanne	**EPFL, Research assistant**

Involved in the VIRGY project (Virtual Surgery) aimed at developing a training system based on virtual reality and force feedback for endoscopic surgery. In charge of the design and implementation (C++, OpenGL) of a virtual reality application that is able to simulate human organs and surgery tools. This project was span off as a successful startup company.

Education

1996 EPFL (Swiss Federal Institute of Technology)	M.S. in Computer Science with highest marks
1991 Engineering School of Geneva	B.S. with highest honors in Electronic Engineering, specialization in computer science and telecommunications