

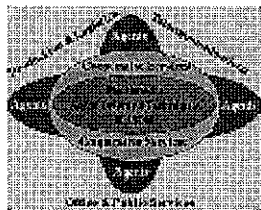
Building a Generic User Agent for Multi-agent Integrated Enterprise

Diploma Thesis

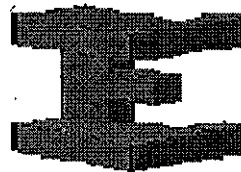
Raik Haase

Enterprise Integration Laboratory
University of Toronto
4 Taddle Creek Road, Rosebrugh Building
Toronto, Ontario M5S 1A4
haase@ie.utoronto.ca

December 1997



DAI Laboratory
Technical University of Berlin
Germany



Enterprise Integration Laboratory
University of Toronto
Canada

Abstract

Today, organizations are faced with permanently changing markets, global competition and rapidly decreasing cycles of technological innovation. The ability of an enterprise to achieve competitive advantages and to survive in such dynamic settings largely depends on organizational flexibility, availability of information and effective coordination of decisions and actions. Business processes need to be engineered and managed across geographical borders and beyond traditional functional barriers. One of the key concepts in modern enterprise operation is the efficient management of its supply chain. The supply chain is a world-wide network of suppliers, factories, distribution centers, warehouses and retailers through which raw material are acquired, transformed and delivered to the customer. In order to optimize performance, supply chain functions must operate in a tightly coordinated manner. However, the dynamics of the enterprise and the market require flexible responses and adaptations with local and global effects on supply chain entities. For implementing such distributed coordinated structures successfully, they need to be equipped with adequate information infrastructure that reflects and supports the organizational tendencies.

The agent view provides a level of abstraction at which we envisage computational systems carrying out cooperative work by interoperating globally across networks connecting people, organizations and machines towards a single virtual platform. The agent's ability to plan and to coordinate their actions in a goal-oriented, dynamical and flexible manner while solving problems autonomously from a local perspective reflects the distributed decision and execution processes in modern enterprises to the most extent. In viewing supply chain management as a distributed decision problem which requires the coordination of autonomous entities, multi-agent systems promise to tackle the issues at a sophisticated and realistic level. A major challenge in building such systems is to involve people interactively and intuitively in the agent's execution process by providing adequate interfaces. The objective is to adjust representation and control of agent systems to the expectations and experience of prospective end-users. The World Wide Web allows for business information systems where intelligent agents and humans cooperate in planning, decision making and execution of functions while the system ensures a coherent and coordinated behavior.

This thesis aims to contribute to the efforts of introducing multi-agent systems in collaborative working environments. It has been written in relation to a joint research program between the Enterprise Integration Laboratory at the University of Toronto, Canada, and the Distributed Artificial Intelligence Laboratory at the Technical University of Berlin, Germany targeted at developing generic agent architectures for applications in industry and telecommunication.

The paper comprises of theoretical and practical elements. The theoretical background focuses on (1) reviewing the basics of agent technology, (2) introducing the supply chain domain agent-based information technology in the business domain applied to supply chain management and (3) presents the COOrdination Language, developed at the Enterprise In-

tegration Laboratory for defining explicitly coordinated multi-agent systems. Based on the theoretical insights, the practical application starts with the introduction of a multi-user gateway for COOL agent applications, the so called Generic User Agent (GenUA). With GenUA, people can access and interact with distributed multi-agent applications directly from their working places through a Web browser. To specify user-agent-interactions, we have developed comprehensive abstract grammars which can be both immediately interpreted at the agent level and adequately represented towards the end-user. Through the mediation of those interactions by the gateway agent, it is possible to involve people in a coordinated and intuitive manner in the agent's execution process, and to provide an essential mechanism to get employed in real working environments. GenUA is based on an open and flexible architecture that allows for adaptation to specific requirements within the application context. A set of graphical components have been developed to represent the activities of GenUA for the benefit of end-users talking to systems of agents. The COOrdination Language and the Generic User Agent presented in this paper provide a framework for building agent-integrated business structures where people and agent collaboratively participate in business and decision processes. We describe a primary model which may guide prospective developers in designing such infrastructures step by step. To demonstrate the synergetic effects of the framework, we turned a COOL supply chain management application into a distributed, team-oriented workflow system on the Web where supply chain functions are executed in a coordinated and collaborative process between end-users and COOL agents mediated by the Generic User Agent.

Contents

1	Introduction	8
1.1	The Thesis Objective	9
1.2	Structure of the Thesis	10
1.3	Boundaries of Work	11
2	Basics of Agent Technology	12
2.1	Agents	12
2.1.1	Agent Definitions	13
2.1.2	Agent Theory	14
2.1.3	Agent Architectures	14
2.2	Multi-Agent Systems	16
2.2.1	The Basics: Agent Communication	16
2.2.2	The Means: Agent Coordination	17
2.2.3	The Goal: Agent Cooperation	19
2.3	User-Agent Interfaces	19
2.3.1	Characteristics of Interface Agents	20
2.3.2	Design Principals	20
2.4	Summary	22
3	Supply Chain Management	23
3.1	Supply Chain Management at a Glance	24
3.2	Decisions in Supply Chain Management	26
3.3	Methodological Issues	27
3.4	Technological Support	30
3.5	Motivation for Use of Agent Technology	33
4	COOL: Coordinating Multi-Agent Systems	34
4.1	COOL Concepts	35
4.1.1	Agent Coordination and Coordination Knowledge	35

4.1.2	Agent Communication	36
4.2	COOL Elements	37
4.2.1	Agents	38
4.2.2	Conversation Classes	38
4.2.3	Conversation Rules	39
4.2.4	Conversations	40
4.2.5	Conversation Manager	41
4.3	COOL Interfaces	41
4.4	Summary	42
5	Building a Web Interface for COOL	43
5.1	Overview	44
5.2	COOL-User-Interactions	46
5.2.1	Modeling Users as "Stub" Agents	46
5.2.2	Performing User-Agent-Conversations	46
5.2.3	Defining Rule-triggered User Interaction	49
5.2.4	The Pattern Grammar	50
5.2.5	Detecting, Forwarding and Visualizing Requests	53
5.2.6	Visualizing Input Requests	54
5.2.7	The Notification Grammar	57
5.2.8	Detecting, Forwarding and Visualizing Notifications and Results	60
5.3	Further Properties of GenUA	61
5.3.1	Generic approach - a trade-off?	62
5.3.2	Multiple Connections and Parallel Execution Mode	63
5.3.3	System Transparency	63
5.3.4	Autonomous Activity and Offline-Management	64
5.3.5	Customization and Adaptivity	65
5.3.6	Distribution and Communication	66
5.3.7	Authorization and Security	68
6	System Architecture and Functionality	69
6.1	The Generic User Agent	69
6.1.1	Towards an Open and Flexible Architecture	70
6.1.2	Insight to the GenUA Architecture	76
6.1.3	The Communication Component	76
6.1.4	The Service Agency	78
6.1.5	The Administration Component	80
6.1.6	The Application Manager	80
6.1.7	The History Component	86
6.1.8	The Offline Manager	87

6.2	The Graphical GenUA Interfaces	87
6.2.1	The Communication Handler	88
6.2.2	The Login Window	89
6.2.3	The COOL User Interface	90
6.3	Summary	102
7	Building Agent-Integrated Business Structures	104
7.1	Step 1: Agent Identification	105
7.2	Step 2: Agent Tasks	107
7.3	Step 3: Information and Control Flow	109
7.4	Step 4: User Role and Interface Character	110
7.5	Step 5: Specification of the User's Interactions	112
7.6	Summary	113
8	The Supply Chain Demonstrator	114
8.1	Motivation	114
8.2	The Company	114
8.3	The Agents	116
8.3.1	The Customer Agent	117
8.3.2	The Logistics Agent	117
8.3.3	The Contractor Agents	119
8.4	Issues of Multi User Mode	119
8.5	The Conversation Plans	120
8.5.1	The Customer Conversation	121
8.5.2	The Logistics Execution Net	123
8.5.3	The Form Large Team Class	130
8.5.4	The Answer Form Large Team Class	131
8.5.5	The Form Small Team Class	133
8.5.6	The Answer Form Large Team Class	135
8.5.7	The Kickoff Execution Class	136
8.5.8	The Answer Kickoff Execution Class	138
8.5.9	The Monitor Execution Class	139
8.5.10	The Find Contractor Class	139
8.6	Summary	141
9	Final Remarks	142
9.1	Summary	142
9.1.1	Theoretical Background	142
9.1.2	Practical Work	143
9.1.3	Meeting the Objective	143

9.2	Future Work	144
9.2.1	Extending the Generic User Agent	144
9.2.2	Improving, Varying and Adopting the GenUA Interfaces	144
9.2.3	Extending and Devising Demonstration Scenarios	145
10	Acknowledgements	146
A	Terminology	150
B	COOL Syntax	152
B.1	Agents	152
B.2	Conversation Managers	153
B.3	Conversation Classes	153
B.4	Conversation Rules	155
C	User Guide for the Supply Chain Demonstrator	159

List of Figures

1.1	Structure of the Thesis	10
2.1	Federated Agent System	18
3.1	Example of a Supply Chain	25
3.2	Decisions in Supply Chain Management	26
3.3	Conventional Infrastructure of Information Technology in Supply Chain Management	32
4.1	Transition diagram showing the Customer-Order-Conversation	39
5.1	Three Layer Model for the COOL Web Interface	45
5.2	User-Agent-Interactions	48
5.3	Visualization of Input Requests	58
5.4	Visualizing a User Message	62
5.5	User Application Mapping	64
5.6	Distribution and Communication	67
6.1	The Architecture of the Generic User Agent	77
6.2	The GenUA Application Management	82
6.3	Logging to the GenUA System	89
6.4	The COOL User Interface	91
6.5	Selecting a Conversation Class	93
6.6	Browsing a Conversation Class	94
6.7	Effects of Initiating a Conversation Class	95
6.8	Browsing a Completed Input Request	96
6.9	Inspecting Received Messages	98
6.10	Browsing Historical Information	100
7.1	Multi-level Supply Chain	106
8.1	Agent-based Supply Chain Model of the Demonstration Company	116

8.2	The interactive Customer Conversation	121
8.3	Composing a Customer Order via Hierarchical Dialogs	122
8.4	The interactive Logistics Execution Net	125
8.5	A Gantt Chart Example for the Logistics Manager	126
8.6	Problem Detection and Alternative Generation	128
8.7	The interactive Large Team Forming	130
8.8	Decision on Large Team Participation	132
8.9	The interactive Small Team Forming	133
8.10	A Gantt Chart Example for the Workshop Manager	136
8.11	The interactive Activity Execution	137
8.12	The interactive Contractor Replacement	140

List of Tables

8.1	The Course of the Customer Conversation	124
8.2	The Course of the Logistics Execution Net	129
8.3	The Course of the Form Large Team Class	131
8.4	The Course of the Answer Form Large Team Class	131
8.5	The Course of the Form Small Team Class	134
8.6	The Course of the Answer Form Small Team Class	137
8.7	The Course of the Kickoff Execution Class	138
8.8	The Course of the Answer Kickoff Execution Class	138
8.9	The Course of the Monitor Execution Class	139
8.10	The Course of the Find Contractor Class	141

Chapter 1

Introduction

Today's business world is characterized by an increasing rate in founding and managing enterprises that operate on the basis of worldwide networks of suppliers, plants, retailers and customers. The tendency to overcome functional barriers within and across organizations, to establish flexible and viable structures and to see individual business activities in a global enterprise context have become cornerstones of management in order to compete in the era of globalization, increasing competition and short technological life cycles.

Within dynamically changing environments, flexibility in defining and managing enterprise functions requires the coordination of business processes across distributed, heterogeneous and (semi-)autonomous units in order to satisfy their individual and shared goals. Precondition for coordinative activities is the availability and accessibility of information and an efficient communication among the organizational units. These tendencies have to be reflected by an adequate design of technological systems which allow for electronical coordination of organizational interdependencies by linking people and machines onto a single collaborative platform. A promising approach lies in *multi-agent systems* which involve users interactively by providing adequate *user interfaces*.

An *agent* is a piece of software that is *semi-autonomous, goal-oriented and entrusted in performing its functions* and that *operates globally on networks by relying on application-independent high-level communication and interaction protocols*. Agents in a business context act on the behalf of employees - they are meant to assist them in daily computer-based tasks, to provide access to information, to support decision making and enable collaborative work. *Multi-agent systems* inherently represent the distributed and cooperative nature of business organizations and business activities. A major challenge in building such systems is to accomplish coordinated behavior among the singular agents in order to achieve individual and shared goals, thus reflecting the inter-organizational actions and decisions in an electronical way. The COOrdination language (COOL), developed as an integral part of an Agent Building Shell project at the Enterprise Integration Laboratory (EIL) of the University of Toronto,

has been designed and implemented for defining networks of agents that exhibit cooperative problem solving through the execution of explicit plans ("conversations") and through the exchange of knowledge by means of KQML-style messages. Each agent is endowed with different types of plans according to its intended role in the system. Several multi-agent scenarios had been implemented in COOL devoted to one of the most challenging enterprise domains - the supply chain management.

However, for employing agent systems in collaborative working environments, coordination is a necessary but not a sufficient condition. The acceptance and utilization of computational systems, particularly if they show semi-autonomous behavior and act in a distributed manner, depends crucially on how the front-ends converge to the user's expectations and requirements. After all, those users are usually non-experts that are expected to deal with the agent system in their daily work. Consequently, a multi-agent system needs to come up with interfaces which involve users actively in the execution process in an intuitive and familiar manner, and which are immediately accessible from any computer-supported working place.

The obvious answer for direct access from anywhere is the Internet and particularly the World Wide Web. Using on-line information, offering on-line services or advertising products via the Web has become commonplace for many companies. We envisage to shift the focus from pure customer orientation or individual on-line actions towards carrying out and coordinating entire business activities and processes using next generation technology. In merging sophisticated problem solving provided by agent technology with the omnipresence of Intranet and Web technology in business, we anticipate a lively and flexible hybrid network of human users and artificial agents across any functional or geographical boundaries working together to achieve the enterprise's goals.

1.1 The Thesis Objective

This thesis is addressing the problem of employing multi-agent applications in real world enterprise scenarios. Against the background of collaborative environments in (virtual) enterprise structures, it will be investigated how users can be interfaced to distributed COOL agent systems in a flexible and domain-independent manner. A generic interface architecture will be designed which provides an open virtual framework and basic mechanisms needed for interactions of multiple users to multiple agent applications.

To enable accessibility from a Web browser and portability across heterogeneous settings, we have seized upon the JAVA language, and we will present a complex JAVA applet which involves users actively and intuitively in the agent's execution process.

To demonstrate the new potentials for collaborative work between users and agents, we will extend and improve one of the existing COOL supply chain scenarios towards a workflow system where distributed users execute routine activities in the context of supply chain management, mediated and coordinated by the multi-agent system.

1.2 Structure of the Thesis

The thesis is composed of two major parts: the imparting of necessary theoretical background and a description of the practical efforts where the majority of work has been put down.

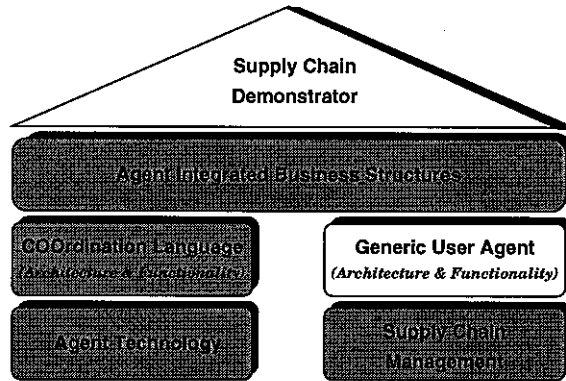


Figure 1.1: Structure of the Thesis

The theoretical background starts with an introduction to agent technology in chapter 2. Proceeding from the notion of an individual intelligent agent and a brief review of theoretical and architectural aspects of agency, key issues of multi-agent systems such as agent communication and agent coordination will be examined. The chapter will be concluded by a listing of major directions of agent applications.

In chapter 3, we introduce one of the key concepts of modern enterprise management - the supply chain management. After declaring the innovative role and crucial issues for managing globally distributed supply chains, the state-of-the-art in technological support will be reviewed. As a result, the use of agent technology for this domain will be motivated.

In the following chapter 4, the COOrdination Language will be described which has set up the technical basic for the entire practical work. This includes an examination of the major components (agents, conversation plans, conversation rules and conversation instances), control structures (conversation management) and available graphical interfaces so far.

The vision of an agent-integrated enterprise and the technological basis provided by COOL established the framework for the practical work. Chapter 5 is dedicated to outline the essential features being realized in a user-agent interface called GenUA ("Generic User Agent") and to explain how the mechanisms provided by COOL have been exploited to design and implement this solution.

We continue with examining the architecture and functionality of the GenUA system in detail in chapter 6. Prototypical graphical interfaces will be presented by means of a sample user session to a multi-agent application.

The availability of the two technological components, the COOrdination Language and

the Generic User Agent allows to think about building agent-integrated enterprise structures in the next chapter 7. As one of the most interesting domains within the variety of agent-oriented business application, we will focus on the incorporation of agents in supply chain management. A model will be presented on how multi-agent system and users can be married step by step towards a hybrid collaborative environment in a business context.

Finally, in chapter 8, the model will be exemplarily applied onto a limited supply chain scenario. As extended version of a predefined COOL multi-agent application, an integrated supply chain will be presented which allows for on-line interaction of distributed end-users with agent-coordinated supply chain functions .

In the last chapter 9, the recognitions and experiences gained through the theoretical and practical work will be summarized and prospective future work will be shown.

1.3 Boundaries of Work

When designing a user interface to multi-agent systems, it was neither the aim to cover an all-embracing theory of human computer interaction nor to tailor graphical representations towards the requirements of specific group of end-users. The focus for the solution provided has been to identify and to supply a number of general mechanisms, independently from purpose and prospective end-users of particular multi agent applications.

Also, considering the variety of different approaches in multi-agent research, an attempt to generalize a user interface for them would be presumptuous. COOL supplies a sophisticated and promising methodology and technology for designing cooperative agent environments, so we tailored the user interface to applications implemented in COOL and attached mechanisms for the on-line interaction of distributed users.

Chapter 2

Basics of Agent Technology

Agent technology has emerged as a result of the increasing tendency and necessity to integrate once self-contained standalone applications into highly distributed computational systems and the progress in computational methodology. The agent approach in software engineering is a sophisticated metaphor in conceiving and developing software systems where the entities are no longer viewed as singular functional elements with pre-defined interaction structures. Rather, we talk about behavioral objects in an environment that coordinate their actions themselves with explicit knowledge about such mechanisms and each other. With agent technology, a completely new paradigm evolved which promises to tackle the issues of distributed systems more adequately and which has opened up a new dimension of problems that can be addressed.

In order to understand the defining influence of agent technology and its utilization in business context, it is necessary to become familiar with the cornerstones of this new methodology. This is the target addressed in this chapter with the following structure. Section 2.1 encompasses a general introduction to the concept of an intelligent agent. We continue to extend the notion of a single agent to issues related to multi-agent environments 2.2. Finally, some principles of user-agent-interfaces 2.3 are reviewed.

2.1 Agents

The concept of an agent has occupied a key role in both Artificial Intelligence and mainstream computer science. The following section is meant to give an overview about the essential principles that underly agent-based software technology. First, a primary notion of an intelligent agent 2.1.1 will be declared. The paragraph on agent theory 2.1.2 deals with formalisms for representing and reasoning about the properties of an agent. Agent architectures 2.1.3 are software engineering models devised to transform the properties described in agent theory into practice.

2.1.1 Agent Definitions

There is no common notion within the scientific community about what an agent is. The term “agent” is nowadays attached to so many pieces of hardware and software, that it appears more like a buzzword than being a clearly distinguishable entity in software technology. However, most approaches define the key criterion for agenthood as a behavioral one. In this paper, we refer to the works of Wooldridge and Jennings [30]. They identify a weak and a strong notion of an agent with respect to an agent’s behavior.

The *weak notion* understands an agent as denoting a hardware or software-based system that enjoys the following properties:

- *Autonomy*: Agents operate without the direct intervention of humans or other pieces of software, and they have some kind of control over their actions and internal states.
- *Social ability*: Agents interact with other agents (which may involve human users similarly) via some kind of agent communication language (see 2.2.1).
- *Reactivity*: Agents perceive their environment and respond in a timely fashion to changes that occur in it. An agent’s environment is a defined scope selected from any kind of periphery surrounding an agent. It may encompass other agents, graphical user interfaces, legacy systems, the Internet and the physical world.
- *Pro-activeness*: Agents do not simply act in response to their environment, they are able to exhibit goal-directed behavior by taking the initiative on their own.
- *Rationality*: An agent is expected to act in a rational way in pursuing its goals as well as in selecting and executing the actions necessary. Rational behavior is constrained by having knowledge about capabilities, resources and appropriate selection strategies for alternatives.

In a *stronger notion*, additionally often human attributes are ascribed to an agent. Under discussion within the community of Artificial Intelligence are the following properties of an agent:

- *Knowledge and Beliefs*: Having knowledge is far beyond having information, it refers not only to the ability to gather information dynamically, but also to reason about it, even further to know what strategy of reasoning should be applied in a particular situation and why (known as *meta-knowledge*). A belief represents the current notion of an agent about a fact. Beliefs are subject to changes in the environment. They can be communicated and shared among agents.
- *Intentions and Obligations*: Intentions reflect long-term objectives an agent may pursue during its lifespan. From intentions, more short-time goals can be derived which in turn

require the execution of actions to get satisfied. Intentions result in a particular pattern of agent's behavior beyond executing individual actions and achieving goals. Agents may commit themselves explicitly to satisfy a particular task. Thus, obligations are directly related to an agent's autonomy: once an agent has expressed his readiness to work on a task, it is responsible to take the necessary steps in a self-responsible manner when being entrusted with the task.

- *Veracity and Benevolence*: An agent is obliged to always tell the truth. As an agent can always state its own inputs, outputs and definitions with confidence and nest conjectures inside of statements about its belief, this principle is said to be not difficult to achieve. Benevolence related to an agent means that the agent will always try to answer what it is asked for or execute what it is requested to do, briefly an agent should not behave counter-productive.

2.1.2 Agent Theory

Most of the properties associated to agents can not be defined completely as an individual item. Some of them seem to be somewhat intervoven. For example, autonomous behavior requires pro-activeness, and when an agent behaves pro-actively it can be seen as an autonomous entity. There are agent properties that can be derived from other properties. For example, only if an agent exhibits veracity, obligations makes sense. Furthermore, properties defined are constrained by other properties. For example, if an agent has to exhibit veracity and benevolence, it may be limited in its autonomy at first sight. This characteristic makes it difficult to come to a universal definition of agenthood.

In order to capture these interdependencies in formal specifications, agents are viewed as *intentional systems*, i.e. entities "whose behavior can be predicted by the method of attributing beliefs, desires and rational acumen" [30]. Again, the behavioral aspect is placed into the foreground. In generally, any object can be described in terms of intentional stance, even a simple light switch could be treated as an agent: it has the "belief" that we want it to transmit current or not when we communicate or "desire" by flicking the switch, and in an act of "benevolence" it satisfies it obeys our order. Of course, there is a much simpler mechanical explanation for the behavior of a light switch. The intentional notion is meant to be an abstraction tool, which provides a familiar way to explain and predict the behavior of more complex systems, beyond architecture and step-by-step operation.

2.1.3 Agent Architectures

Research in agent architectures faces the construction of computational systems that integrate the results of theoretical investigation. An agent architecture is a particular methodology for building agents: how agents can be decomposed into a set of components, how these components interact with each other and how they contribute to the agent's internal state and

external behavior. Wooldridge and Jennings [30] have identified three major architectural approaches: *deliberative*, *reactive* and *hybrid* agent architectures. As all three of them are relevant for the system presented in this paper, they are worth mentioning here briefly.

Deliberative Architectures

The classical approach for building agents is to view them as a particular type of knowledge based system, following the paradigm of *symbolic artificial intelligence*. *Deliberative agents* contain an explicitly symbolic model of the world, and make their decisions (e.g. what action to perform next) via logical reasoning based on pattern matching and symbolic manipulation. To construct such an agent, real world entities must be translated into accurate and adequate symbols, and appropriate mechanisms must be provided to reason with this information in a timely fashion. The emergence of speech understanding, learning, knowledge representation and automated reasoning has been driven majorly by the symbolic computational view.

Reactive Architectures

As the symbolic computation has been subjected to many unsolved problems, researchers have proposed *reactive architectures* as an alternative approach. Here, neither a kind of central symbolic world mode nor complex symbolic reasoning is involved. Reactivity adopts the view of an agent behaving like a *situated automata* based on a modal logic of knowledge. A set of "nodes" (tasks, modules, etc.) is declared, and explicitly or implicitly linked to an execution network at compile time, based on the node's input and the node's output. This pre-supposes that the agent's activity can be situationally determined, and that the agent's goal or desire system can be represented implicitly in the agent's structure according to the scheme. We will see in the corresponding chapter 6.1.1, that the Generic User Agent is based on a reactive architecture.

Hybrid Architectures

In hybrid architectures, it is attempted to combine the advantages of deliberative and reactive approaches. Hybrid agents may be assembled from plan or protocol elements which are created or executed when external events or changes in the agent's internal state occur. Any of these agent elements may be encoded in symbolic constructs. Hybrid architectures are currently a very active area of research. The COOL system 4, used in the scope of this thesis, can be put into this category. However, hybrid architectures tend to be application-specific, it is difficult to extract a formalized, general model for an agent architecture out of the variety of approaches.

2.2 Multi-Agent Systems

Agent-based software engineering was invented to facilitate the creation of software that operates in heterogenous settings. The next step is, naturally, to talk about the issue in systems of agents. A *multi-agent system* (MAS) can be defined as “a network of agents each endowed with a local view of its environment and the ability to respond locally” [18]. The systems performance emerges through dynamic interactions among the agents in a cooperating manner. However, enabling interactions among heterogenous entities requires that they have something in common. Research in multi-agent systems is concerned with coordinating intelligent behavior among a collection of autonomous agents, how they coordinate their knowledge, goals, skills and plans jointly to take actions or to solve problems. Towards this objective the following issues need to be addressed:

- how do agents communicate with each other 2.2.1
- how do agents coordinate their actions and 2.2.2
- how do agents collaborate to achieve their goals 2.2.3

This three-stages model to make a multi-agent system work will be discussed in the respective paragraphs in this section.

2.2.1 The Basics: Agent Communication

Coordination among autonomous entities requires that they have got some knowledge about each other. Though knowledge can be acquired implicitly by having the agents making assumptions or simply perceiving their environment, massive support for coordinated activities comes from explicite *communication*.

Genesereth et. al. adopt the view that “software agents are meant to communicate with their peers by exchanging messages in an expressive *agent communication language*” [12]. Such a language should allow the exchange of complex information and knowledge structures without growing excessively over domains. In order to get employed among heterogenous entities, the need for a universal communication language design is immediate, one in which inconsistencies and arbitrary notational values are eliminated. Research groups associated to the ARPA Knowledge Sharing Effort have defined an agent communication language (ACL) to satisfy these needs.

ACL provides a communication model consisting of two layers, an “inner language” called *Knowledge Interchange Format* (KIF), representing the communication content, and an “outer” language called *Knowledge Query and Manipulation Language* (KQML), representing the communication act.

KIF is a prefix version of first order predicate calculus with various extensions to enhance its expressiveness. It facilitates the encoding of simple data, constraints, negotiations, disjunctions, rules, quantified expressions, metalevel information, and many more. By means of the *KIF* language constructs, domain specific knowledge can be transformed into formal, contextually independent declarations. Because of its declarative nature, knowledge encoded in *KIF* can be transmitted and interpreted across heterogeneous systems.

KQML supplies a linguistic layer for wrapping content information encoded in *KIF*. Essentially, it supports dialogs by defining a message template. Initially, each *KQML* message is composed of a message type, called *performative*, a sender and a receiver attribute, as well as the content of the message in form of a *KIF* expression. *KQML* provides a system of standardized message types which are based on Searle's classification of speechacts [27]. The performative is a statement which may be used to define a potential intention of the sender towards the receiver about what he is supposed to do with the content. However, *KQML* does not pose an inherent semantics to performatives, the receiver may process the received content in any way. The standard message format can be enriched by additional attributes such as defining the communication protocol to enable message transmission among agents operating in distributed systems.

2.2.2 The Means: Agent Coordination

In a multi-agent system, the environment is populated by a number of agents, while each agent may pursue its own goals and may have different capabilities. Actions performed by one agent constrain and are constrained by the actions of other agents. The key problem of *agent coordination* is on how to enhance, to organize and to manage agents so that they can achieve their goals resulting in a coherent behavior of the system as a whole. To put it more simple, we need control structures or protocols that answer the question: "Who is doing what and when?". Genesereth et. al.[12] have identified two different strategies to coordinate agents: direct communication and assisted coordination.

Direct communication stands for agents that handle their own coordination by exchanging messages peer to peer. They do not need a special external coordination program and can choose their coordination strategy freely. A popular architecture for direct communication is the *contract net* approach. Here, agents in need for a particular service, issue requests to a community of other agents. Each recipient evaluates the request and, upon deciding that it is able to provide such a service, submits a (constrained) bid to the originator. After gathering all the bids, the originator decides which agents to task and then awards contracts to them. A major disadvantage of direct communication is cost. For a small number of agents, this does not pose a problem. But in a large community of agents, for example in an Internet environment, a procedure such as broadcasting requests and processing thousands of bids is prohibitive. Moreover, each agent must contain all of the code to support such a coordination scheme, which increases the complexity.

Assisted coordination is based on having a special system component within the agent community to achieve coordination. Rather than communicating directly with each other, agents share a service facility to coordinate their actions. A popular architecture is to organize agents in a *federated system*.

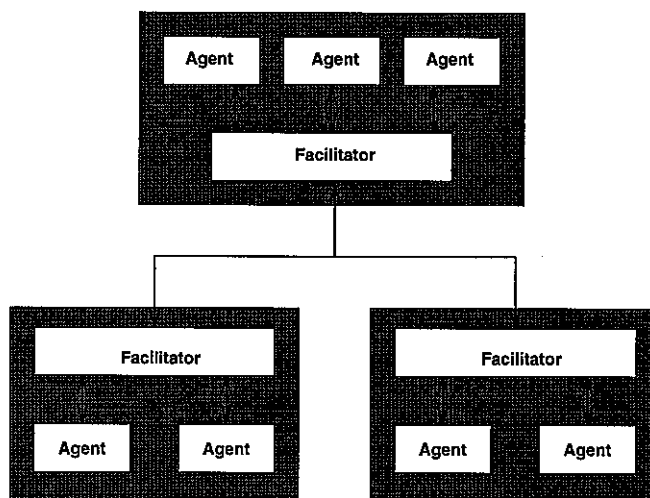


Figure 2.1: Federated Agent System

Figure 2.1 illustrates a possible structure for a federated system. Each subsystem (not necessarily associated to a machine) is comprised of several agents and a specialized coordination component, the so-called facilitator. Agents document their concrete needs and capabilities towards the facilitator by means of an agent communication language. This one uses the acquired agent knowledge to transform and route messages to the appropriate place, either to its supervised agents or to another facilitator. Facilitators can also (de-)compose multiple messages. For an efficient message processing, they can take advantage of a variety of mechanisms from simple pattern matching to automated reasoning technology.

Federated systems allow for a very dynamic system structures as agents can be introduced or removed at runtime. They may also vary their capabilities during the execution through propagating or withdrawing message patterns the agents are “able” to process. Moreover, except from providing a common communication interface to the facilitator, federated systems can be built from a set of heterogenous agents. These properties makes the assisted coordination a powerful and flexible approach which can be applied to other loosely coupled system structures, as we will see in the design of the Generic User Agent later on 6.1.1.

2.2.3 The Goal: Agent Cooperation

Individual agents and their interactions affect the characteristic behavior of the multi-agent system as a whole. However, the question is arising how to view the system from a global perspective or how to make the complete agent system work in a certain coherent manner.

Cooperation in the original sense means “working together on shared tasks”. In the wider sense, however, there is a range of different types of cooperation with respect to the balance of willingness to cooperate and self interest: unselfish, obliging, compromising, competing, antagonistic. Moreover, cooperation is closely associated to the idea of an organization. An *organization* is characterized by an objective and specific rules or politics that influence the behavior of its members.

With respect to the agent view, cooperation among agents addresses the issue how agents come together to work on a shared task. This is a question, where much influence from organization theory, cybernetics and social sciences comes into play. We talk about competencies, roles and relationships, about hierarchical structures, market metaphor and team work, about systems stability, emergence and adaptation. There is a lot of publications that treat organization and cooperation of distributed agent systems in analogy to the human world, and it would be expensive to enumerate them all in detail.

However, they all have one thing in common. The objective is to form societies of intelligent agents that cooperate consciously and effectively towards achieving a higher-level goal or task while adopting the most efficient organization for communication and coordination. We will see in the chapter on a multi-agent scenario in the supply chain domain how agents act within a team in order to efficiently satisfy customer orders.

2.3 User-Agent Interfaces

In the past, computational systems were mostly passive and ignorant towards the user. People had to supply the power to operate and the intelligence to guide its applications. Through the advent of computers and the emergence of agent paradigms, the roles of user and machines are changing. Machines will become colleagues and partners to human. Of course, people remain in overall charge of the process, but “emergent control presumes that equipment assumes more responsibility” [18].

Unlike user interfaces to standard applications that have been studied since computer systems evolved, the area of conceiving and developing agent-oriented front-ends is relatively new. The challenge is to endow the user interface with mechanisms that reflect and reinforce the virtues of agent technology for the benefit of end-users. Those kind of interfaces are in generally referred to as interface agents or intelligent user interfaces.

2.3.1 Characteristics of Interface Agents

Throughout the variety of different approaches and application domains, there is one major characteristic:

Interface agents are computer programs that employ artificial intelligence technology in order to provide assistance to users in dealing with a particular application. ... The metaphor is that of a *personal assistant* who is *collaborating with the user* in the same environment. [17]

From the perspective of the interface agent's purpose, it does not matter, whether the particular application is one of a standard type or a complex multi-agent system. However, the representation and interactions may differ to a great extent.

There is much related work being done by the computer supported cooperative work (CSCW) community. CSCW is defined by Baecker as "computer assisted coordinated activity such as problem solving and communication carried out by a group of collaborating individuals" [2]. The emphasis of CSCW is on the development of tools to support collaborative human work. The use of agent technology in these groupware tools has been proposed by several authors.

2.3.2 Design Principals

For designing an intelligent user interface, there are two different approaches how to "marry" an artificial and a human part towards a singularity, depending on the perspective one assumes. The classical artificial intelligence claims that agents can be made so intelligent that people come to view them as peers. This notion has turned out ideally and above all impracticably to solve the immediate problems people are facing at present time. For this reason the engineering approach has become more attractive, where people are represented in the network by artificial agents that make them look to other agents like computers. Some of the agents may implement conventional automatized functions integrated by a shared model or goal while computerized personal agents are assigned to each human in the network so that the other agents see only this one and not the human directly. [18]

For example, there are agents that find documents on the Internet which are of interest to the end-user. The user may order his personal agent to find documents on "Agent technology in supply chain management". The agent can consult databases, news servers or other agents for example the library agent at the University of Toronto. If both agents communicate via the same protocol, the library agent is not aware that the original request has been made by a human. It just transmits a prioritized list of documents to the user's agent in whatever format which in turn displays them nicely onto the user's screen. From the user's point of view, it is hard to distinguish whether this list has been constructed by standard software, several intelligent agents or by, say, the user's secretary.

In analogy to the objective of Computer Supported Cooperative Work, where computer technology is used to integrate communities of humans, interfacing users to agents deals with building (systems of) artificial agents into which people can be introduced. In a way, the experiences and accomplishments gained in research on CSCW over the years have opened the door to practical human agent interaction.

Cornerstone for involving people into multi-agent systems in any kind of domain is acceptance by the user. The primary goal for a user interface in such a setting is work support. The challenge is to come to an open and viable agent-based approach for user interface design. This means, that the variety of guidelines established for user interfaces to standard applications need to be transformed for interface agents. Summarizing Sanchez et. al. [16] and Hall et. al. [19], we outline the following set of requirements:

- *User awareness:* A user must become aware that he occupies a place in a distributed network and interacts with a system of different cooperating agents.
- *Accessibility:* The interface has to provide access to the available agents, their capabilities and limitations and to visualize them appropriately.
- *User control:* Users should be allowed to delegate their tasks to agents expressly. Tasks may be assigned, suspended, resumed or cancelled at any point in time. If possible, the user may undo and counteract agent actions.
- *Transparency:* The interface should guarantee access to knowledge on task progression (current state, approximate completion), task proceeding (how a task is decomposed and which agents fulfil the parts) and execution results.
- *Representation:* The interface should represent interactions between users and agents in a way consistent with the end user's expectations.
- *Security:* The interface should allow free agent operation while preserving data integrity and user privacy.
- *Resiliency:* In presence of system crashes, the interface has to record the current execution states of ongoing tasks, so that the user is enabled to resume pending tasks afterwards.
- *Adaptivity:* To adjust the interface towards specific end users, it needs to come up with general mechanisms to represent, to maintain and to apply user-specific knowledge.

These demands established also the guidelines for designing the interface between end-users and multi-agent systems in an enterprise context which will be the major part of this thesis paper.

2.4 Summary

This chapter has presented the primary concepts in agent technology. It has given an insight on what an agent is, and how agents can be formalized, composed, implemented and used. As computational systems become increasingly distributed and interconnected, intelligent agents are considered as key technology to tackle the issues arising. Having the ability to plan, to pursue their goals autonomously, to cooperate, coordinate, and negotiate with each other, to respond dynamically and flexibly to changes in the environment, and to support collaborative work across networks, intelligent agent are expected to lead to significant improvements in the quality and sophistication of software systems that can be conceived, and the application areas and problems that can be addressed.

By introducing intelligent interfaces between users and agents, a symbiotic interaction between human and machines is envisioned where users delegate tasks of varying and increasing complexity to systems of autonomous agents. The computer is considered as a medium to trigger and to represent agent actions in which users are actively participating. Agent systems are turning into a *“hybrid system”* in which some agents are artificial and others are human but the artificial agents do not distinguish between interactions with a user and interactions with other agents.

Chapter 3

Supply Chain Management

In order to maintain competitiveness in the global markets, one of the determining elements in enterprise operation is the efficient management of its supply chain.

Hinkkanen et. al. [13] outline that in the past organizations have focused their efforts on making effective decisions within individual domains and within cooperative boundaries. By treating various organizational functions, such as assembly, logistics or storage, independently, the decision complexity could be reduced. The last decade has seen the rise of a plethora of acronyms, such as “Just-in-Time”, “Total Quality Management” or “Vendor Managed Inventory”. However, most of the methodologies were aimed on one particular problem that may occur in doing business. But with market globalization and increasing competition, the costly consequences of ignoring the interdependencies particularly those with non-producing departments become more and more apparent.

For example, as the authors explain, a marketing promotion campaign should be coordinated with production planning since a higher demand may be expected. More products require more investment in raw materials or even in new facilities, which cannot be done without consulting the finance department. Likewise, the delivery of finished goods generates financial income, and so the forecast can be used to calculate the accounts payable and receivable in future. Another example, a factory that strictly keeps inventories low and produces and distributes goods in a timely manner according to the “Just-in-Time”- metaphor may suddenly face the fact of uncertain or irregular availability of input materials.

These simple description explains that the functional domains are highly interrelated and cannot be managed independently. In order to keep the market share and increase profits, a company needs to move from decoupled decision making towards more coordinated design and control of all their components in order to provide goods and services to the customer at low cost and high process level.

Against this background, the concept of *supply chain management* has been developed, a framework for integrating the actions and decisions of individual organizational functions.

The *supply chain* of a modern enterprise is a world-wide network of suppliers, factories, warehouse, distribution centers and retailers through which raw material is transformed into products, delivered to customers, serviced and enhanced. It is evident, that such a model in its entirety becomes very complex and cannot be handled without adequate computational infrastructure. It is not humanly possible to carry out this task without the use of distributed system of information technology.

This section is targeted to give an overview on the context of supply chain management and how agent technology can contribute to address the problems. After a first introduction 3.1, some decision dimensions in supply chain management are examined 3.2. Then, we will dwell on methodological issues associated to this domain 3.3. While reviewing conventional support methods at a technological 3.4 level, we are going to motivate the introduction of software agent technology into the supply chain management domain 3.5.

3.1 Supply Chain Management at a Glance

Throughout the voluminous literature, there is a universal agreement on how to define a supply chain. Janyshankar et. al. refer to a supply chain as:

... a network of (semi-) autonomous business entities collectively responsible for procurement, manufacturing, and distribution activities associated with one or more families of related products. [24]

Ganeshan and Harrison have a similar definition:

A supply chain is a network of facilities and distribution options that performs the functions of procurement of materials, transformation of these materials into intermediate and finished products and the distribution of these products to customers. [10]

Figure 3.1 illustrates an example of a supply chain. Raw material is supplied and flows downstream to the manufacturing level where it is transformed to intermediate products. These can be assembled to form finished products on the next level. Finally, products are shipped via distribution centers to customers or retailers. Though this description refers to a principle supply chain in the manufacturing industry, it can be mapped to similar structures that exist in the service industry. The individual elements of a supply chain may reside in different locations all around the world and may belong to an individual or several enterprises.

Schary and Skjott-Larsen /citesc-global explain that managing the physical product flow within an enterprise has been traditionally the focus of *Logistics*. The objective of logistics is to deliver products efficiently at the precise time and location required while ensuring a maximum of customer service and minimizing costs. This is done by managing a serie

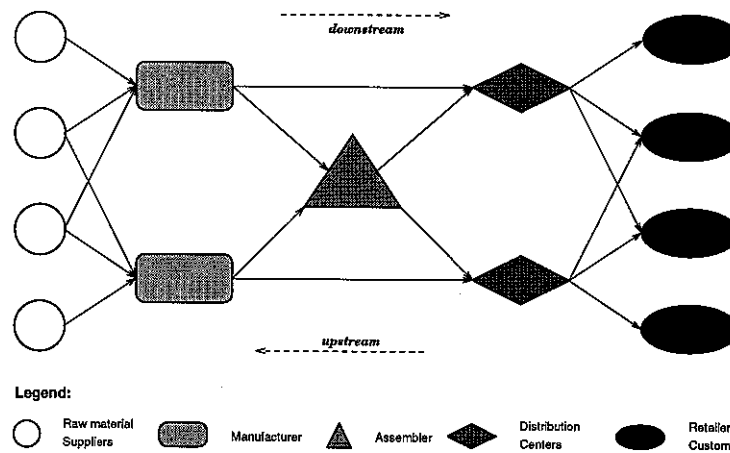


Figure 3.1: Example of a Supply Chain

of functional activities, i.e. transportation, production and inventory management across departmental boundaries which are linked through product and information flow. The focus has been on internal operations and external links to suppliers and customers. In adopting a component view, Logistics balances resources in each area through trade-offs in order to achieve integrated performance.

With the increasing tendency of globalization, outsourcing and virtual enterprise structures, the supply chain for a particular product crosses not only department boundaries but entire enterprise networks. Individual supply chain functions may be distributed across legally independent organizations. Moreover, the close interdependencies of product flow to non-productional units have become more and more apparent. Traditionally, marketing, distribution, planning, manufacturing and purchasing operate independently along the supply chain. These organizations have their own objectives which are often conflicting. For example, the objective of marketing to provide high customer service and maximum sell conflicts with the manufacturing and distribution limits. Manufacturing operations are oriented towards maximum throughput and low costs while putting less attention to inventory levels. The more companies are participating in the supply chain, the higher is the degree of conflicting goals among functional entities. As a result, there is not a single, integrated plan for the (virtual) organization.

Supply Chain Management is a strategy through which such an integration can be achieved [20]. It extends the scope of logistics to the entire set of organizations collectively participating in the product flow: from procurement of material to the delivery of finished products to the final customer. The supply chain view reflects the organizational tendency of moving away from centrally coordinated multi-level hierarchies towards a variety of flexible network structures. Supply chain management focuses on the total effectiveness of the supply chain as

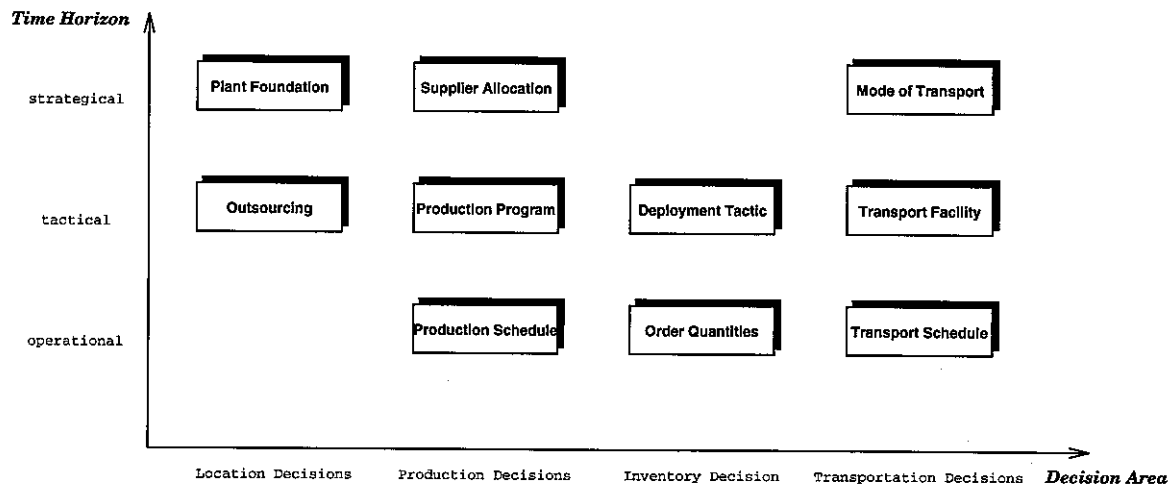


Figure 3.2: Decisions in Supply Chain Management

a whole more than on the performance of individual units. However, all members of the supply chain have stakes in it - by their actions and ability to integrate operations. Performance is not only measured how well the supply chain routinely delivers products on time or how the costs of resources are minimized. It includes the ability to flexible responses to changes in both markets and suppliers and optimal coordination of the supply chain members. A supply chain is coordinated through an information system which is accessible to all members. The agility with which the supply chain is managed in order to enable timely dissemination of information, accurate coordination of decisions and the management of actions among people and systems, is what ultimately determines the efficient achievement of enterprise goals and the viability of the enterprise on the world market.

3.2 Decisions in Supply Chain Management

Supply chain management is a procedure of continuous planning and decision making in response to changes. These decisions can be classified into a matrix of planning horizon and area. Figure 3.2 illustrates some typical supply chain decisions for each category.

At the *strategical level*, long-term decisions are made. Those are frequently related to investments, commitments or goals which cannot be withdrawn without suffering a significant disadvantage. Decisions at the *tactical level* aim at a medium-term horizon without having such an immensely binding effect. They may refine strategical targets. Finally, the *operational level* is concerned with issues of daily business. Here, the constraints given by the higher decision levels are applied and implemented.

Location Decisions refer to the geographic placement of production facilities, stocking

points and sourcing points as well as to the outsourcing decisions. The former one has inherently a long-term character, and is of great significance to a company representing the basic to access new markets with considerable impacts on cost and revenue. Such decisions require an optimization routine which includes all crucial parameters related to it, based on market analysis, cost analysis, or even political analysis. Outsourcing of business activities, such as accounting, have a more tactical nature as they are usually not connected with such immense cost as the foundation of new plants. However, a company cannot afford to establish and remove entire departments at will.

Production Decisions are made with any time horizon. At the strategical level, suppliers are allocated and associated to plants, plants to distribution centers and distribution centers to customers. Tactical decisions include what products to produce, which plant should produce them and how much capacities are needed. The operational level deals with production quantities, with all kinds of schedules from a global master production schedule down to the schedule for an individual machine and with quality control.

Inventory Decisions stipulate the means how inventories are managed. Inventories can be found at every stage of the supply chain, as raw material, intermediate and finished goods or in-process within one location. Their primary purpose is to buffer against any uncertainty in the supply chain. Efficient inventory management is crucial in the operation of a supply chain, as their holding can cost up to 40 percent of their value. Aside from the deployment strategy (push versus pull), inventory decisions are made at an operational level, concerning optimal levels of order quantities, reorder points and safety stock levels at each stock location. All of them determine, at last, the customer service level.

Transportation Decisions are closely linked to inventory decisions. For example, air shipping may be fast and requires lesser safety stocks, but it is expensive. Shipping by rail or sea may be much cheaper, but it requires to hold relatively large inventories. Operating its own car pool is another alternative for a company. Customer service level and geographical location are the determinants for such decisions at the strategical level. Transportation is more than 30 percent of the logistics cost. Thus, operating efficiently requires many tactical and operational decisions such as transportation vendors, shipment sizes, routes and schedules.

3.3 Methodological Issues

Operating an efficient supply chain needs to address three highly inter-related goals: *customer service, inventories and flexibility*.

Customer Service describes the level of satisfaction among a company's customers. Typical indicators are the ability to satisfy orders within the due date, or to deliver products at the time given.

Inventories are a central cost factor in supply chains. They bind capital in form of the products held at a time.

Flexibility is the ability to quickly respond to changes in the environment. Applied to the supply chain context, it is the ability of any entity to change its output when the demand of the successive entity changes. Thus, the flexibility of the entire supply chain depends on the interdependencies between and the flexibility of each individual entity.

Having a good customer service is an emblem for a company which is likely to attract more customers resulting in more orders and consequently in more revenue. An attractive company would be one that can satisfy any order within one day. This would require a total just-in-time production across the entire supply chain without any inventories at all. However, it would also require a constant investment in new capacities, if the quantity of orders increases. On the other hand, when the demand drops at a time, the company will find itself with much capital bound in capacities which are not used. Furthermore, a supply chain is subject to many unforeseeable events downstream: raw material does not arrive on time, production facilities may break down, employees become ill, products lack in quality. Given such an event, there would be no safety stock to satisfy the demand for all the entities upstream.

Inventories serve as a buffer to keep the supply chain flexible towards changes in demand and unforeseeable events. However, the size and the management of an inventory is the determining factor. Oversized inventories may lead to better flexibility, but they will bind a lot of capital. Inventories which are too small tend to stock-outs, when an entity downstream fails or an entity upstream increases its demand.

Thus, we need a production and inventory plan based on demand forecasts. The longer the planning horizon, the less accurate the plan will be, and the harder it will be to estimate the effects of unforeseeable events. Plans need to be continuously revised and coordinated at least with the immediate predecessors and successors in the supply chain. Nowadays, it is common for manufacturing companies to make their production and/or material requirements plan available to their suppliers who, in turn, can use this information to drive their production and distribution plan.

Introducing inventories happens initially at cost of customer service. The lead time for an individual product is affected to a great extent on how it makes its way from inventory to inventory all along the supply chain until it can be finally delivered to the customer. Moreover, there is a direct dependency between the inventory level and the ability to satisfy an order. Stock-outs in any of the inventories may result in lost contracts, which will damage the company's reputation. By monitoring the inventories at the customers, one can predict when

a stock-out would occur at the earliest, and when a customer is ready for a new shipment. This flexibility for inventory replenishment makes it possible to smooth out production and distribution peaks, eliminating either over-time labor or excess inventories of finished goods to accommodate those peak demands.

In this context, supply chain management strives not only for internal efficiency of operations, it includes managing and coordinating activities upstream and downstream in the supply chain. According to Hinkkanen et. al. [13], the following questions have to be answered for an effective supply chain management:

- How much of each raw material or intermediate or finished products should be procured or converted at each facility?
- Which supply sources should be chosen and what are target inventory levels?
- What is the best production schedule, the optimal batch size and the optimal production sequence?
- What should be the target levels of finished goods be and how can we forecast demands most accurately for each customer?
- What is the best mode for transportation, and which should be used for which shipment?
- What are the optimal warehouse locations and sizes?
- Which financial resources should be used to finance the production plan?
- How will the current material ordering policy and the customer's payment policy influence the cash flow, and how should be hedged against price fluctuations of finished goods and/or commodities?
- Which products should be manufactured in which countries and what are the global implications in terms of duties, tariffs and taxes?

For many of the tasks and decisions mentioned above, mathematical models, operational research algorithms, planning and decision supporting tools have been developed and, by and large, incorporated into computational systems. As already single issues had lead to a considerable complexity, they had been treated in the past more or less separately, resulting in a variety of "island solutions" for specific problems where interdependencies to other organizations or processes have not been taken into account adequately.

However, we have seen that all the decisions and actions in the supply chain management domain are closely inter-related. Thus, maintaining an efficient supply chain and increasing its performance is driven by two keywords: Integration and Coordination.

Integration takes place when individual organizational units trade their autonomy for membership within a larger organization, in effect removing organizational barriers [20]. The units retain the right to withdraw or to change membership but they effectively operate as part of a larger unit. Integration is a mean to achieve a higher degree of control over the product and information flow. It features a more collective approach of the individual members in order to achieve the enterprise's goals.

In order to operate efficiently in such settings, supply chain functions must work in a tightly coordinated manner. But the dynamics of the enterprise and of the world market make this difficult: exchange rates unpredictably go up and down, customers change or cancel orders, materials do not arrive on time, production facilities fail, workers are ill etc. causing deviations from plan. In many cases, events cannot be dealt with locally, i.e. within the scope of a single supply chain entity, they require the coordination of several ones to revise plans, schedules and decisions. *Coordination* takes place through mechanisms of mutual adjustment of people and organizations, standardization of processes, and shared data and values [20]. Mutual adjustment includes negotiations over operations leading to synchronous planning, scheduling and execution. Standardization encompasses specific rules that must be invariable followed by the organizational entities concerned. Information sharing refers to situations where the same information is made available to both parties for decision making.

To achieve both factors, adequate support on methodological and technological level is required. Modern supply chain management makes heavy use of network models, mathematical algorithms, information exchange and coordination strategies which are facilitated or enabled at all through the advent of information technology.

3.4 Technological Support

Supply Chain Management is driven by data. Coordinating the operations of a global supply chain requires information exchange used as a basis for decisions and actions. This process involves suppliers, production processing, assembly, transportation and distribution.

The source of data is a stream of transactions triggered by customer orders, initiating a serie of transactions within the supply chain that ultimately results in the shipment of finished products to the customer. Efficient and transparent information flow becomes the basis for coordinating operations among the units in the supply chain. They ...

- signal the start of functional activities such as procurement, production or transportation
- provide the foundation for operational forecasting and planning, for capacity planning, and for identifying trends and projecting future activities
- serve as input for analytical models

The sheer amount of relevant information to be exchanged and processed within the supply chain, the complexity of planning and analyzing procedures and the physical distribution of supply chain units requires the installation of an effective information system.

Following the organizational structure in the past, conventional systems in enterprises had been centralized units of databases and enterprise plans, that had to manage a large number of interacting entities, and on the other hand, a "mountain" of specific individual applications for a particular function which could not be integrated because they were not accessible from outside. For operating globally distributed enterprise organizations, these approaches had turned out ineffective and inflexible. With the appearance of cheap personal computers and workstations, distributed business information systems have become commonplace. Schary and Skjott-Larsen /citesc-global describe the shift from the self-contained organization to an organization as a network driven by the development in information technology.

Phase 1 Change from personal stand-alone computing to workgroup computing, from individual applications to those involving collective processes with several participants. Work activities were reorganized around specific business outcome. Fast communication protocols and shared databases were introduced which allowed for concurrent document management and computer supported decision making.

Phase 2 Change from island applications to integrated information systems. Computational systems connect and unite the enterprise as a whole. For supply chain applications, interfaces to financial management and control, human resources and physical asset management have been integrated.

Phase 3 Change to the inter-organizational network. The challenge is to build value networks in which separate enterprises are connected through a shared information system in order to create supra-organizations at a long-term or temporarily level. For the supply chain domain, they are meant to achieve superior coordination and efficiency in operations embracing both suppliers (e.g. joint product development and planning) and customers (automated ordering, joint planning)

Moreover, the company's system is now connected world-wide via the *Internet or World Wide Web (WWW)*. With that, online and real-time information has become abundant, and company's that exploit the new opportunities will keep on extending a competitive benefit in the near future. The major part of business activity on the Web is currently targeted at customer service and marketing. Electronic shopping, transportation vendors and tour operators have occupied the leading role in using Web technology to involve and to attract customers online. But the Internet medium is also widely used for internal information exchange and acquisition. Real-time information pertinent to the company's operation can be acquired in real time: stock quotes, commodity prices, etc. *Electronic Commerce* allows for even more automation and less clerical work. Ordering can be handled via Electronic Data

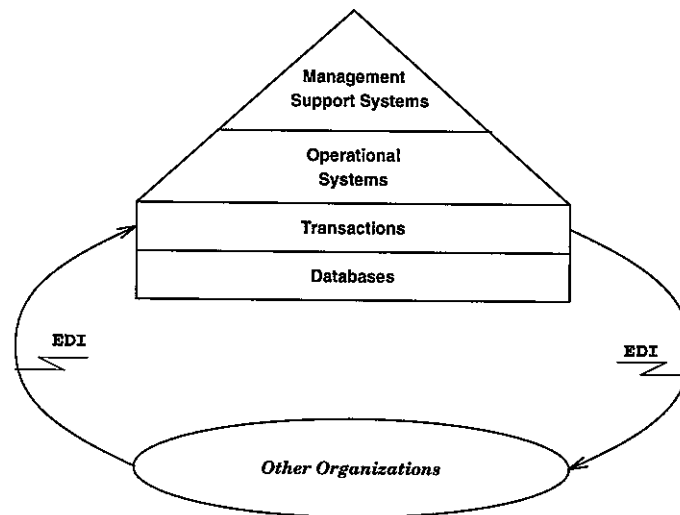


Figure 3.3: Conventional Infrastructure of Information Technology in Supply Chain Management

Interchange (EDI) messages and the whole procedure of shipment notification, billing up to and including payments via systems like Electronic Funds Transfer can be done automatically and electronically, if so desired.

With all these elements, current information systems used in the domain of supply chain management follow more or less an infrastructure as depicted in figure 3.3.

Data among organizational units are exchanged through *Electronic Data Interchange (EDI)*. EDI refers to the electronical transmission of business information between business partners, intermediaries, public authorities and others in a structured format without any need for human intervention. In the supply chain context it is now routinely used for the transmission of data including purchasing orders, invoices, shipping documents and information for monitoring the progress of orders. With EDI, business partners and organizations could reduce costs while, through eliminating re-entry errors, the accuracy of information increased. EDI also compresses time through instantaneous transmission and provides audit trails for verification.

These raw data are stored in databases for later retrieval and processed by *routine applications* which are often largely automatized. With these standard tools and their graphical interfaces, information can be monitored, manipulated and/or forwarded to other locations or higher-level applications.

Beyond that level are the so-called *management decision support systems*. These provide analytical tools and planning instruments which access and process the data at a more sophisticated level for specific analysis, interpretation and decision making.

Such a decentralized information system is deemed to connect operations across organizational boundaries making lateral organizations such as supply chains possible. It enlarges the span of effective control and coordination of operations in ways not limited by personal management contact. Decision making power is distributed to local units as it allows them to become aware of the impact of their decisions on other units.

3.5 Motivation for Use of Agent Technology

However, the current state of information technology used to support global supply chain management exhibits a number of obstacles. The major factor is the lack in coordination. Though information can be made available to the different organizational units, the computational systems do not “know” explicitly the global context for receiving and processing the information received at that moment. They do not support more sophisticated protocol to exchange meta-knowledge about the raw data between distributed system components, to request updates, to obtain more details, to communicate with a specific remote partner and so on. The responsibility to interpret the information for the management process adequately and to come to the right conclusions is still almost completely left to the human user, irrespective of his position. So it may well be that local planning actions with the corresponding tools go against the direct interest of other organizational units as there is no explicit coordination layer in the information system. The lack of immediate peer-to-peer communication in certain situations is also responsible for the fact that managers tend to reject reliance on information systems as basis for planning and decision making.

What is needed is a technology beyond distributed systems which allows for integrating information exchange and processing locally while coordinating activities with remote units. It should be enabled to interface the legacy systems used in the (virtual) enterprise structure and link people and information and decision processes throughout the organization in a flexible way. These are exactly the qualities of coordinated multi-agent systems and that's why we focus on information systems which integrate supply chain functions through systems of cooperating software agents. The determining elements, a system to develop and execute such agent systems plus a general end user interface, will be presented in the next three chapters 4, 5 and 6. With these components, we will have the necessary means to think about building agent-integrated business structures which are applicable for the supply chain management domain. This procedure will be presented in detail in chapter 7.

Chapter 4

COOL: Coordinating Multi-Agent Systems

Building a Generic User Agent providing access to a multi-agent applications requires an underlying model which substantially reflects the requirements towards coordination issues of the agents participating. The coordination problem in multi-agent applications is the problem of managing dependencies between the activities of autonomous agents, characterized by incomplete knowledge about the dynamically changing environment and about the actions, reactions and goals of the agents populating it, such that to achieve the individual and shared goals of the participants and a level of coherence is the behavior of the system as a whole.

The COOrdination Language (COOL), developed at the Enterprise Integration Laboratory, supplies such a precise conceptual model of coordination as structured “conversations” involving communicative actions amongst agents. The model is extended to a complete language design that provides objects and control structures that substantiate its concepts and allow the construction of real multi-agent systems in industrial domains. The language has been fully implemented and successfully used in several industrial applications where the most important is the integration of multi-agent supply chains for manufacturing enterprises.

From the viewpoint of the intended functionality, COOL can be considered as:

1. A language for describing the coordination level conventions used by cooperating agents.
2. A framework for carrying out coordinated activities in multi-agent systems.
3. A tool for design, experimentation and validation of cooperation protocols.
4. A tool for incremental, in context acquisition and debugging of cooperation knowledge.

With COOL, a network of distributed cooperating agents can be defined that interact with each other by using explicit coordination protocols, so called *conversation classes*. Such a

protocol spawns a state diagram, where the transitions from state to state are specified in *conversation rules*. A rule is executed under specific environmental conditions, leading to a change of the conversation state and effects in the environment. Each agent is endowed with different types of coordination plans according to its intended role within the application scenario. COOL belongs to the family of hybrid agent architectures (see section 2.1.3), it combines elements of symbolic computation such as pattern matching with reactive agent behavior upon receipt of messages or changes in the environment.

The conceptual model and design of the COOrdination Language as well as the variety of possibilities in practical use makes this system an ideal candidate for attaching a generic user interface, dedicated to provide users an easy and comfortable access to multi-agent applications and to guide them throughout the entire interaction process.

The chapter is based on several internal paper from the Enterprise Integration Laboratory (EIL), which give an overview of cool [3] [5]. It illustrates the basic concepts incorporated in COOL in section 4.1, and outlines the essential constructs of the language in section 4.2.

4.1 COOL Concepts

Before throwing a light on the individual constructs provided by COOL, the generic ideas of the framework will be presented which are:

- how do agents coordinate their actions in COOL and
- how do agents communicate in COOL

4.1.1 Agent Coordination and Coordination Knowledge

Achieving coordinated behavior among its entities is a major issue associated to multi-agent applications. For the design of COOL, the view has been adopted that the coordination problem can be tackled by having knowledge about the interaction processes taking place among agents. This kind of knowledge refers to the problem-solving competency of a multi-agent system as opposed to that of individual agents. Thus, COOL tries to come up with high level constructs for describing coordination processes and to fully support these constructs in a programming environment for building multi-agent systems. These elements are used to guide interactions among agents.

COOL poses several assumptions about the way agents are likely to achieve coordinated behavior. These are as follows:

- Autonomous agents have their own plans according to which they pursue their goals.
- Being aware of the multi-agent environment they are in, agents plans explicitly represent interactions with other agents. Without loss of generality, it is assumed that this interactions takes place by exchanging messages.

- Agents can not predict the exact behavior of other agents, but they can delimitate classes of alternative behaviors that can be expected. As a consequence, agents plans are conditional over possible actions/reactions of other agents.
- Agents plans may be incomplete or inaccurate and the knowledge to extend or correct them may become available only during execution. For this reason, agents are able to extend and modify their plans during execution.

The most important construct of the language is the *conversation plan* or *conversation class*. Conversation plans specify states and associated rules that receive messages, check several conditions, transmit messages and update the local status. COOL agents possess several conversation plans which they instantiate to drive intercatations with other agents. Instances of conversation plans, called *conversations*, hold the state of execution with respect to the plan. Agents can have several active conversations at the same time and control mechanisms are provided that allow agents to suspend conversations while waiting for others to reach certain stages. Thus, they can create conversation hierarchies dynamically in which child conversations are delegated issues by their parents and parents will handle situations that children are not prepared for.

4.1.2 Agent Communication

An agent is viewed as essentially performing a *transduction*. It takes a stream of input messages from the environment (in general composed of other agents which may include users similarly) and generates a stream of output messages to the environment, mediated by its internal state.

COOL provides a communication mechanism that implements an extended version of the KQML language. KQML [7] has been designed as a universal language for expressing *intentions* such that all agents would interpret a message identically. It supports communication through explicit linguistic actions, called *performatives*. As such, KQML relies on the speech act [21] framework developed by philosophers and linguistics to account for human communications. In COOL, the communicative action types, as defined by KQML, can be utilized and extended by further, more specific directives. Also, no particular content language is imposed. The same information content can be communicated with different intentions. For example:

- (*ask* (produce 200 widgets)) - the sender asks the receiver if the mentioned fact is true;
- (*tell* (produce 200 widgets)) - the sender communicates a belief of his to the receiver;
- (*achieve* (produce 200 widgets)) - the sender requests the receiver to make the fact one of his beliefs;

- (*deny* (produce 200 widgets)) - the sender communicates that a fact is no longer believed;

Consequently, a typical COOL message notation may look like this:

```
(PROPOSE
  :language KIF
  :sender LOGISTICS
  :receiver PLANT
  :content (produce 200 widgets)
  :conversation C1
  :intent (explore fabrication possibility))
```

A new performative *propose* is introduced, which can be seen as an refinement of the common *ask* structure. The sender *logistics* makes a proposal to the receiver *plant*, expecting a response such as *accept* or *reject*. The additional slot *:conversation* is used to relay the message to a shared conversation instance between the communicating agents while *intent* specifies a particular intention the sender is pursuing by sending this message to the receiver. Such a descriptive element allows the receiver to find a corresponding rule to be executed in the current state or to trigger another conversation of his own which matches the intent slot.

4.2 COOL Elements

For building multi-agent applications, COOL provides a number of templates to capture the coordination knowledge needed among the entities and to execute structured interactions. The most important objects are:

- COOL Agents 4.2.1
- COOL Conversation Classes 4.2.2
- COOL Conversation Rules 4.2.3
- COOL Conversations 4.2.4
- COOL Conversation Manager 4.2.5

Each of these objects is described in the next paragraphs.

4.2.1 Agents

As stated in section 2.1.1, an agent is considered as an entity which acts significantly autonomous, goal-oriented and is entrusted in performing its functions. It operates based on internal and shared knowledge, beliefs and intentions. In COOL, an agent is a programmable entity that can exchange messages within structured plans with other agents targeted at a particular goal. These plans are defined in *conversation classes* and carried out by plan instances *conversations*. As the complete agent interactions are controlled by these plans, it is possible to define coordinated behavior up to complex cooperation patterns, for instance agent negotiation [31] about their goals and tasks.

Agent knowledge is bundled in environment variables while different environments are associated to the agent level (“global” knowledge) and to the conversation level (“local” knowledge). Moreover, knowledge, particularly coordination knowledge, is implicitly represented in an agents’ conversation classes and rules.

An agent is defined by giving it a name, setting the variables that form its local database and optionally specifying a conversation plan for its *initial conversation*. When an agent is created, its initial conversation starts running and while it runs, the agent is “alive”. If there is no initial conversation defined, the agent remains in a “waiting” state until a message from another agent has been received that can be mapped to one of the agents’ plans, hence triggering a conversation dynamically. Agents are run as lightweight processes inside *execution environments* that reside on local or remote sites (see 4.2.5). The transportation medium used among different sites is a TCP/IP connection.

4.2.2 Conversation Classes

COOL agents interact by carrying out plans. COOL provides a construct for defining generic plans, the *conversation classes*, and a corresponding instance construct, the actual *conversation*.

Conversation classes are rule based descriptions of what an agent does in certain situations (for example when receiving a message with given structure). COOL provides ways to associate conversation classes to agents, thus defining what sorts of interactions each agent can handle. A conversation class specifies the available conversation rules (see below), their control mechanism and the local knowledge base that maintains the state of the conversation (see below). Conversation rules are indexed on the finite set of values of a special variable, the *current-state*. Because of that, conversation classes and actual conversations admit a finite state machine representation that is often used for visualization and animation.

Figure 4.1 depicts an example of a transition diagram for a conversation class governing the Customer’s conversation with Logistics in the supply chain application. It is used for handling a product order submitted by the online customer between the Customer and the Logistics agent. Nodes represent the states of the conversation, arrows indicate the existence

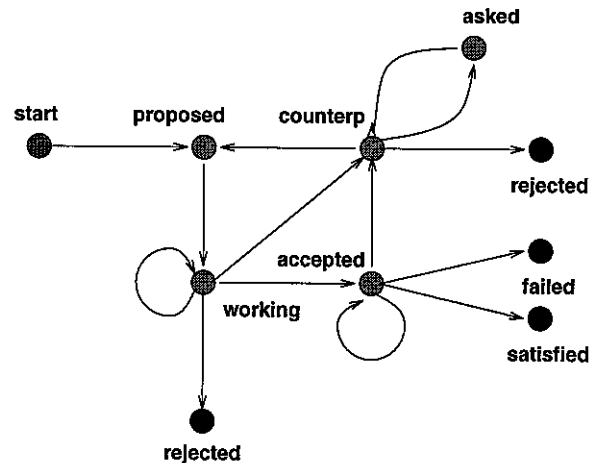


Figure 4.1: Transition diagram showing the Customer-Order-Conversation

of rules that move the conversation from one state to another.

4.2.3 Conversation Rules

Conversation rules describe the actions that can be performed when the conversation is in a given state. If there are more than one rule applicable, it depends on the matching and application strategy of the conversation's control mechanism which of the rules is actually executed and how this is done. Typically (and this is for all the use in here), the first matching rule in the definition order is executed.

A rule matches when certain conditions are fulfilled. These conditions include receiving a message of a particular structure and/or applying a predicate in the current environmental state successfully.

The execution of a rule always leads to changes (actions) within a conversation. These include one or more of the following items:

- The state of the conversation will be changed
- A message to another agent will be created and transmitted
- A (number of) function(s) will be executed which may manipulate conversation and agent variables or trigger new conversations

A rule may be defined as incomplete or complete. Complete rules are executed "silently" without notifying the user. In incomplete rules, conditions and/or actions can be omitted, for instance the exact message composition to be transmitted. When the system encounters

an incomplete rule, it can not know what rule it should execute in a given state (missing condition) or what should happen by executing a rule (missing action). Thus, a graphical interface 4.3 is popped up allowing the user to inspect the current conversations and prompting him to edit the missing parts or to decide which conversation rule should be applied.

4.2.4 Conversations

Actual conversations instantiate conversation classes and are created whenever agents engage in communication. An actual conversation maintains the current state of the conversation, a set of *conversation variables* whose values (being persistent for the entire duration of the conversation) are manipulated by conversation rules, and various historical information accumulated during conversation execution.

Each conversation class describes a conversation from the viewpoint of an individual agent. When an agent wishes to initiate a conversation in which it will have the initiative, it creates an instance of a conversation class. Once this instance is created, messages will be sent and received according to the rules defined in the conversation class. For two or several agents to “talk”, the executed conversation class of each agent must generate sequences of messages that the other’s conversation class can process. Thus, agents that carry out a particular conversation *C* actually instantiate different conversation classes internally with the same name in each agent. This allows the system to route messages appropriately and is the reason for having an additional slot *:conversation* attached to each message.

An agent may have multiple conversations at the same time. COOL provides mechanisms for deciding which conversation to continue next and when to suspend or resume a conversation. Moreover, *nested conversation execution* is featured in which the current conversation of an agent is suspended, another conversation is created or continued, with the former conversation being resumed when specific conditions hold (like the termination of a conversation). Because conversations can be accessed and inspected, the states and variable values of a conversation that another conversation waits for can be used by the waiting conversation when the latter is resumed. In this way, concurrent conversations can be synchronized. Such an execution mode makes it possible to break complex protocols into smaller parts.

For example consider again the supply chain application. The Customer agent may have a conversation with the Logistics agent about a new order. Logistics may temporarily suspend this conversation in order to start a new conversation with a Plant agent to inquire about the feasibility of a particular manufacturing process. Having obtained this information, Logistics will resume the suspended conversation with Customer telling him whether the order can be satisfied or not. This mechanism is discussed in detail in the description of the GenUA demonstrators /refdemo-conv-classes

4.2.5 Conversation Manager

Agents carry out conversations with other agents or perform local actions within their environment. Cooperating agents exist in local or remote *execution environments*. To control agent execution within an environment, *conversation managers* are used. This is necessary as COOL agents are not separate processes, hence have to be run successively by a controlling routine. A conversation manager defines the set of agents it manages, specifies a control function that at each cycle selects an agent for execution and defines the instruments (e.g. tracing, logging, etc.) for it. The purpose of execution environments is to “run” agents by message passing and scheduling agents for execution. Environments exist on different sites (machines) and a directory service makes message transmission work just the same among sites as within sites. Consequently, an application composed of COOL agents can run in an execution environment that exists on a single machine, or the agents can be distributed in several execution environments running on several machines. Each execution environment gets its own conversation manager to execute the agents associated to that environment.

4.3 COOL Interfaces

Coordination knowledge for comprehensive applications like agent-integrated supply chain management is generally very complex, hard to specify at any time and very likely to change even dramatically during the lifespan of the application. The nature of designing a coordinated multi-agent system with wide-ranging effects on business processes involving a variety of users makes it difficult to capture all the knowledge needed beforehand. Consequently, such a large-scale system should emerge by acquiring knowledge during the online interaction rather than by offline interviewing experts. This is the principal idea of the system’s execution mode and a full visual environment developed for this purpose. They allow:

- an *incremental modification* of coordination protocols, e.g. adding or modifying conversation classes and rules
- a system operation mode on *incomplete knowledge* giving the users the chance to intervene and take any actions they consider as appropriate
- a system operation in a *user controlled mode* in which the user can inspect the state of the interaction

The basic elements for this kind of in-context knowledge acquisition are incomplete conversation rules. Here, either the condition or the action part is missing (as the knowledge about it has not been captured in detail). If the condition is not exactly specified, the system can not decide whether this particular rule is applicable in a given state, hence it does not try to execute the action part. Similarly, if the action part is incomplete, the system does

not execute any actions as there might be a need to manipulate environment variables or transmitting messages in the given state. Rather, the user should handle such a situation. Therefore, a graphical interface is popped up where the user can decide to make choices, execute actions and edit rules and conversation objects. The effect of any user action is immediate, hence the future course of the interaction can be controlled in this manner.

Before taking any action, the user can inspect at that moment the state of interaction as a diagram, the values of environment variables, the history of exchanged messages and applied rules and so on. The entire system execution is initiated and supervised from a separate Conversation Manager window

4.4 Summary

In this section, the COOrdination Language (COOL) has been presented, a language and environment for building multi-agent applications. COOL agents coordinate their actions based on explicit plans, the conversation classes. Those plans can be represented as finite transition diagrams consisting of states as nodes and conversation rules defining the transitions. A conversation rule specifies the conditions in which it is applicable and the actions to be performed. Possible conditions include the receipt of a particular message structure from another agent and predicates applicable on the state of environment variables. Possible actions involve transmitting a message to another agent, creating new conversations and manipulating environment variables. Conversation classes are instantiated on demand, moving from state to state as defined by the conversation rules. COOL agents may have many conversations ongoing at the same time. They can reside in distributed execution environments where each of them is controlled by a conversation manager.

COOL allows an incremental acquisition of coordination knowledge from the user during the execution. By defining a rule as incomplete, the system will pop up a graphical interface prompting the user to fill the missing parts or to decide on how a conversation should be continued in a given state.

The design and implementation of the COOrdination Language makes it an ideal platform for developing coordinated agent systems. Building rigorously on structured interactions among agents, COOL offers a comprehensive way to involve users actively in the execution and emergence of large-scale agent applications. Thus, it supplies a fundament for creating a Generic User Agent, an interface solution, that allows users to participate in multi-agent applications independently from their location. How the COOL features have been exploited consequently in designing the interface solution will be described in the next chapter.

Chapter 5

Building a Web Interface for COOL

Looking back to section 4.3 in the previous chapter we can see how an end user can be involved in the execution of a complex multi-agent scenarios, and how the application itself evolves by in-context acquisition of knowledge. However, the graphical COOL interfaces are aimed at developing, maintaining and enhancing multi-agent environments from the perspective of system developers. They display internal views and require inputs that come up very close to implementation details of the language. Furthermore, the interfaces provided by COOL are only accessible from within the execution environment itself. There is no way to use them across heterogenous distributed platforms.

For involving people in collaborative settings into distributed multi-agent applications, the graphical COOL interfaces are not adequate. A real user interface should be tailored to the needs of those humans who are supposed to interact with the multi-agent system in their daily work, under the assumption that most of them are non-system experts. Moreover, they should be allowed to access the multi-agent system in a familiar way directly from their working places.

As a consequence, we have built a gateway solution called *Generic User Agent* (GenUA) which should bridge these gap. It acts as an intermediary between COOL agent executing plans in a multi-agent environment and the associated end user(s) monitoring and controlling the behavior of agents through corresponding graphical components. By making corresponding visual components available on the World Wide Web, every user will be able to talk to a COOL multi-agent application through a Web browser.

This chapter is meant to present the basical ideas for the design of the COOL Web gateway. After a brief introduction 5.1, it will be shown how the agent conversation mechanism provided by COOL can be used to involve users interactively in the execution process and about the role GenUA has to play in this context 5.2. Afterwards, we will dwell on additional features of the gateway for operating in a distributed environment 5.3.

5.1 Overview

Following the specification of Hall et.al.[16], we define our interface as a component of a COOL multi-agent system which supports interaction with the user. The major role is that of a router between users and agent applications. For that we need to handle two key aspects:

User representation: The interface should model and embody the user within the multi-agent system, so that all agents have an interlocutor when user interaction is required, despite the fact that the user might be not aware of their existence in its entirety.

Agent representation: The interface should exhibit the agents and their contribution to the problem solving process in a way consistent with the end user's expectations and provides as much transparency as needed.

To achieve this, the interface needs to meet several functional dimensions:

1. *System Coherence* It should exactly reflect the communication and coordination mechanisms used among the COOL agents. In this way, the agents participating do not distinguish whether they are interacting with a user or another agent. By hiding the existence of a human participant towards the rest of the multi-agent system, it can be executed without major changes.
2. *Accessibility, Control and Visualization* It must give the user access to the elements of the scenario (agents, plans, conversations). It should visualize structured interactions between users and agents in a familiar manner while abstracting completely from internal representations of the multi-agent system.
3. *Web Server Features* It has to handle aspects of multiple and parallel interaction between users and COOL applications while both parties are distributed in the network.

In order to implement these functionalities in a configurable way, we divide the COOL Web interface into several layers with different responsibilities. Figure 5.1 shows these layers at a simplified level.

At the lower level, we find the COOL multi-agent system, which one is initially comprised of an agent application being implemented in the COOL language and executed in an agent execution environment. On top of that we are going to place an interface layer called *Web API*. This one introduces specific objects representing Web users that behave similar to COOL agents with respect to communication and control and provides a number of applicative functions which are accessible from outside the execution environment.

At the middle level, we have a component called *Generic User Agent (GenUA)*. This is where the major interface logic resides. The purpose of GenUA is to act like a Web server with respect to linking distributed users with distributed applications while enabling

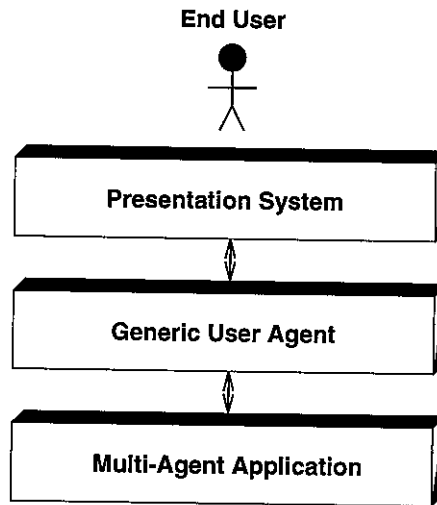


Figure 5.1: Three Layer Model for the COOL Web Interface

straight-forward interactions between them through the Web API. The essential aspect is to completely separate the visual control objects and the presentation platform from the actual interaction logic. In this way, we are free to design different graphical components for end users while using the same interaction mechanism.

Looking at these characteristics and the context of being employed in a collaborative environment frames this interface component as a candidate for an *interface agent* (see definition in section 2.3.1). An individual multi-agent system is in generally meant to perform some kind of cooperative problem solving, for example an agent-integrated supply chain or an office system of collaborating desktop agents. By facilitating the aspect to link users with such a system, this interface component can be seen as an active part of the problem solving process. That the Generic User Agent enjoys a number of further agent properties will become more clear particularly in the chapter on its architecture 6.1.

Finally, on top of the layer model, there is the actual *presentation system*. This is meant to be a set of graphical objects which represent objects inside and interactions with COOL multi-agent application directly onto the user's terminal. In the context of this paper, we address a World Wide Web browser as universal presentation platform. However, the design the Generic User Agent allows that other platforms with different graphical elements can be linked in a similar manner. How such a graphical user interface may look like and how it is used to represent user-agent-interactions will be presented in paragraph 6.2.

5.2 COOL-User-Interactions

COOL allows programming multi-agent systems in which the agents operate by executing explicite plans and interact by transmitting messages. For designing a user interface, the question to be solved is: How to make users part of these plans and to allow them to interact with agents? Our approach is based on a explicite representation of users in the multi-agent system and on performing “conversations” between users and agents which are derived from the mechanisms provided by COOL. The following paragraphs treat these issues in detail.

5.2.1 Modeling Users as “Stub” Agents

A key axiom for linking users to multi-agent systems is that its agents should not distinguish whether they interact with another agent or with the user. Our approach tackles this by representing every human participant in a scenario as a *stub agent*, one that by means of a number of applicative functions pretends to be an agent by embodying exactly the same communication and coordination mechanisms as used among the rest of the agent community. Whenever a user accesses a particular COOL scenario, such a personalized stub agent is created and becomes part of the agent execution environment. This means it can be addressed as an object in the same way as agents or conversations. However, neither agents nor conversations need to be aware explicite from the presence of such a stub agent. Basically, stub agents enable:

- the detection and evaluation of input or decision requests from agents according to their plans and their redirection to the user
- the sending of completed input or decisions to any agent in order to be processed as part of a conversation plan
- the reception of a variety of different notifications or execution results and their redirection to the user

By means of the applicative functions, the stub agent allows the user to browse through the entire world of defined agents and conversation plans without bothering the agent community in their execution.

5.2.2 Performing User-Agent-Conversations

As we have seen in section 4.2.1, COOL agents become active when one of their conversation plans is instantiated. After that, the plan is executed by moving from state to state according to the associated conversation rules. We now want to establish user interaction as an essential part of executing a plan. However, asking a user for input or presenting results should not happen at arbitrary moments during the system’s execution. Thus, we assign

user interactions to specific *execution states*. COOL conversation plans already feature state-based mechanisms. In section 4.2.3 we mentioned that in each state of a plan instance, the system checks for rules that are applicable under given conditions and executes them. As a consequence, we base user interactions on extending the COOL conversation rules by two mechanisms

- Input Requests, for defining a template for the input expected
- User Messages, for transmitting execution results or notifications to the user

Diagram 5.2 outlines the mode which we envisage for user interaction with COOL agents.

As we can see, essentially a plethora of asynchronous, sequential dialogues will be spawned, for any interaction between one user and one conversation. According to the conversation plans, either an input request to the user will be issued, or a result/notification will be transmitted or the execution is running “silently” without bothering the user. He only cares about completing the requests and sending them back to the agent environment. There is no way to create arbitrary messages and to send them at will. Rather, the initiative is taken by the agents with respect to their plans. When they “want” to have user input, they ask the person to provide it, when they “want” to notify the user, they can feel free to do so.

The major effect is that by making the multi-agent system issuing input requests at defined states, the user will be freed from any wondering what to do and when during the execution of the system. Rather, he will be *guided* through the entire interaction process. This is essential in dealing with such a highly complex and sensitive software like multi-agent systems. All we need is ...

- to ask users for input or decisions when necessary and otherwise to run in the background, and
- to visualize requests in form of templates which just have to be filled by the users with appropriate values.

A similar approach is taken for sending messages from the multi-agent system to the user. Every conversation plan encompasses a variety of local actions and interactions with the environment. We introduce sending user messages as one of the possible actions. As all users will be represented in the scenario as personalized stub agent all we need is ...

- to transmit a message to the stub agent concerned with an expressive content, and
- to visualize the content adequately onto the user's screen

Furthermore, our model allows *asynchronous* and *multi-threaded* interactions between the user and the multi-agent system. Remember that agents can have multiple conversations and

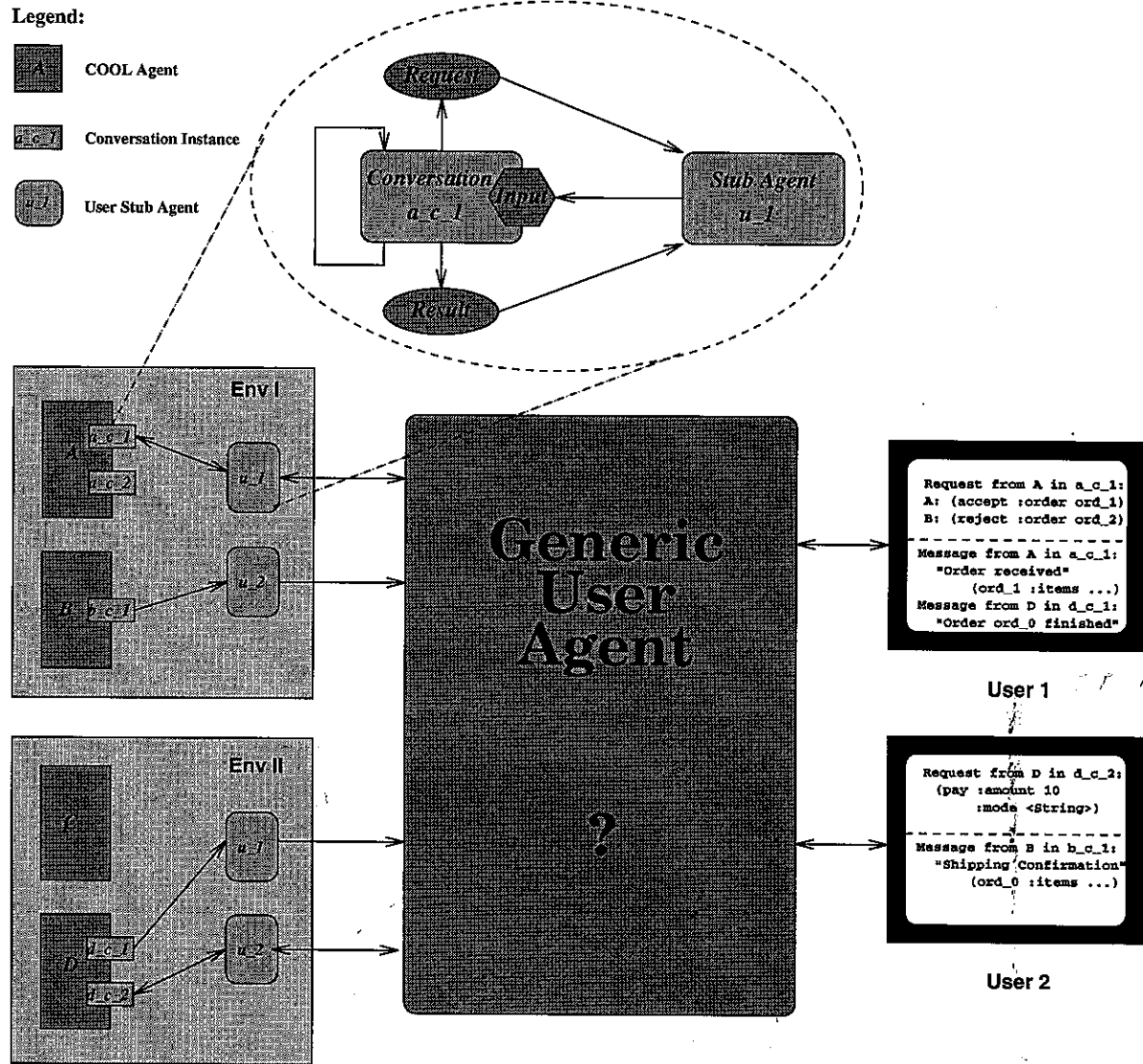


Figure 5.2: User-Agent-Interactions

a scenario may be comprised of many different agents. As all of them run concurrently, they can issue a multitude of input requests to different users at the same time (see diagram 5.2). Each user can select to answer the requests appearing on his screen whenever he wants to and in whatever order he prefers. Assuming that the requests are visualized in an expressive way, and the user is aware that nothing will happen unless he responds, he can focus on a particular dialogue of interest at any point in time, and has full control on his ongoing dialogues. Analogous, messages can be received by a user from anywhere in the multi-agent system if so desired. Assuming that they come up with clear associations where they belong to, the user can inspect and save the message, and use the information gained later on, perhaps in a completely different dialogue.

This kind of *agent-based activation* establishes the core how we model user interaction with a multi-agent system. One may argue that this is a very restrictive view as the user does not play a really active role within the scenario. But the benefit is to involve users in a *clearly structured* interaction process supplied by the conversation plans. Wrong input and user's confusion as a frequent source of error is eliminated beforehand. Also, we do not need to provide additional elements to represent user tasks. By capturing task knowledge in corresponding plans and interactions of agents, they can exhibit to work on specific tasks on behalf of end-users. Both multi-tasking and multi-user operation are supplied inherently.

The following paragraphs discuss in detail how these interaction mechanisms are achieved.

5.2.3 Defining Rule-triggered User Interaction

COOL conversation plans consist of a set of rules which spawn a state diagram. A conversation rule specifies the conditions under which it is applicable and an action part which is executed when the conditions are satisfied. Conversation rules are always related to specific states of a plan. (see 4.2.2 and 4.2.3).

The key to involve a user during the execution of the multi-agent system are those conditions where a certain message pattern is expected. From the point of the conversation instance, it does not matter whether this message will be received from another agent or from an external supplier, i.e. the user. If a conversation comes to a state where no rule is applicable at the moment, because none of their conditions is met, this conversation is simply suspended until something happens in the environment, while the rest of the system proceeds. Now, if the conversation is in a state where rules are associated that expect a message pattern in order to get activated, and a message that matches one of the pattern arrives, the corresponding rule will get applied, the conversation will be resumed and moves to a new state, and the process continues.

Consequently, the problem of having a COOL conversation (or a COOL agent) requesting the user for input is reduced to (1) propagating the expected message pattern to the user at the right moment, (2) visualize the message pattern to the user and having him fill the missing elements and (3) making him send back the completed input to the agent environment.

How such an interaction request can be specified in a generic and comprehensive way will be explained in paragraph 5.2.4. Paragraph 5.2.5 treats the issue how these requests get delivered to the user(s) concerned and how they can be visualized.

Vice versa, a conversation rule may specify in its action part a variety of activities: manipulating agent knowledge, transmitting messages to other agents, querying legacy systems, initiating new plans. Thus, the action part is the ultimate place to have the conversation (or, say, the agent) sending expressive results or notifications to the user. However, agents or conversation plans are not aware of the user's presence explicitly. We need to (1) handle user notifications just as any other kind of rule action (2) map those actions to the personalized user stubs, (3) having the stubs forwarding the notification to the concrete user and (4) visualize the results to the user. How such a interaction result can be specified in a general way will be explained in paragraph 5.2.7. Paragraph 5.2.8 deals with the delivery and visualization of results to the user(s) concerned.

5.2.4 The Pattern Grammar

In generally, input to a system is checked on correctness when it occurs resulting either in a successful processing or in an error message. For conventional interactive systems such as shells or word processors, this approach might be sufficient. But interacting with a multi-agent system requires more sophisticated approaches both for the benefit of the user, who should clearly know what to do in a particular situation, and for the benefit of the system, as wrong or incomplete input may cause unexpected effects within the entire agent community. For this reason we attempt to prevent errors by defining the input format beforehand. The question is how agents do formulate their requests to an user.

For formalizing a dialog request in a generic and expressive way, we devised a *pattern grammar*. The specification of the grammar presented below obeys the standard EBNF notation where

```

<word> are non-terminals
  -> represents an derivation
  | represents the logical OR
[...] is an optional element
{...}* symbolizes repetition 0 or more times
{...}+ symbolizes repetition 1 or more times

<request>          -> (<comment> <pattern> <described-vars>)

<comment>          -> <string>

<pattern>          -> ([symbol] {<keyword> <value>}+)
```

```

<keyword>      -> :<symbol>

<value>        -> <number>
                | <string>
                | <variable>
                | <symbol>
                | <pattern>

<variable>     -> !<symbol>

<described-vars> -> ((<variable> <described-value> [<default-spec>]))*)

<described-value> -> <number-spec>
                    | <string-spec>
                    | <date-spec>
                    | <symbol-spec>
                    | <any-spec>
                    | (listof [<length>] [<described-value>])
                    | (list <described-value>+)|
                    | <pattern>

<length>       -> <integer>

<default-spec> -> <number>
                | <integer>
                | <natural>
                | <string>
                | <symbol>
                | (date (<integer> <integer> <integer>))
                | (list <default-spec>*)

<number-spec>  -> *number*
                | *integer*
                | *natural*
                | ...

<string-spec>  -> *string*
<date-spec>    -> *date*
<symbol-spec>  -> *symbol*

```

```

<any-spec>      -> *predicate argnumber arg1 arg2 ... argn*
                  where
                    'predicate'
                      the name of a predicate to be applied
                    'argnumber'
                      the number of predicate arguments
                    'arg1 ... argn'
                      the individual arguments for the predicate
<number>        -> 'a floating point number in decimal notation'
<integer>       -> 'an integer number'
<natural>       -> 'a natural number'
<string>        -> 'anything enclosed between ' and ''
<symbol>        -> 'any token that is not a number.
                  Tokens must not contain whitepaces.'

```

The explanation of the grammar's purpose and operation is done best by means of a simple example. Consider the following instance:

```

( 'Please fill the customer order!'
  (propose
    :sender rh
    :receiver customer
    :content !order)
  ((!order (customer-order
    :product bananas
    :amount !amount
    :max-price-per-item !max-price
    :properties !properties
    :payment !pay)
    (!amount *integer* 10)
    (!max-price *number* 1.0)
    (!properties (description
      :color !color
      :size !size
      :quality-level !quality))
    (!color *string* 'yellow')
    (!size *number*)
    (!quality *integer* 1)
    (!pay (listof 1 *string*)
      (list 'cash' 'credit card' 'cheque')))))

```

This is an example for a dialog request where a rule for a Customer conversation provides a template to user “rh” to help him specifying a customer order on bananas.

The request tuple starts with a *comment*, which may be anything deemed to be useful to describe the purpose of the request or to give hints for particular elements to the user.

Following there is a *pattern* composed of a header (here a KQML performative) and a number of “:key value” pairs. Such a structure can be easily visualized as spreadsheets or dialog boxes which are familiar to most of the users. The pattern is supposed to match exactly those in the condition part of a conversation rule, following the symbolic *pattern matching* algorithm. Pattern matching provides a convenient way to parse structures and to extract values. As stated above, once a conversation rule becomes applicable because a message matching its pattern has been received, the rule will fire and execute its actions.

A keyword is any meaningful symbol to name a value. In general, values that are no variables (no leading “!”) are fixed, they cannot be changed by the user but can be displayed. Any variable inside a pattern will be replaced by a matching *variable description*. For instance, the “!order” value for the “:content” keyword is a further pattern, again with fixed values and variables. In this way, we can construct input requests by a hierarchy of nested elements.

The *variable description* specifies the exact content for each variable used in the pattern. There is no need to obey a particular order, however, all variables used should be also described. Take a look at the description of “!amount”. It simply states that the input value for the keyword “:amount” has to be an integer which is set to 10 by default. Of course, a user may order more or less than 10 bananas and can change this value, but he cannot delete it. As for the bananas’ “!size”, the user may enter any floating point number, or he may leave it blank indicating that he does not care. Take a last look on “!pay”. Here, a list of one string element is expected on how the customer wants to pay his bananas. We provide a *selection list* of standard methods for payment, but the user may feel free to declare another arrangement as well.

To summarize, with the pattern grammar, requests of arbitrary complexity can be build. Patterns visualized as hierarchical dialog boxes are a common notion for an user. Values can be predefined as fixed or variable. Variables are declared with standard data types and may be set to defaults. The essential questions to be solved are how to detect and to forward requests and how to visualize them adequately.

5.2.5 Detecting, Forwarding and Visualizing Requests

Detecting requests to users from (suspended) conversations and submitting them to the user is a team work of GenUA and the user stub agents. At any moment of a COOL scenario’s execution, there might be conversations waiting for input from users. GenUA observes the complete agent execution environment for waiting conversations and asks the individual user stub agents continuously for possible requests. In turn, the stub agents consult the rules

for the conversations concerned (depending on the conservation state) and retrieve all the requests pertinent to the user they represent. As a result, there may be:

- (a) different requests for different users (“multi user mode”)
- (b) the same requests for different users (“broadcasting”)
- (c) different requests for the same user pertinent to instances of various conversation plans (“multiple dialogs I”)
- (d) different requests for the same user pertinent to several instances of the same conversation plan (“multiple dialogs II”)
- (e) different requests for the same user pertinent to one instance of a conversation plan where several rules are potentially applicable in a state, while expecting different patterns to get fired (“decision dialogs”)

All these situations may occur simultaneously at the very moment when GenUA polls the user stub agents. However, GenUA is capable to gather all the requests and to forward them to the correct users in an act of broadcasting. For better evaluation in a GUI, it attaches agent, conversation and state appropriately to the request lists. The final request format to be sent to the user’s GUI will look like this

```
<input-requests> -> (<input-request1> <input-request2>...)
<input-request>  -> (<agent> <conversation> <state>
                    (<alternative1> <alternative2> ...))

<agent>          -> <symbol>
<conversation>   -> <symbol>
<state>          -> <symbol>
```

The process of observing conversations and asking user stub agents for requests goes on and on as long as there are user participating in the multi-agent application and conversations ongoing.

5.2.6 Visualizing Input Requests

As we have seen in the previous section, one user may receive at any moment different input requests from different conversations and/or different input requests for the same conversation. Before turning to the visualization of a single request, we first need to visualize the received lists adequately. It is essential not to overwhelm and confuse the user with such an “unstructured” plethora of requests at the same time. However, in using the complete

request information provided by GenUA, they are simple mechanisms to sort and display them in a way which is easy to grasp by the user.

For input requests from different conversations or conversation instances in different states, the user needs to select first, which "dialog to turn in". For example:

AGENT	CONVERSATION	STATE	TYPE
Desktop-Agent	Document-Publishing:1	Ready-To-Broadcast	Input
Information-Agent	Document-Search:3	Criteria-Definition	Decision

Here, we are in an office application and have got two requests from different agents and conversations, and the user may now decide whether he first wants to deal with publishing one document (here: to specify the recipients) or searching another one.

We do not pose a particular order of working on input requests. In using expressive specifications, the user itself will be enabled to select the next one adequately, knowing that either nothing will happen to the conversation object concerned until he answers the request or, after a possible timeout, the conversation does something on its own making the request obsolete.

Once inside a request pertinent to a conversation, the user will have either only one possibility to respond or several alternatives. The former situation is that of a simple input action, the latter one a decision action which in turn may need input as well. However, we just display all of the requests in a separated list giving the user the chance to select and inspect all of them, and finally to fill and submit one of them.

For example, when the user chooses the request for searching a document, he may get a list like that:

```
AGENT:      Information Agent
CONVERSATION: Document-Search:3
STATE:      Criteria-Definition
ALTERNATIVES:
```

-
1. SearchByTitel 'Search for documents that match a titel'
 2. SearchByAuthor 'Search for documents from an author'
 3. SearchByKeyword 'Search for documents that contain keywords'

The user will need to decide, which search algorithm he wants to apply, and then he can specify the corresponding criteria. Also, the user must be allowed to go back, and answer another request first.

Now, let's turn to the visualization of a singular request. Each request is one instance of the pattern grammar 5.2.4. All the elements used in the grammar can be transformed

into graphical representations which allow the user to compose input intuitively and guarantee correct input types for values. We have identified four general modal dialogs to be hierarchically nested due to the request composition.

The Top Pattern Dialog is meant to present the “*pattern*” component in form of a dialog window where the key-value pairs are transformed into a list of label-field components. Fields representing simple structured variables (such as symbols, integers etc.) are made editable and show default values if they exist. For every simple field, there will be a check function on the correctness of the value. A user should not leave a dialog until all fields have a correct value. All incorrect fields may be shown to the user, when he tries to do so. Fields representing complex variables (such as nested patterns, lists etc) will appear as buttons which pop up another dialog window displaying the “inner” structure. Fields representing fixed values are made non-editable and show the value just as it is. Furthermore the top level dialog should allow the user to browse the entire request composition with default values plus the new values provided by the user. When the user has completed the request, he can submit it directly to the agent environment in order to get evaluated.

The Pattern Dialog’s job is to represent every “inner” pattern associated to a variable following the same procedure as above. It may allow browsing, but no submitting.

The Single Type List Dialog is dedicated to the composition of lists where each of the elements has the same type. Those types may range from simple structures (strings, numbers etc) to complex types (patterns, lists). Moreover, the single type list dialog, should keep track of the number of elements required. Typical list manipulation functions such as “add Element”, “delete element”, “modify element” have to be provided. Depending on the complexity of elements to be manipulated, the user may do that within the Single Type List Dialog or he will get another nested dialog (Pattern Dialog, Single Type List Dialog, Multiple Type List Dialog) for more complex elements. Default elements should be automatically added to the list. Possible elements may be displayed in a separate choice box in order to get selected and added to the list. A user must not leave the dialog until the required number of elements is provided (if a number is specified). Otherwise he can stop to add elements at any time.

The Multiple Type List Dialog allows to represent and compose list of different values needed without having a pattern, for example a structure like “(!x (list *string* *integer* *string*))”. Other than that, it works pretty much the same way as a Pattern Dialog.

For example, consider the composition of the keyword search alternative. The original structure may look like this:

```
( 'Search for documents that contain keywords'
  (SearchByKeyword
    :sender alison
    :receiver Information-Agent
    :conversation Document-Retrieval:3
    :content !criteria)
  ((!criteria (listof !keyword-op)
    (list (list and 'students' 'employees'))))
    (!keyword-op (list *symbol* string* *string*)))))
```

The idea is to define a keyword search as an (arbitrarily long) AND-ed list of binary or unary logical operators applied on string elements. We assume, that users are familiar with pre-order logical operations. The default value may serve as an example.

As being implemented in our Java applet, this syntactic description can be transformed into a dialog structure as illustrated in figure 5.3.

On top we see the *Top Pattern Dialog* showing the comment, the “ipattern” element and its composition. By clicking on the button “Define List #0”, we browse the nested element, which is mean to be a list of binary or unary operators applied on keyword strings. We display that as a *Single Type List Dialog* with typical list manipulation functions such as *add element*, *delete element*, *modify element*, and *clear list*. In the green choice box, we have the only default element which can be inserted directly if needed. We have already added some elements which are shown in the middle.

By clicking on “Add Element”, we display the structure of a single element as defined above. This is a list of multiple elements without having an explicite pattern structure, so we got a *Multiple Type List Dialog* without explicitly named labels. Beside each field, we show the expected value for orientation. Of course, there is a checking function for each field as well. The dialogs are all modal, which means, the user will have to compose a correct element first before it is added to the list and if the list is completed, the content is associated to the key element where it has been invoked from. Cancelling a dialog does nothing but closing the dialog window.

If everything is specified, the user may browse from the *Top Pattern Dialog* the entire composition and then submit the completed input back to the conversation concerned.

5.2.7 The Notification Grammar

While notifications are somehow restricted to some kind of textual information, results of a conversation’s execution may be taken out of the entirety of different formats for text, graphic or sound. However, GenUA was not devised to care about the content of results in any way. It is only meant to forward results and notifications to the user just the way they had been constructed in the agent execution environment. Presenting the content adequately to the

Message Composition

Search for documents that contain keywords

SearchByKeyword:

sender:

receiver:

conversation:

content: Press please!

Legend

Name	The name of this form	Key	The key for a value
Value	Field with predefined value that cannot be changed	Value	Field to enter/modify a value (may contain defaults)
Define	Button to compose a complex value	Type	Type description for value expected in a field

Single Type List Composition

content:

You are supposed to provide an arbitrary number of "LIST" elements.

Incomplete: Defaults:

[AND, "intelligent agents", "end users"]
[NOT, "software robots", ""]

Multiple Type List Composition

content:

You are supposed to provide an element of the following structure:

Field No. 1	<input type="text" value="Of"/>	"SYMBOL"
Field No. 2	<input type="text" value="user interface"/>	"STRING"
Field No. 3	<input type="text" value="interface agent"/>	"STRING"

Figure 5.3: Visualization of Input Requests

user, is the responsibility of the GUI. Accounting that, we stipulated a general format for an message from a scenario to a user, which can be used for both simple text notifications as well as complex results. This grammar corresponds to the pattern grammar 5.2.4 used for specifying input requests. Again we use the EBNF notation.

```

<message>      -> <msg-type>
                  :sender <sender>
                  :receiver <receiver>
                  :conversation <conversation>
                  :comment <comment>
                  :content-type <content-type>
                  :content <content>
                  :date <date>

<msg-type>     -> <symbol> ;; an arbitrary message type, e.g. in KQML-
                  style: 'Tell', 'Propose', 'Confirm' etc.

<sender>       -> <symbol> ;; the agent who sends this message

<receiver>     -> <symbol> ;; the user's name

<conversation> -> <symbol> ;; the conversation where the message
                  was generated

<comment>      -> <string> ;; any kind of short textual description
                  for the following content

<content-type> -> <symbol> ;; a descriptor for the type of content.
                  may be used by the GUI to trigger the
                  corresponding evaluation procedure
                  for the content
                  e.g. 'text', 'chart' , 'gif', 'fax', etc

<content>      ;; the actual content which can be any
                  type of data

<date>         -> <string> ;; the date when the message had been
                  generated
                  e.g. "Tue Oct 7 17:33:00 EDT 1997"

```

5.2.8 Detecting, Forwarding and Visualizing Notifications and Results

Any conversation rule may specify in its action part such a message and transmit it to a stub user agent. Those provide a kind of message box. If the rule becomes active during the conversation's execution, the message will just be delivered there. Again, GenUA continuously polls all stub user agents' message boxes for received messages and if it detects one in there, it will be forwarded exactly as it is to the user's GUI concerned. In this way, any user may get any kind of message at any moment of the system's execution from any conversation.

Analogous to input requests, notifications and results can be submitted to the user from anywhere in the application at any point in time. The structure proposed in 5.2.7 allows at least to provide a context for the user in order to associate received messages to the execution state of the application. We do not pose the application to confirm any provided input or to propagate all changes continuously to the user. This should be left to the designer of the application. So we just display every message in the order it has been received, and give the user the chance to select each one of them at any point in time in order to inspect it closer.

For example, we may be involved in an industrial team forming process arranged by a mediator, where we first announce our principal interest and then commit ourselves to join the team.

The accumulated messages during some steps of the process may appear on the screen this:

	AGENT	MESSAGE-TYPE	CONTENT-TYPE	COMMENT
4.	TRANSP1	TELL	CHART	'Proposed activity scheduled'
3.	TRANSP1	ANNOUNCE	TEXT	'Small team proposal'
2.	TRANSP1	TELL	CHART	'Large team joined'
1.	TRANSP1	ANNOUNCE	TEXT	'Large team proposal'

The last received message appears on top of the list. If the user wants to inspect message no. 1 (again), he just selects it from the list. Suppose the message has a structure like this:

```
(ANNOUNCE
  :SENDER TRANSP1
  :RECEIVER ANONYMOUS{97}
  :CONVERSATION FLT45
  :CONTENT-TYPE TEXT
  :COMMENT "Large team proposal"
  :DATE "TUE OCT 21 18:11:55 EDT 1997"
  :CONTENT ("LOGISTICS asks for interest in the following activity:"
    (LOGISTICS-ORDER
      :EXECUTOR (TRANSP)
```

```

:LINE-ITEMS
  (( :START-DATE 25
    :DURATION 3
    :OPERATION TRANSPORT
    :ID INV2
    :PRODUCT NUTCRACKER
    :DATE 0
    :DUE-DATE 30
    :QUANTITY 1
    :PRICE 8
    :UNIT-TRANSPORTATION-COST 0.25)
  (:START-DATE 14
    :DURATION 4
    :OPERATION TRANSPORT
    :ID INV1
    :PRODUCT GARDEN-GNOME
    :DATE 0
    :DUE-DATE 30
    :QUANTITY 1
    :PRICE 10
    :UNIT-TRANSPORTATION-COST 0.25))))

```

The essential character of messages to users is, that in principal arbitrary data formats can be contained (as long as they are somehow convertible into a string or bit representation). However, the message header is always the same. On the GUIs side, we just need a handler for each “:content-type” that evaluates and displays the actual “:content” adequately.

In reality, the sample structure above will be received as one single line. Thus, we just need a format routine here for content type “TEXT” that can transform complex list structures. In this way, we are able to combine textual statements and values at will. The visualized message may look like in figure 5.4:

Handlers for other data formats may replace the text field in the middle perhaps with a chart or a gif picture, or with a button that opens a postscript browser for the content, whatever is needed.

5.3 Further Properties of GenUA

Besides integrating users in the agent execution process, the Generic User Agent needs to provide a number of features and functionalities that allow for operation in a collaborative Web environment. Such guidelines can be derived essentially from related literature on WWW services, multi-agent applications, interface agents and human-computer interaction.

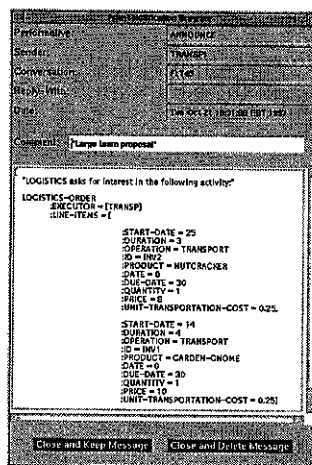


Figure 5.4: Visualizing a User Message

Because of the extensive ongoing discussion in this domains and the overwhelming amount of publications, we could not come up with an optimal model for integrating the interface agent in the Web domain. However, we have identified basical principles and implemented them in our solution.

5.3.1 Generic approach - a trade-off?

The design of GenUA is aimed at building a viable application-independent *template*. How can such a template be constructed? Swaminatham et. al. [24] have proposed the following method: "The locality that typically exists with respect to the purview, operating constraints and objectives of a business entity can be captured in different classes. By incorporating them dynamically into a generic agent architecture, a library of predefined agents emerges that just need to be instantiated for a particular entity". However, in the context of being employed in an enterprise environment, a *trade-off* arises: On the one hand, the interface agent has to meet the requirements of the end-users in order to become accepted in work environments. Hence it should be based on user analysis in terms of user characteristics and tasks, and the user's conceptual model has to be incorporated. On the other hand, a multi-agent system for an enterprise is very likely to be built in a bottom-up fashion from multiple built and pre-existing modules.

We attempt to overcome this problem by supplying an open architecture and general mechanisms needed for the online-access of employees to multi-agent systems in enterprises. The design of the interface agent is completely separated from both the purpose of the administrated scenarios and the graphical representation. GenUA is comprised of a set of independent modules working on general data types without any semantical notion. The

modules have clearly structured interfaces to the rest of the system and just need to be implemented and linked towards the requirements for concrete workplaces. Additional functions can be easily introduced in the template modules. Graphical presentation systems may display GenUA output and capture GenUA input according to the preferences and needs of end users.

Another aspect of generality is, that we delegated the responsibility for implementing the task and interaction structures required for a particular user completely to the designer of the multi-agent system. This includes, that the COOL application may be populated by a number of personalized “desktop” agents which by means of adequately structured conversation plans may reflect exactly the conceptual model of the end users. However, GenUA does not pose the need to have such agents in a scenario, rather it enables end users to talk to any kind of agent in the community. In turn, the agent execution environment makes sure that all the agents can communicate and coordinate their actions without needing to be mastered by end users.

5.3.2 Multiple Connections and Parallel Execution Mode

GenUA is neither limited on association to a single end user only nor to assist a community of end users by default. It is also not limited in the amount of COOL multi-agent applications being accessible to users at the same time.

GenUA administrates the association between end users and applications in a way that every user may access many scenarios at the same time, and many, in turn, users may participate in each of the applications. By means of configuring a simple file, users and the scenarios they are allowed to access can be declared. GenUA provides the necessary authorization and administration mechanisms for both users and applications.

Figure 5.5 shows possible combinations of applications and end users deemed to be facilitated by GenUA.

Another key feature supplied by GenUA is the ability to manage *asynchronous, multi-threaded interaction modes* between users, multi-agent applications and the agents within. Once linked to a scenario, a user may create as many “conversations” to and among agents as he wants by initiating conversation classes. A single locus of control allows the end user to switch between various ongoing conversations and to *focus his attention* on a specific one at any point in time. Ongoing interactions with a particular application can be suspended at will in order to be resumed later on.

5.3.3 System Transparency

A major challenge for getting agent systems accepted by prospective end users is to convey transparency. Following the recommendations of Hall et. al [16] and Sanchez et. al.[19], GenUA meets this requirement initially as follows:

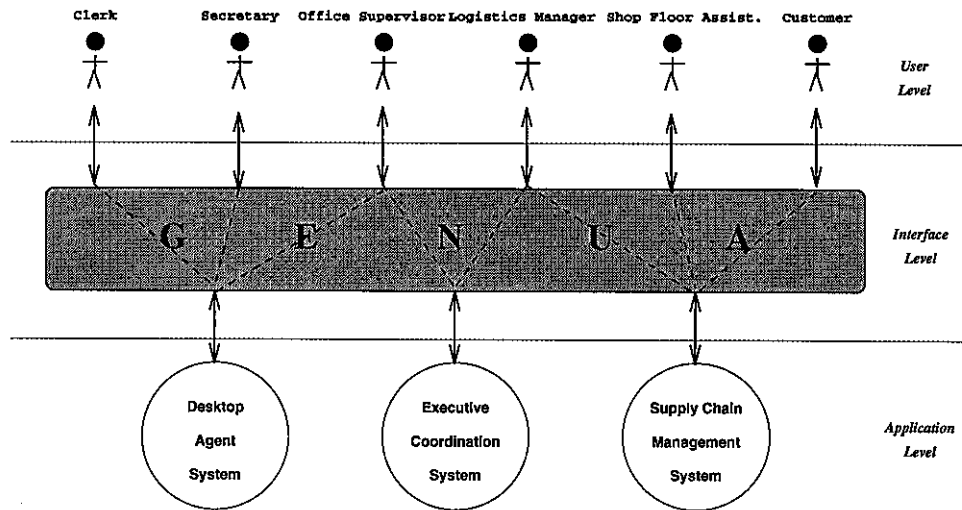


Figure 5.5: User Application Mapping

Cognition: The end-user will become aware of its place in a distributed network and of its interaction with a system of cooperating agents.

Accessibility: The end user can browse and inspect all the components within the multi-agent application, he is concerned with (agents, conversation classes, own conversations), at any point in time.

Control Information: Conversations with the agent system are created expressly. The mechanisms provided by GenUA guide the user through the entire interaction process and supply continuously information on conversation progression (what is the current state of execution), conversation proceeding (which agent works on a user's action), and conversation status (what are the results so far).

History: All interaction information within a session of a user to an individual multi-agent applications is recorded and can be inspected during the session at any point in time. If so desired it can be persistently stored and re-used in upcoming sessions.

5.3.4 Autonomous Activity and Offline-Management

For interfacing a multi-agent system in real enterprise scenarios, there is a strong need to separate the presence of the interface agent from the online presence of the user. Otherwise, there would be no interlocutor for scenario agents, if a user is offline at the moment. Consequently, the execution of the agent will stop as soon as user interaction is required. As a

result, with time the process of the entire multi-agent chain might breakdown, since agents participating are waiting for results or requests of other agents.

This impacts the activation and execution concept for the GenUA and reinforces a strict separation of interface agent and graphical components. GenUA instances run as processes *independently* of the invocation or presence of any presentation system.

However, GenUA must be capable to tackle the problem of (non-) presence of an user adequately. We integrated a comprehensive *session management* concept which allows the user flexible participation in multi-agent applications where he can resume his previous state and can keep track about important changes and requests during his absence via multimedial telecommunication messages.

5.3.5 Customization and Adaptivity

In order to become accepted in a real working environment, any support system must not only fulfil the functional requirements for assisting and guiding a user through his daily tasks in a standardized way. The high expectations and, at the same time, the fear of humans towards application systems are determined by the extent of individual control and customization the software can provide. This is especially true, for every piece of software that comes up as "autonomous agent" and much more for a distributed multi-agent scenario, where components may run somewhere remotely without direct control or even without the knowledge of the user. Moreover, business and enterprise domains undergo continuous changes in organizational structure, task models, working place descriptions and also humans. Consequently, an interface agent in such settings will always be subjected to new requirements. The key for software systems is to provide adequate mechanisms for *customization* and *adaptation*. As the Generic User Agent is targeted to get employed within the same context, we incorporated a level-wise *configuration and adaptation* concept:

COOL Agent Customization: Any COOL agent may be turned into a personalized assistant which supports preference administration, adaptive behavior or even learning capabilities by means of specially designed conversation plans. The need for *adaptive user agents* in multi-agent applications is seconded by the fact that it is almost impossible to extract all the knowledge needed beforehand for bundling into appropriate user agents, considering the wide-ranging area of business applications towards multi-agent systems and the prospective variety of different people acting with them. GenUA makes sure that users can configure their individual agent in exactly the same way as they perform dialogues to other agents during the system's execution. However the rest of the agent's community behaves, the end user and "his" agent will be always connected through GenUA.

Interface Agent Customization: GenUA comes up with a set of module templates which can be easily implemented, extended and linked towards the requirements of an indi-

vidual working place or a group of end users.

Presentation System Customization: Continuous changes in look-and-feel of end users and state-of-the-art in GUI development make it hard to stipulate a universal presentation style or platform for multi-agent applications. GenUA does not pose the utilization of a particular graphical interface type. The basic functionalities identified for GenUA operate on data streams at a high level of abstraction from any kind of display or generation format. Though primarily designed for Web access and coming up with a sample JAVA applet/application, GenUA's driver concept allows to link a variety of different presentation systems. The association between a user and its graphical interface in order to talk to different scenarios is made only at login time according to the configuration. Here, GenUA allows not only an association at an abstract level between user and GUI, it is possible to manipulate the appearance of the GUI itself by evaluating raw preference data attached to the GUI's name. How name and preferences are specified syntactically and evaluated semantically depends on the GUI system chosen.

5.3.6 Distribution and Communication

Total distribution and universal communication mechanisms had been cornerstones for the design of GenUA. We envisage distribution at several levels:

1. *Distributed Users:* The users of GenUA may reside anywhere in the network. The network may refer to an Intranet or the Internet. Users can access GenUA, and consequently all its administrated multi-agent applications, worldwide simply through a browser or another graphical platform.
2. *Distributed Interfaces:* Different GenUA instances may be employed within distributed domains. For example, an enterprise may have one nation-wide Intranet in Germany and one in Canada. The GenUA implementation for Germany may provide an extended functionality, while each of the GenUA instances is accessible through a browser only from within the corresponding Intranet. Nevertheless, both of them can potentially access the same multi-agent applications across the Internet. The JAVA implementation allows full portability across heterogeneous platforms.
3. *Distributed applications:* A COOL multi-agent applications is composed of a number of agents associated to one or more agent execution environments. Each of these environments (and thus the agents) may be executed locally or remotely. Agents within an environment communicate with each other and coordinate their actions remotely in the same way as they do locally. GenUA keeps track of changes in all the distributed components of a scenario, and ensures to relay requests from and responses to users adequately.

Figure 5.6 illustrates a possible structure for users, GUIs, GenUA instances, multi-agent applications and agents across the network.

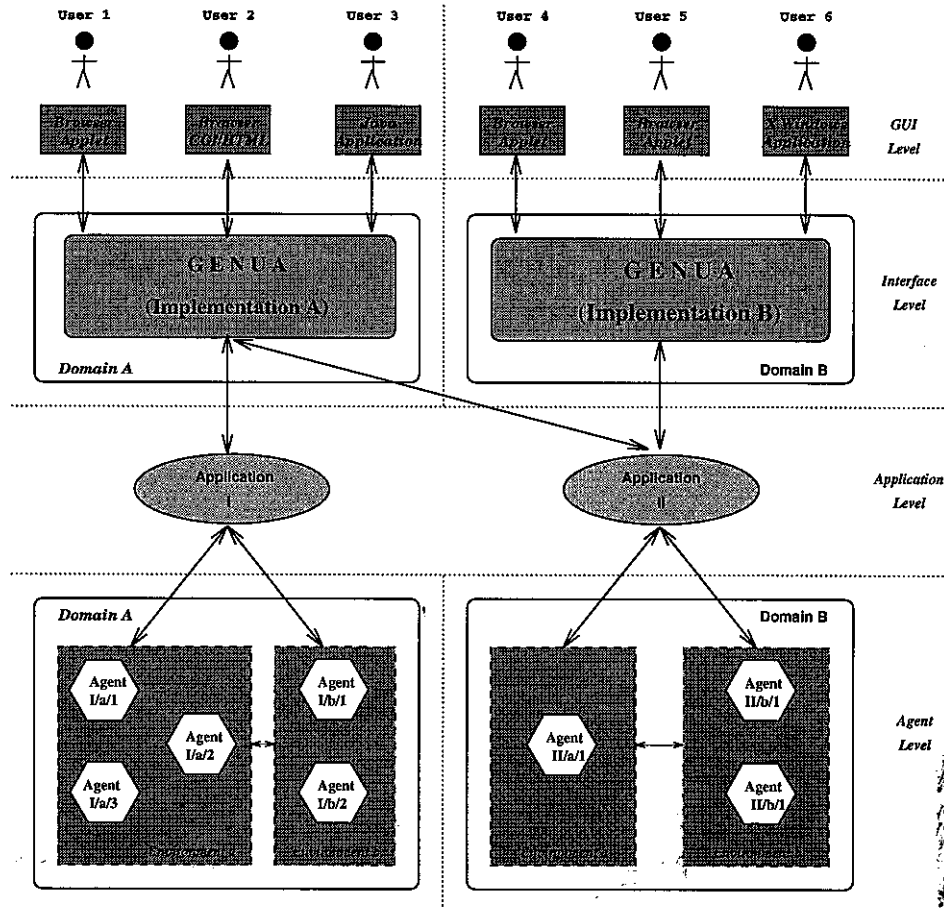


Figure 5.6: Distribution and Communication

Irrespective of all the distribution “below” it, GenUA links agents and applications towards a single virtual platform, being uniformly accessible through a single interface. The communication channels among the components are as follows:

1. *Agent To Agent:* As described in section 4.1.2 COOL agents communicate within and among execution environments by exchanging KQML-style messages. Local communication is ensured by the execution environment itself while communication among environments is done via TCP/IP socket connections.

2. *Execution Environment to GenUA*: GenUA communicates with the individual environments by using the same TCP/IP interface mechanisms as utilized for remote agent communication.
3. *GenUA to Browser/GUI*: Though primarily aimed to be accessible from the Web, GenUA does not pose to do so. The system incorporates a driver concept which allows access from a variety of different GUI components ranging from JAVA applets/applications via CGI/HTML to X widgets (see communication and data format).

5.3.7 Authorization and Security

Sanchez et al. stated that a user (interface) agent should “permit free agent operation while preserving data integrity and user privacy” [19]. GenUA address this requirements on the one hand by sophisticated login and access mechanisms on both system and application level. This includes a general authorization for the GenUA system itself, an authorization for start and shutdown of applications and an authorization for participating in running applications. On the other hand, every interaction from and to the system is a self-contained and personalized transaction. Users cannot inspect or manipulate dialogues or interactions from other users directly, unless the application itself is designed to allow this. However, they will always encounter the interaction’s (impersonal) effects in the multi-agent scenario resulting in changes of own dialog states, notifications, input requests etc.

Chapter 6

System Architecture and Functionality

The last chapter has described the properties of the Generic User Agent and how the functionality had been tailored both towards the requirements of end users and the technical features of COOL multi-agent systems. We will now turn to the concrete realization aspects with respect to how the mechanisms had been captured in a flexible and open architecture and where the individual functions are produced. Furthermore, we will present a sample for a presentation system: a comprehensive Java applet running in a Web browser which allows users to interact with multi-agent systems through GenUA.

First, the *Generic User Agent* will be described in detail 6.1. This includes essential architectural aspects and the examination of the main components. The next section 6.2 treats the realized graphical interface for GenUA, which one we named *COOL User Interface*. By means of a sample session, it will be shown, how a user can interact with a multi-agent application through the new interface.

6.1 The Generic User Agent

The Generic User Agent is the mediator between users and multi-agent systems. It is supposed to allow concurrent interactions from multiple users to multiple applications. For this reason it has to provide both an intelligent administration concept and sophisticated routing mechanisms. At the same time, it should be flexible in its configuration, so that different instances can be build for different requirements. We devised an architecture for the interface agent deemed to be appropriate to satisfy those demands.

However, before we present the individual components and their functionalities in detail, we are going to outline the principal ideas underlying the GenUA architecture.

6.1.1 Towards an Open and Flexible Architecture

Strict Modularization

As an interface agent, GenUA is a self-contained, identifiable and autonomous unit from the perspective of its environment. From inside, it can be divided into a set of entities, which are encapsulated in an agent shell. GenUA is meant to be composed of a *set of independent components* which are *responsible for different tasks* in the context of user agent interaction. A component may need the capabilities of other components to do its tasks, which means components will need to *interact* with each other. It can be seen as a container for a set of functionalities and resources satisfying the component's tasks. Those elements are encapsulated inside the component they are not directly accessible. If every component comes up with the same interface, they can be connected and interact directly without problems.

Even though basic components for GenUA have been identified and implemented, one cannot be sure, if future requirements do not make additional components necessary. Another factor is, that the functionality of existing components, or even the responsibilities of a set of components might be subject to changes with respect to the needs for different organizations. For this reason, a fixed-wired interaction mechanism between components is unfavorable and hard to realize. A more flexible approach is that components do not know each other explicitly, and forward their requests to a *meta component* ("*name server*") in the first place. This one maintains a list of all processable "inputs" for each component. The list will be created while the server instance is initialized by having each component registering its possible inputs. However, the list itself is not static as components may register new inputs or unregister previous ones at runtime.

It may well be that several components specify a similar input. In this case, an incoming request will be routed by the name server to the component which had registered the most specific input related to the request. However, if this component realizes, that it cannot work on the request, it may send it back to the name server upon which this one routes it to the next candidate. In case that several components had registered an absolutely equal input, we have automatically a broadcasting mechanism. Inside the component, there is only a singular handler needed that maps incoming requests onto the component's functions and resources.

To understand the realization of the described mechanism, one may read the paragraph above once again while replacing "request" with "speechact" and "input" with "pattern". *Speechacts* allow a high-level abstraction mechanism for defining requests and responses in an intuitive and familiar manner. E.g. if one component receives an "ask"-speechact about a fact, it will try to retrieve the fact. If it finds something it will send a "reply"-speechact back, otherwise it may perhaps send a "sorry"-speechact stating the reason for failure, and so on. On the other hand, component's inputs registered in form of patterns allows the name server easily to identify candidates to work on an incoming request by *matching* it against each input pattern.

Basically, we have mapped in this way the coordination concept of a *federated agent community* as described in section 2.2.2 onto the composition of the Generic User Agent itself. The components correspond to the facilitated agents while the name server is the facilitator that enables the components to interact with each other.

The architecture becomes both flexible and extendible. New components can be added easily to GenUA while existing ones may determine their capabilities dynamically. A GenUA instance can be build following the principle of a construction set: First, we pick the frame elements (components) needed, then we fill each frame with contents (functions) and finally attach them to the frame (input patterns and handler).

"On-The-Fly" Interactions between Components

Components can indirectly use the capabilities of other components to fulfil their tasks. This means, that it is not predefined which component really executes a request in order to avoid fixed coupling among the components. Precondition is the existence of a meta component which takes on the routing job and that each component informs this about its capabilities in detail.

The capabilities of a component (its processable input) are propagated to the router by means of a set of services either, when the component is started or at runtime. A service is composed of a speechact pattern where the attributes can be fixed values, variables or wildcards and a priority for this service. The router attaches the component's reference to each service and inserts it into a global service directory.

From the name server's and a component's perspective, it does not matter whether the meaning of an incoming speechact is that of a request or that of a response, both always follow the same procedure to work on it. Thus, we will replace both terms by the notion of an "event" in the next paragraphs.

When the router receives an event in form of a speechact, independently whether it comes from outside the server or from one of the components, it always tries to find a matching pattern in its directory. If it finds one, the request is routed to the component. If there are several matching pattern, the speechact will be routed to that component which had registered the highest priority. If even this is equal, the speechact is broadcasted. If no handler can be found at all, the router notifies the sender appropriately so that it can take the necessary actions.

On the components side there will be a general handler which is invoked by the router whenever a matching event for this component has been received. The general handler will trigger special handlers inside the component which process exactly one particular event. In turn, a components may want to send a response after processing an event. Again, this will be a speechact which is passed to the name server and the procedure continues.

The routing mechanism can be simplified for events which expects an answer, i.e. "ask"-speechacts, by remembering the :reply-with attribute and the sender of the event. If an

answer, i.e. "reply"-speechact, has been received by the router, from wherever, where the :in-reply-to attribute is set to exactly the same value, the answer can be directly routed to the sender of the original request without pattern matching.

Abstract Interface Agent Functions

Each component enjoys, independently of its actual purpose or functionality, the following properties:

- specification of its capabilities
- registration/unregistration of services at runtime
- enqueueing requests according to their priority into a "to do" list
- handling incoming requests at an abstract level
- running as an independent light-weight process

Components and modules that acquire "knowledge" during the lifecycle dynamically, are enabled to archive important data persistently and to reconstruct it in case of restart. Persistency refers here to pure data persistence, which means to store the content of variables durable. State persistence, which means to freeze a component or a module in its current execution state and to resume it later is not generally needed for all of them.

Every component or module is enabled to leave a trace of its actions. Tracing can be done in three different modes: (1) tracing on standard output (2) tracing into a file or (3) no tracing. Potentially, every component may operate in a different trace mode, may use different trace files etc. In this way, we have a highly flexible mechanism to observe and inspect the variety of actions.

The Generic User Agent is a wrapper for all its components and sets up a singular identifiable entity to the environment. The environment in our context consists on the one hand of a number of users talking to GenUA through graphical interfaces and on the other hand of a variety of executable multi-agent applications. Consequently, the interface agent itself provides only the following functionalities:

- setting up an identity by adopting an unique address
- adding and removing components
- starting, suspending, reactivating and shutdown components
- starting, suspending, reactivating and shutdown itself

Also, the agent is the first handler for incoming requests from outside. This is necessary as the interface agent can be controlled by sending corresponding speechacts from a GUI.

Declarative Knowledge Processing

A primary goal for the design of GenUA was to operate on knowledge and data structures which allow both a complete abstraction from the semantics and an uncomplicated transmission and (re-)construction. An attempt to cast the vast amount of context-sensitive knowledge used in the business domain into semantical structures for the interface agent would have lead to an unnecessary blow-up and distraction from focusing on its generality. Consequently, GenUA processes and exchanges *declarative knowledge structures* without thinking about the meaning. The strict syntactical orientation of the interface is one of the steps towards being open to a variety of different presentation systems.

We implemented a subset of the KIF specification needed for our context, resulting in a knowledge library composed of basical data types and a lot of more complex objects. The library can be readily extended by new elements for being processed by GenUA and transmitted from and to presentation systems. Basical data types include lists, strings, symbols, integers, variables, wildcards etc. Complex data types are build from that set and include speechacts, input requests from COOL agents to users, user notifications from agents, etc. All of them can be transformed into and parsed from a string representation which is a uniform notion across heterogenous platforms.

Complex data types are serialized simply as named lists of key value pairs. Consider the following example for specifying a GUI network address.

```
class GUIAddress {
    static String user = "raik";
    static String gui = "\"Java Application\"";
    static String host = "timmins.ie.utoronto.ca";
    static int port = 8000;
}
```

The string representation for this object may look like this (with single whitespaces as delimiter):

```
(gui-address :name raik
             :gui "Java Application"
             :host timmins.ie.utoronto.ca
             :port 8000)
```

Such a structure can easily be parsed and the original object can be reconstructed by checking the header and evaluating the following tokens successively. The list-like structure

and a unique header for each object ensures that complex data types can be nested and combined at will.

Open Communication Platform

Though primarily meant to being accessed from a Web browser, which means with respect to our sample GUI from a JAVA applet, GenUA is expected to provide openness towards different presentation systems. Those may include CGI scripts to create HTML pages, or X widgets or Windows applications. To keep the communication of GenUA towards heterogeneous platforms as generical as possible, we provide only one KQML-based communication interface, which can be connected to different platforms by corresponding GUI drivers. Perhaps, there is a conversion necessary, if platforms do not support the transmission of raw strings. However, similarly to the communication among GenUA components, GenUA is supposed to interact with the presentation system via the same speechact format. In this way, a request from a presentation component can be forwarded as it is directly to the component which satisfies it, and vice versa may send a response to be evaluated by the presentation component.

We devised a speechact object which comes up as an extended KQML notation for a communication message. It is composed of mandatory and optional attributes:

```
(<Performative>           ; 'Ask, 'Tell, 'Reply, 'Evaluate, etc.
  :sender <Address>         ; the sender of the message
  :receiver <Address>       ; the receiver of the message
  :tag <Symbol>             ; short characterizer for the content type
  :content <Serializable>  ; the content of the message
  [:ontology <Symbol>]     ; the context of the message
                           ; default: 'genua-interaction
  [:topic <Symbol>]         ; the topic of the message
                           ; default: 'general
  [:language <Symbol>]     ; the format of the message content
                           ; default: 'kif
  [:reply-with <Symbol>]   ; for messages expecting an answer
  [:in-reply-to <Symbol>]  ; for messages representing an answer
  [:priority <Integer>]    ; the priority for the message
                           ; default: 5
  [:platform <Symbol>]     ; the platform the speechact has been
                           ; or will be transmitted
                           ; default: 'tcpip
```

- *jAddressj* characterizes uniquely where in the network the speechact has come from or should be sent to. As a general rule, speechacts exchanged among GenUA components

have the same sender and receiver, which is exactly the address of GenUA itself. On the other hand, requests from presentation systems will specify their unique "network" address as sender attribute and the address of GenUA as receiver attribute, and vice versa for responses. As we do not pose a specific presentation system, we do not stipulate a specific address format. The detailed specification of the address object depends on how the presentation system can be addressed in the network, it only must be uniquely to relay GenUA responses correctly to it. This is, of course, related to the existence of a GUI driver for that platform which is able to receive and to sent messages to the presentation system used by interpreting the address.

- The *:tag* attribute to typify the message content can be used in many different ways. A component in GenUA or in the presentation system may use it to trigger an appropriate handler for the content. One can define a function name in there, while the content are the parameters.
- The *:content* itself may be anything, in general, but there is one restriction. It must be possible to transform it into a format which can be transmitted through the GUI driver, and vice versa be reconstructable from that format into an object GenUA can deal with.
- *:Reply-with* and *in-reply-to* can be readily used to relate sent requests, i.e. "ask"-speechacts, to received responses, i.e. "reply"-speechacts. This is particularly useful, as we need to handle multiple requests at the same time.
- The *:priority* determines, how fast the speechact will be processed by GenUA or one of its components. Speechacts can be ranked from 0 (unimportant) to 10 (very important).
- Finally, the *platform* is used by the communication component of GenUA to relay a message to be sent to the correct GUI driver.

How speechact interaction between GenUA and a presentation system by means of a driver concept is described in detail in section 6.1.3.

The "Right" Implementation Language

Java offers in contrast to other implementation languages essential benefits for programming agent-oriented and web-related systems, e.g. explicit thread support, automatic storage management, communication interfaces. The strict object orientation explicitly reflects the idea of agents and other system components communicating with each other by exchanging messages. All the benefits of object-oriented software engineering such as inheritance, re-usability or abstraction can be utilized during the development process. Also, an

interpreted language such as Java is more appropriate for development because of the lower turnaround time and a better error debugging.

Java is promoted by many vendors and developers throughout the agent software and web community, and has already become a kind of semi-standard. The major benefit of Java is its portability. Java bytecode can be executed on any machine. As for graphical representation and universal access, Java applets come up with a variety of pre-defined graphical components and are accessible through a Web browser from anywhere in the world.

For those reasons, we have seized on the Java language for the implementing the Generic User Agent and provide a comprehensive Java applet .

6.1.2 Insight to the GenUA Architecture

After this introductory design issues, we move on to have a closer look at the internal composition of GenUA. Figure 6.1 provides a complete overview.

We have identified five main components needed for GenUA to fulfil its tasks:

- *Communication* to send and receive messages to the individual graphical user interfaces
- *Administration* for giving access to multi-agent systems and maintaining user configurations
- *Application Manager* for enabling interactions between multiple users and multiple agent applications
- *History* to record those interactions in a personalized way
- *Offline Manager* to gather application events occurring during a user's absence and informing the user via a telecommunication medium
- *Service Agency* to mediate among the components for interactions

Some of the components employ sub components and resources (indicated by white boxes). The exact purpose will be explained in the corresponding paragraphs following.

6.1.3 The Communication Component

We begin our journey with GenUA's interface to the outer world - the *Communication* component. The purpose of this component is to receive requests from and send responses to not only a variety of different users but also different presentation systems.

For this reason, we have integrated a GUI driver concept into Communication. GUI drivers are meant to be tailored exactly to the addressing and transmission mechanism, a GUI may use. For example, for our Java Applet we utilized a simple TCP/IP driver which

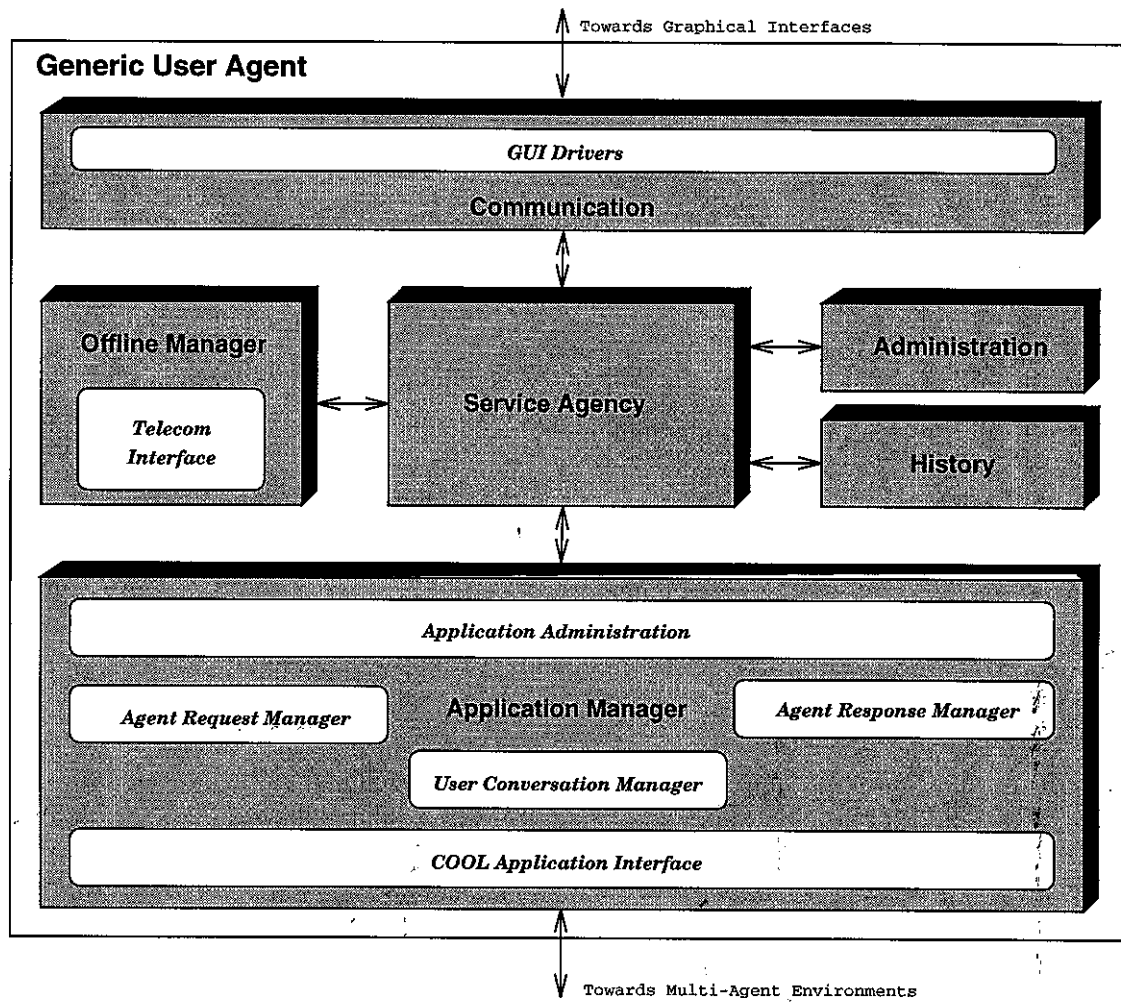


Figure 6.1: The Architecture of the Generic User Agent

one transmits data through a socket to the Applet residing in the Web browser. It is possible to replace this connection with object brokers such as CORBA, or with the HTTP protocol. Instead of the Java applet, it is also possible to have a CGI script running which communicates with GenUA through files or pipes, fetches requests from HTML pages and generates HTML results. If one has a X widget execution environment, he can talk to GenUA through one of the standard inter-process communication.

With respect to the limited time, we could not experiment with more GUIs and drivers than the Applet-TCP/IP connection mentioned. However, the mechanism described above is supported in a generic way. We have initially implemented an abstract description of a GUI driver, which one can be instantiated by any number of arbitrary parameters due to its purpose. Every driver can be associated to Communication and is enabled to send and to receive speechacts in an abstract way. Also, we provide a default driver which operates on the TCP/IP protocol and is active through the lifespan of GenUA, and we have a default address to send any kind of speechacts to if necessary.

As we do not know beforehand, from where and how a user wants to talk to GenUA and have no idea about the parameters needed for a GUI driver to establish a physical connection to that particular GUI, Communication will start GUI drivers only on explicit demand while the request includes the parameters to be passed to the driver's instance. Of course, this has to be made through an independent driver which is active through the lifespan of GenUA. We provide a default driver which operates on the TCP/IP protocol. Upon receiving such an installation request and if Communication finds a corresponding driver, it will be instantiated and administrated so that all further communication from and to GenUA can be made through the new driver. Vice versa, GUI drivers are shutdown upon receiving a corresponding request.

In this way, opening and closing drivers appropriate for their purposes is the responsibility of the presentation component, a user wants to use. We can keep GenUA free from opening lots of predefined drivers which are perhaps never used, or which may need a re-initialization at runtime as the parameters of a particular presentation component slightly differ from the default parameters in a running GUI driver.

Drivers are always associated to unique addresses in the network, so that users at different locations can talk to GenUA through different drivers at the same time. The identification is simply made via the ":sender" respectively ":receiver" attributes of speechacts exchanged.

By means of the parsing and serialization process for knowledge as described in paragraph, it is possible to send and receive data to different platforms in pretty much the same way without extensive conversion.

6.1.4 The Service Agency

The *Service Agency* is the core of the Generic User Agent. With it we realized the meta component or name server mentioned in paragraph which allows all GenUA components to

interact with each other in a flexible and sophisticated manner. It combines routing and brokering capabilities.

While startup, every server component registers with the Service Agency by providing a number of services. A service is a speechact pattern, the server component is able to interpret and to process. In this way, components announce their *capabilities* or *interests in events*. In the following the Service Agency will know about the existence and the capabilities of every registered component and is enabled to route incoming events to a corresponding receiver. Components may "improve" or "reduce" their capabilities at runtime by sending corresponding register and unregister speechacts to the Service Agency. This one maintains and updates all the registered input patterns in a global directory for referencing, and can sent a snapshot of it on request.

The Service Agency is also enabled to distinguish between *internal requests* (received from a component) and *external requests* (received from a GUI), and on the other hand between responses to be routed internally among the components or outwards. As every request and every response is a speechact event, this can easily be done by evaluating the ":sender" and ":receiver" attributes. It is obvious, that requests from outside will always come from the Communication component, and responses to GUI's will always passed to this component for sending them away.

When the Service Agency receives an event in form of a speechact, independently whether it comes from outside of GenUA or from one of the components, it always tries to find a matching pattern in its directory. If it finds one, the speechact is routed to the component by invoking the component's general event handler. If there are several matching pattern, the Service Agency builds a list of candidates ranked by the priority, the components had attached to their service registration. Due to the list, the component's are asked to handle the event until one will work on it. If no handler can be found by pattern matching, the Service Agency tries in a last act to pass the dangling vent in a first-come-first-served order to any component. This is useful, as a component may have "forgotten" to register a new capability at runtime. If no component wants to handle the event, the router notifies the sender appropriately so that it can take the necessary actions.

Events which expects an answer, i.e. "ask"-speechacts, can lead to a simplified mechanism, with respect to an upcoming answer, i.e. "reply"-speechact. In that case the Service Agency simply remembers the event's :reply-with attribute and its sender. Events with corresponding :in-reply-to attribute will be directly routed to the marked sender without pattern matching.

As an additional feature, the Service Agency can be linked to a special network address, where a copy of all events passing the agency's loop can be sent to. This allows for monitoring the entire GenUA activity, for observing the components' interaction, for recognizing deadlocks or even for visualizing the activity in an animated picture.

6.1.5 The Administration Component

The purpose of GenUA's *Administration* component is, as the name indicates, to administrate users, to allow access to the system and to provide user configurations.

Users are uniquely identified by a login name and an encoded password. Those are checked before any other interaction is possible. Associated to each registered user is a profile with the following elements:

- the set of multi-agent applications he is allowed to administrate
- the set of multi-agent applications he is allowed to participate in
- the name of the graphical interface to talk to the applications
- a set of preferences controlling the appearance and function of the GUI

When a user is allowed to access the system, the Administrator sends those four elements back to where the login request came from, in order to get evaluated there. The idea is, that the destination system in turn invokes the user's preferred GUI instance while interpreting the preferences and displaying the available applications. In the next step the user can start to interact with the multi-agent scenarios.

Beside the registered users with name and password, we have defined a special user, who may login as "anonymous" without password check. This guy may get demonstration scenarios and a default interface without preferences. However, interaction with real ongoing applications is only possible, if the application itself allows participation of unknown or new users.

After a user has logged in, Administration broadcasts the online presence of the user to every server component. A component may use this information to load user-related resources, to install tools for him or to make specific information for this user accessible upon request. From this moment on, a user is uniquely identifiable by its name/password AND its current location in the network. An obvious principle used throughout the entire implementation process states: "A user may have sessions to multi-agent applications from different locations but not at the same time. Thus, keep user-related elements across sessions associated to his name/password, and use his current online location for receiving and sending."

User profiles are persistently maintained and updated regularly. Administration allows simple creation, removal and modification of user profiles at runtime via corresponding speechacts. Of course, user profiles can also be edited manually in a simple file format, which is interpreted by the component while startup.

6.1.6 The Application Manager

The *Application Manager* is the entity in GenUA, which incorporates the essential functionalities to link users to multi-agent applications, to enable interactions, to detect input requests

and agent notifications, and to forward them to the correct user. In order to satisfy this task, the application manager employs a variety of sub components and resources. Figure 6.2 gives an overview to the structure.

The Application Manager is the immediate mediator between distributed multi-agent execution environments and the rest of GenUA. On top, we can see the *Main Event Handler* pertinent to every component to handle incoming events when assigned by the Service Agency. This handler interprets the events and invokes the corresponding functions inside the the application manager. It is also responsible to pass speechacts, i.e. responses or notifications, created in the component to the Service Agency for routing to other components or external entities, i.e. the plethora of graphical user interfaces.

Administrating and Acting in Multi-Agent Applications

Attached to the Main Event Handler there is a module called *Application Administration*. It is responsible to administrate and control multi-agent applications. General administration functions include:

- self-configuration by loading the specifications of all multi-agent applications provided
- listing all specifications
- adding, removing and modifying specifications at run-time
- storing changes of specifications permanently

An application specification is composed of a unique name and a description of its distributed elements. This is a list of host-port-resource tuples needed to create and to access COOL execution environments on remote machines.

The more complex functions are those which allow management and control of complete multi-agent applications:

- creation and shutdown of distributed multi-agent application
- linking and unlinking a user to a multi-agent application as well as installing and decoupling the necessary auxiliary tools
- providing information about the main elements in applications which are agents, conversation classes and conversations
- initiation of conversations at agents by the user
- forwarding of user input to multi-agent applications being requested from there at runtime and filled by the user

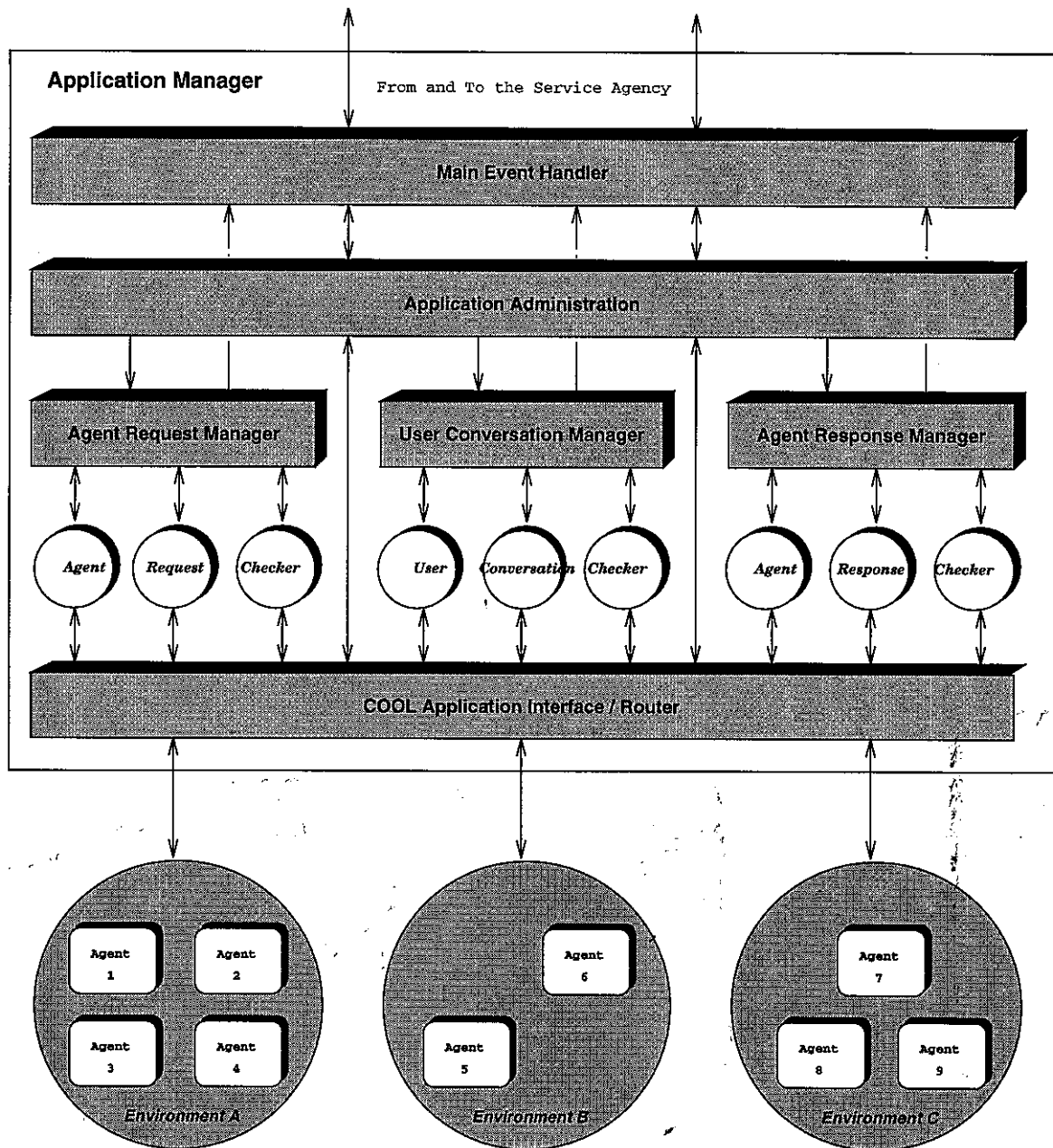


Figure 6.2: The GenUA Application Management

- execution of distributed multi-agent application

To *create* a distributed application means to open up a number of COOL agent execution environments on remote hosts and to load the necessary sources automatically. Only users with a corresponding profile are allowed to create applications.

Linking a user to a multi-agent application is done in two different modes.

1. A user may be "new" in the application. In that case, we create a personalized stub agent in each of the distributed environments. From that moment on, the user will be enabled to access and to interact with the elements in the application, to receive input requests, to submit input and to receive notifications and results from there. A number of auxiliary tools will be installed for the user to keep track of what's going on for the user in any of the distributed environments.
2. We allow users to suspend sessions with applications temporarily and to resume them later on (perhaps from another location). In that case, both stub agent and tools will remain active all the time during the user's absence. Only the operation mode of the tools has to be changed (see below).

After being linked, the user may first *get information* about available agents and their conversation classes. He can select one or more and *initiate* them explicitly. The user may receive input requests, to fill them and *submit his input* back to the application and to inspect notifications sent to him. Even though an multi-agent application usually executes itself without the user's initiative, there are some cases, the user needs to trigger the *execution* of the multi-agent application explicitly.

When *unlinking* from a multi-agent application, the user needs to choose, whether he wants to resume his session with the application later, or if he wants to finish his participation. Depending on that, either his stub agents and all auxiliary tools installed from him are removed, or everything remains active and just operates in an offline mode during the user's absence (see below)

Shutdown an running multi-agent scenario is an easy action in terms of killing all the distributed environments but a tricky problem considering the ongoing interactions of users. First, only those user, which is allowed to create an application, can shutdown it. Second, applications can be shutdown in two different modes:

1. In a *user-friendly* mode, we determine if there are still users interacting with that application (irrespective whether they are online or offline at the moment). Only in the case, where no one participates, the application is really shutdown. Otherwise, we provide a list of participating users to the one, who has send the shutdown request. He may use this information to discuss the issue with them.

2. In a *user-ignorant* mode, we shutdown the application, but we send an explicit message to all the users still participating, so that they are at least informed automatically.

Observing Multi-Agent Applications

Application Administration is connected to three further subcomponents the *Agent Request Manager*, the *User Conversation Manager* and the *Agent Response Manager*.

The managers are frameworks which supervise a number of polling elements needed to keep track of what's going on for every user in any part of distributed multi-agent applications. Those polling elements are created on demand, always associated to exactly one user in one application, and they keep on running as light weight processes until they become obsolete. As the name may indicate, they serve the following purposes:

Agent Request Manager and Agent Request Checkers The checkers continuously watch whether there are any input requests for users in applications. Detected requests are wrapped into corresponding speechacts and passed to the manager. From there they will be forwarded without further manipulation via Main Event Handler, Service Agency and Communication directly to the graphical interface where the user concerned can answer them. Moreover, the checkers keep track which input requests already sent to GUIs have become obsolete for a number of reasons and they will notify the GUI about that in the same way.

Agent Response Manager and Agent Response Checkers These checkers do exactly the same for notifications and results from multi-agent applications to users. Detected notifications undergo the same procedure of transmission to the graphical interface where the user concerned can inspect them.

User Conversation Manager and User Conversation Checkers The purpose of these checkers is to continuously update conversations, a user has instantiated, on the user's screen. As a result, he will get complete instance descriptions which will be updated continuously, so that the user can keep track of what's happening to "his" conversations during the execution process. It is essential to mention here, that this inspection and update service is available only for self-created conversations. The reason is that, as a result of the user's initiative, quite a number of internal conversations may be created inside the multi-agent application a user should not be overwhelmed with. Detected changes include changes in states, modified variables, termination etc. All of them are submitted in the same way to the GUI as mentioned above.

One of the key elements for all of those personalized checkers is that they may run in online or offline mode. This means they behave slightly different depending on whether a user

linked to a multi-agent application is currently online or offline. Particularly, it will make no sense to send speechacts to non-existing GUIs. Rather, they will be deposited at the Offline Manager and transmitted through a standard telecommunication medium to wherever the user wants to receive information during his absence. Those effects are controlled by the Application Administration whenever a user links to an application and due to the mode he unlinks from it.

Accessing and Manipulating Multi-Agent Applications

At a certain point, it is necessary to break all the execution and management logic to simple functions which can be executed towards agent execution environments and whose results can be captured and interpreted. Basically, we needed the following mechanisms

- a mapping from JAVA to LISP for both responses and results
- an interprocess communication between the JAVA virtual machine and an LISP execution environment
- a routing mechanism from and to distributed execution environments

This is exactly the task of the COOL Application Interface. As you can see in figure 6.2, all of the other modules and checkers communicate with the actual agent environments via the COOL API after all. The API provides a number of basical functions needed for our context. There is an API in each agent environment which provides exactly the corresponding functions just in another notation. The individual functions are:

- loading an application into an execution environment
- creating user stub agents
- removing user stub agents
- getting the agents of an application
- getting the the conversation classes of agents
- getting the the ongoing conversations of agents
- initiating a conversation at an agent
- getting input request from a conversation to user
- sending input to a conversation
- getting notifications for a user

- executing an environment

With these little set it is possible to enable concurrent interactions of multiple users with multiple applications.

Fortunately, agent execution environments are accessible via TCP/IP socket connections. As the COOL API knows where the individual environments of all distributed applications reside, it only needs to open a socket there, to send the encoded request, to read the response and to convert it. Again, we could benefit from the declarative knowledge structures mentioned earlier. We have the API in the LISP environment sending results in exactly the same string notation as we transmit responses to graphical user interfaces on different platforms. Aside from parsing a string there is no additional data conversion necessary.

It is evident, that the API establishes the bottleneck of interaction. During the execution there will be not only continuous but parallel queries from one of the Application Manager's elements. We needed to set up a synchronization mechanism in here which allows only one request to be executed at a time in a FIFO order. However, as the socket communication is a matter of milliseconds, this poses no obstacle.

6.1.7 The History Component

The purpose of the *History* component is to record every essential interaction between users and multi-agent applications and between graphical interfaces and GenUA itself. Interactions are captured in form of retrievable events. Every other component of GenUA can create such interaction events and sends them as speechacts to the History component in order to get recorded.

Interactions are always user-related, thus they are maintained in a global user-event table. With respect to one user, events either fall into the category of application-independent events, such as login or creating/shutdown applications and application-interaction events, such as input requests, user input or notifications from applications to a user.

The heterogenous nature of events requires a retrieval mechanism for the history across sessions and across interactions pertinent to particular applications. A user must be allowed to inspect everything that happened while interacting with GenUA at any point in time. In this way, a user will be enabled to re-use results obtained from interacting with one application for interacting with another application. On the other hand, this will lead to a considerable growing of events for a user during the lifespan of GenUA.

We deal with that in the following way. During a session with GenUA we force the graphical interface to keep events persistently on its side, once they are retrieved. Subsequent retrieval requests will only provide those events that happened in the meantime, thus avoiding to transmit information twice. Events pertinent to a application will be removed, if the user has decided to terminate his participation with that application (see "unlinking" in the description of the application manager). Moreover, we create from time to time dumps of

events on a persistent medium if the absolute number exceeds a certain limit. This means, the events are no longer retrievable from GenUA, but they still can be inspected if needed.

However, a snapshot of all events currently maintained in GenUA can always be persistently stored and retrieved.

6.1.8 The Offline Manager

The *Offline Manager* is responsible to inform users about essential changes in applications during their absence. The idea is, that a user may be actively involved in a multi-agent scenario but is momentarily not able to have a session with GenUA. Nevertheless, there should be a way to inform him about requests, results or other changes detected for him during his absence. The Offline Manager is meant to send those events to users as multimedial telecommunication messages (fax, voice or e-mail).

Whenever a user suspends a session with a particular application temporarily, he can define where and how he wants to get informed in the meantime. During his absence, the user-related tools of the Application Manager work in offline mode. This means all events detected are not relayed to the graphical interface but they will end up here. A copy of those events is supposed to be converted into the selected format and transmitted to the selected destination. This means, the internal textual representation will be either converted into a E-mail message and sent to an E-mail address, or a fax message sent to a fax number or even a voice message resulting in an automatic phone call.

We have currently implemented a mechanism for sending e-mails in this way. However, we identified a general interface, which makes it easy to attach handlers for fax or voice similarly without modifying the Offline Manager.

Irrespective of that, changes detected during the user's absence are stored in an application-related manner. This means they can be retrieved when the user resumes his session with a particular application, no matter where he had been located during the previous session. The events will be transformed to the GUI in exactly the same way, as if he would have been online all the time and just would get a large amount of requests and results at an instant.

6.2 The Graphical GenUA Interfaces

Users may interact with the Generic User Agent from a variety of graphical interfaces. However, for worldwide access to multi-agent systems, the best approach is to use the World Wide Web. We anticipate that interacting with multi-agent systems through a Web browser will become as familiar and convenient as using on-line information or searching for documents. Downloading and executing Java Applets is supported by most of the standard browsers. For this reason, we developed a graphical interface as a Java applet. In terms of design, it should be emphasized, that it was neither the purpose to come up with the ultimate graphical

realization nor to address a particular group of end-users. Rather, we wanted to demonstrate the principles of interaction to multi-agent systems by utilizing the features of GenUA.

For our approach in designing the graphical representation, we tried to keep the look and feel as clear and intuitive as possible. The user may operate freely within the elements. Most of the logical work will be done by the Applet. Also, we force the user for specific interactions only if its really necessary in order to leave GenUA in a defined state. We limited communication activities by maintaining most of the objects permanently on the applet's side, once they had been obtained from the interface agent. GenUA poses a strict model for user interactions to multi-agent applications. The applet reinforces this by a context-sensitive event handling. With respect to control and transparency, the graphical interface follows an all-in-one approach which means that all the essential functionalities and all dynamically changing components are bundled into a singular frame. The user will be always aware of the current context and can keep track of the effects his interactions will cause in the multi-agent applications.

The following description of the graphical interfaces starts with the way how they communicate with GenUA 6.2.1. After that, we will examine the individual features by means of a sample interaction with a multi-agent system 6.2.3.

6.2.1 The Communication Handler

The graphical interfaces are composed of two elements: the actual presentation frame and a communication handler. The latter one enables the presentation frame to communicate with the generic user agent and updates the content of the frame dynamically.

With respect to security issues, Java applets are restricted in using network services on the client's side. They cannot write onto the local filesystem and are not allowed to open socket connections to arbitrary hosts. However, they may talk to sockets on the host, from where they had been downloaded. As the applet resides on the same host, where GenUA is running, we can have the applet talking to GenUA via TCP/IP socket connection. For that purpose, we could instantiate exactly the same TCP/IP driver as used by GenUA.

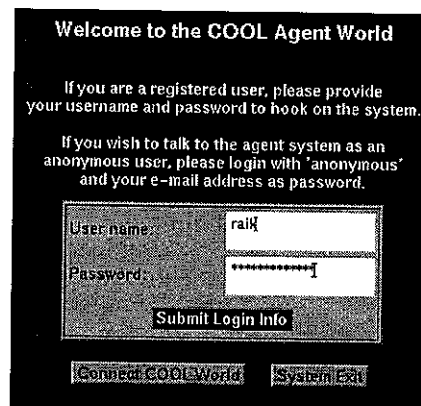
The main responsibilities for the Applet's communication handler are:

- to send requests to GenUA and to receive responses from there
- to receive any kind of notifications, GenUA issues autonomously to the applet
- to convert user actions detected by the components of the presentation frame into adequate speechacts in order to trigger the desired action in GenUA
- to translate speechacts received from GenUA into corresponding manipulations of the presentation frame

It is obvious, that conversion and presentation have to work very close together. However, the conversion layer could be separated from the actual communication layer. This allows for re-using at least the communication facility, if the requirements for the applets; functionality are changing significantly.

6.2.2 The Login Window

In the beginning, we need to identify the user uniquely and to recognize, from where in the system he is talking to GenUA. A user may log to the system from any web browser. Once accessing the URL, he will first obtain a login window as shown in figure 6.3.



The screenshot shows a login window with the title "Welcome to the COOL Agent World". It contains two paragraphs of instructions: "If you are a registered user, please provide your username and password to hook on the system." and "If you wish to talk to the agent system as an anonymous user, please login with 'anonymous' and your e-mail address as password." Below the text are two input fields: "User name:" with the text "rail" and "Password:" with masked characters. A "Submit Login Info" button is positioned below the password field. At the bottom of the window are two buttons: "Connect COOL World" and "System Exit".

Figure 6.3: Logging to the GenUA System

Aside from registered users, which are known to GenUA with name and encoded password, we allow "anonymous" users to use its services. Those users may get specially designed multi-agent applications which offer functionalities to arbitrary users, e.g. communication or information services. To distinguish them from each other, we use their e-mail address as password.

Once GenUA has received this information and the user has been accepted, it determines which graphical user interface should be provided to the user. As mentioned in section 6.1.5 it maintains a profile with an abstract GUI name and preferences for each user in order to be evaluated on the presentation systems side. In our Java Applet case, we transmit a class name and some preferences back to the Login Window upon which the actual graphical user interface is instantiated. So far, we had developed only one representation deemed to be appropriate for our purposes. However, one may feel free to design other GUIs while using a similar execution logic.

6.2.3 The COOL User Interface

Figure 6.4 shows the complete user interface developed for interacting with multi-agent applications as it will present itself upon the user has logged in to the system.

As you can see from the design, we followed the all-in-one approach. On top we have a *control panel* for dealing with application objects itself and the history across application sessions. *Application Administration* refers to the creation and shutdown of agent applications while *Application Linking* refers to participating in and dissociating from ongoing applications.

Below, we have a *browsing panel*. The purpose of that is, beginning from the available agents, to guide a user to find an appropriate conversation class in order to get instantiated.

The next panel deals with *ongoing conversations*. Here you will find a constantly updated list of conversations, the user has initiated from his screen and that are still active. The single elements are inspectable at any point in time.

Underneath, there is the *action panel*. Here, all incoming input requests for this user will appear in order to get answered and filled by the user during the execution process.

Finally, we have an *response panel* where all kinds of notifications and results pertinent to that user will get listed and can be inspected at any point in time. Those results can be browsed in a clear and familiar fashion.

To explain the execution logic and its graphical effects, we will slip into the role of a user and have a sample session with a multi-agent application mediated by the Generic User Agent.

Creating an Application

From the choice box in the upper left corner, the user can create an application. By interpreting the user's profile, GenUA automatically only provides only those applications here, the user is allowed to create and to shutdown. This usually refers to a kind of application administrator. There will be many users who have no right to do that at all, where the choice box is simply empty from the beginning. However, an administrator may create as many applications as he like and as exist. Of course, GenUA allows this process only once for each application. From the perspective of the user, it does not matter whether an application created will run distributed or not. Once an application is up, an arbitrary number of users may participate in, each one with his own GUI, from wherever they are located.

For our demonstration session, we will create a multi-agent application, that runs on a single machine and is called "Supply-Chain+Scheduling" defining a supply chain demonstrator with scheduling elements incorporated.

Linking an Application

From the choice box in the middle, the user can link a particular application which means

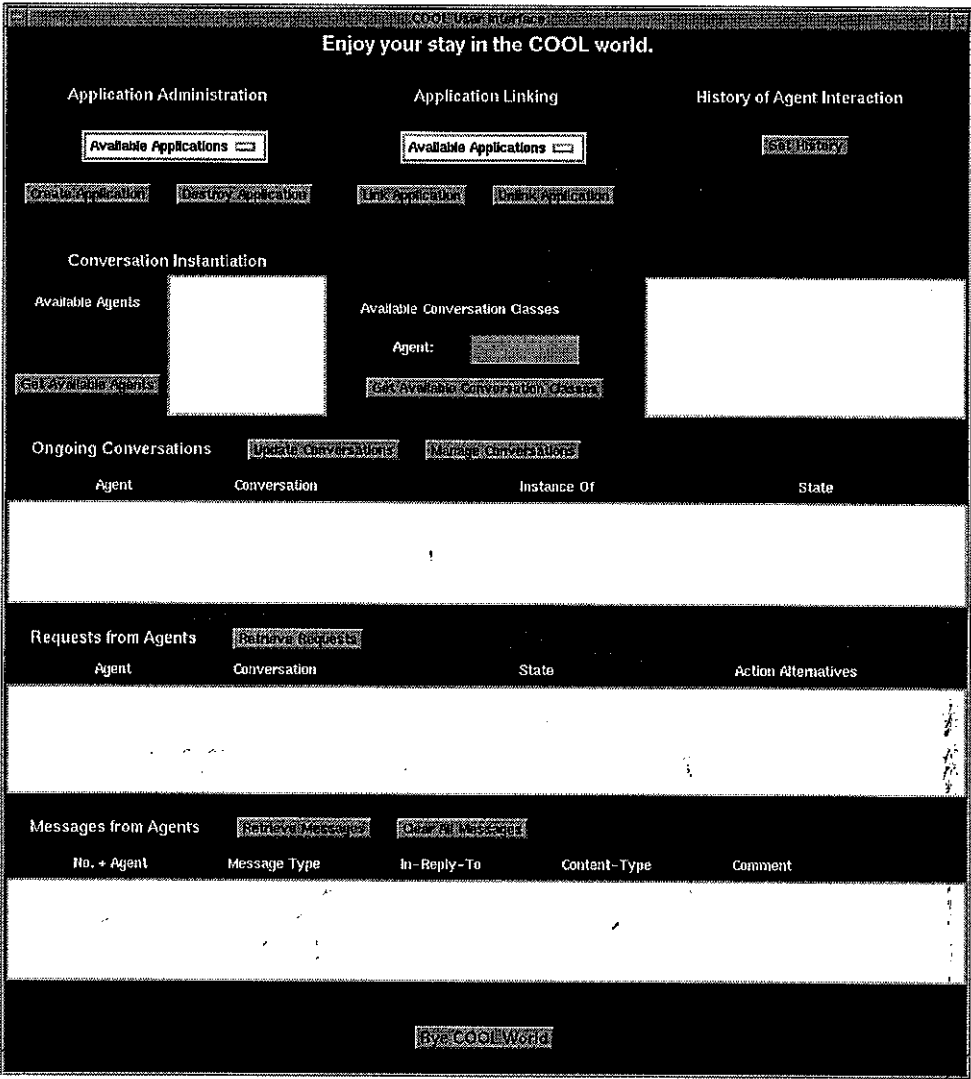


Figure 6.4: The COOL User Interface

he can participate in its execution process. However, we allow active participation only in one application at a time. The reason is to ensure a certain context for the user while interacting. Mixing up elements from perhaps completely different applications in a single window will confuse the user. Constant flipping from one application window into another upon every kind of event issued by GenUA is prohibitive as well. Having multiple windows on the screen do not help the user to focus his attention. Instead of all that, in our approach the user can concentrate clearly on being involved in and interact with one particular scenario at a time.

Becoming a participant in a multi-agent application means that the user will get his own personalized stub agent inside the scenario for browsing its elements, being requested for input, submitting responses and receiving notifications or results from the application.

In our demonstration session, we now let two users link the "Supply-Chain+Scheduling" application each one from his own GUI instance, one "client" that can order products and one "executor" that cares for logistics and manufacturing to satisfy the order.

Navigating through the Application

A multi-agent application consists of numerous objects. However, only a limited subset (1) will be of interest to the user and (2) should be made accessible to users. Essentially, the purpose of browsing in our context is only, to allow the user to instantiate a particular conversation class at an agent.

The question is arising, how a user will find exactly this class which satisfies his needs. Names and object structure may give a hint, but are not sufficient. The point is, that every conversation may spawn or interact with a multitude of other conversations inside the agent environment, the user will be not aware of in the first place, and should also not be overloaded with. Moreover, these interdependencies will be subject to changes as the multi-agent application requires an evolutionary development and constant adjusting to new requirements. However, we kept the issue of searching the "right" service aside for the present, and assume that the user will somehow know with respect to daily experience, which classes he has to use in order to do a certain job.

At any rate, upon linking to an application, all available agents are listed on the left handside. Again, it does not matter where these agents physically reside. From the user's perspective they are all linked towards a single virtual platform. The user can select agents and request its conversation classes which will be listed by name on the right handside, for each agent separately.

As you can see in figure 6.5, we did that on behalf of the "client" for the Customer agent and got its only conversation class - the Customer Conversation.

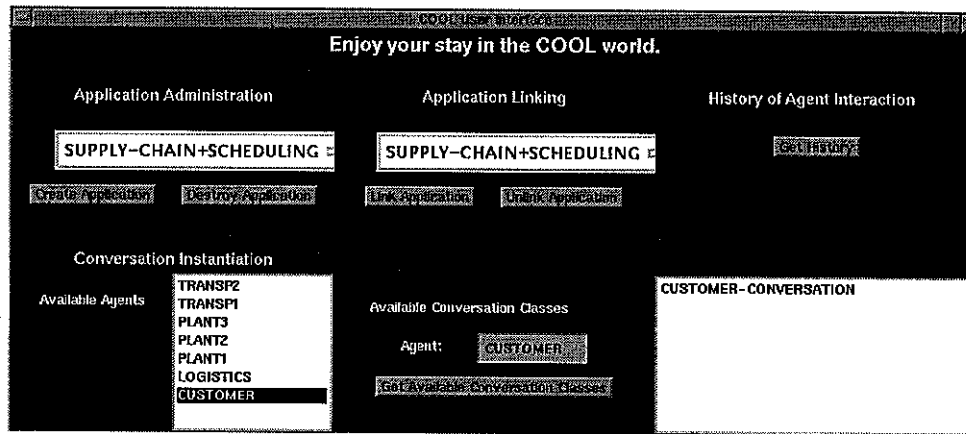


Figure 6.5: Selecting a Conversation Class

Instantiating a Conversation

From the conversation class list, the user may inspect every object in detail. A conversation object will be browsed as illustrated in figure 6.6.

This gives the user an idea what the conversation class is about and how it is structured in terms of states. Due to limited time, we did not implement a graphical representation for a state diagram in here. However, future work needs to convert the elements in the lower box (states & conversation rules) into a state diagram which will be much more expressive than just a textual listing.

From this window, the user may create an instance of the conversation class browsed. The new object will be returned to the user. Moreover, the object will automatically fall into the observance of GenUA meaning it keeps track about both possible input requests the conversation may have to this or other users and changes in its execution states.

As a "client", we have selected the Customer Conversation from agent Customer here. This class is responsible to acquire orders, to forward them to the Logistics agent to take care about its execution, to keep track about its processing state and to assist the client in decision making if problems arise. We are now going to create an instance of this class, which mean essentially that we create a file or workflow object:

Handling Requests from the Application

Once, a conversation has been created, it will get executed inside the agent environment, moving from state to state as defined by the conversation rules, interacting with other conversations, spawning new conversations, doing local actions. At some states, input needed from a particular user may be requested. This is detected by the Generic User Agent in teamwork

Here we have conversation class: CUSTOMER-CONVERSATION
 Associated to Agent: CUSTOMER
 This conversation class allows INTERACTIVE manipulation.

Name:	CUSTOMER-CONVERSATION
Ontology:	CUSTOMER-ORDER-HANDLING
Initial State:	START
Final States:	REJECTED FAILED SATISFIED
Variables:	
States & Conversation Rules:	(ASKING CC-16) (CANCELLED CC-14) (COUNTERP CC-11 CC-12 CC-13 CC-15) (ACCEPTED CC-7 CC-8 CC-9 CC-17 CC-18) (WORKING CC-3 CC-4 CC-5 CC-6 CC-10) (PROPOSED CC-2)

Initiate Conversation Close Frame

Figure 6.6: Browsing a Conversation Class

with the user's personalized stub agent. Those requests have been encoded explicitly using the definitions in the pattern grammar, and will be automatically transferred to the correct user screen.

Input requests for a particular user may come from any conversation running in the agent application at any point in time. This includes conversations started by other users or even conversations which have been created internally when required. As a result, all these requests will appear immediately after being detected on the user's screen, successively or in parallel - a process accompanying the entire life cycle of the user-integrated multi-agent application.

For visualizing the incoming requests adequately, we implemented the navigation and modal dialog model proposed in section 5.2.6. The result can be found in figure 6.7.

In our case, after initiating the Customer Conversation, we will get the new created conversation object and a first request for specifying a customer order. The upper elements in figure 6.7 indicate that to the user. By clicking on the request, it will be decomposed first into a selection list. Remember that input requests may refer to decision making processes, where the user is required to choose from different ways to respond.

Here, we have no decision making, so that only one action is possible which is to fill a customer order (see frame in the middle of the figure). By clicking on that, the elements of the pattern grammar constructing the request are evaluated and transformed into modal dialogs. In the lower frame, we see the top level for such a dialog consisting of a name (here

Ongoing Conversations

Agent	Conversation	Instance Of	State
CUSTOMER	CUS:ANONYMOUS[97]:0	CUSTOMER-CONVERSATION	START

Requests from Agents

Agent	Conversation	State	Action Alternatives
CUSTOMER	CUS:ANONYMOUS[97]:0	START	1

Here we have a list of actions you may perform

Agent: CUSTOMER
Conversation: CUS:ANONYMOUS[97]:0
State: START

1. PROPOSE Welcome! Fill the order form, please...

Message Composition

Welcome! Fill the order form, please...

PROPOSE

SENDER ANONYMOUS[97]

CONVERSATION CUS:ANONYMOUS[97]:0

RECEIVER CUSTOMER

CONTENT Define Form #0.1. Press please!

Legend

Name The name of this form

Value Field with predefined value that cannot be changed

Define Button to compose a complex value

Key The key for a value

Value Field to enter/modify a value (may contain default)

Type Type descriptor for value expected in a field

Figure 6.7: Effects of Initiating a Conversation Class

a KQML-performative), the sender (which is the user itself), the receiver (which is the agent to send the message to), the conversation concerned and the actual content. All the values in orange boxes are predefined, the cannot be changed by the user. The actual customer order is a complex object itself, which can be accessed by clicking the button beside :content upon which another modal dialog will be displayed. How the user can fill a customer order there is shown in figure 8.3 in the chapter describing the supply chain demonstrator.

After that, the user can *browse* the completed request (see figure 6.8) before *submitting* it through GenUA back to the agent environment. If the message has been received by the agent and the conversation concerned, the input request will be removed automatically as GenUA informs the applet about that. Now, the input values provided can be evaluated and the execution process of the conversation continues.

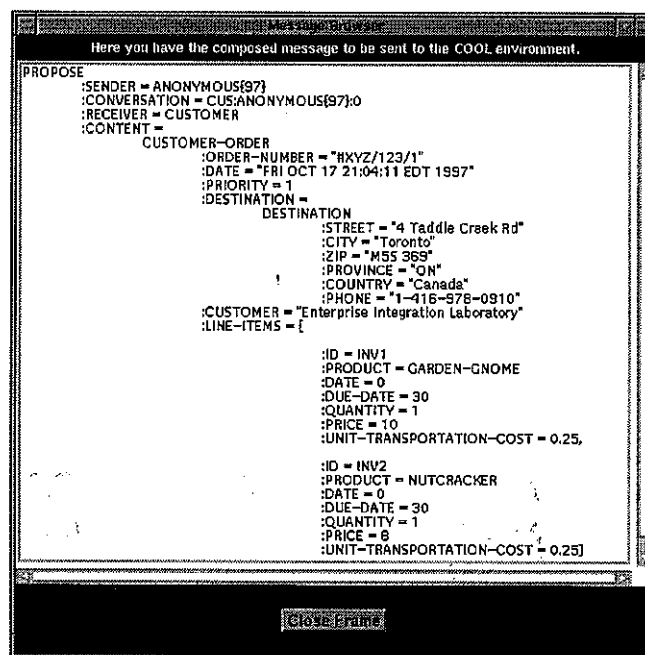


Figure 6.8: Browsing a Completed Input Request

We do not support deleting requests neither in GenUA nor from the GUI. The reason is, that if a user deletes a request, he will never have the chance to provide adequate input, as it is needed right now from a particular conversation. As a consequence, the conversation will simply remain in that state forever. With that, perhaps the entire scenario and all other users are affected unless you access the running application directly with system calls. Thus, the politics for GenUA must be to deliver input requests to users immediately, to expect a response from them sooner or later and to take care about cleaning up obsolete requests.

Inspecting Messages from the Application

Analogous to input requests, notifications or results for a particular user may come from any conversation during the applications execution process at any point in time. To ensure a minimum of transparency, the actual content is wrapped into a message layer that helps the user to associate it to particular agents and conversations. Message performatives and comments provide a context for the result or notification. As the actual content may be any kind of data type (text, charts, gifs, audio, etc), a type attribute is attached so that the graphical interface is enabled to invoke a corresponding handler which displays the content adequately onto the user's screen.

At present, we support two content types which are text including printable values of arbitrary complexity and composition, and a special format for a Gantt Chart specification used in our demonstration scenario. However, we kept the representation facility as open as possible so that new result types can be introduced readily.

As an example, we move on to the user which has the executor role in our little "Supply-Chain+Scheduling" application and controls the Logistics and all the Plant and Transport agents. We take a closer look at what happens on this user's screen, when an order from a client has been received. After Logistics has done some actions like decomposing the order into activities, scheduling them and identifying facilities to execute them, the results are made available to the user in form of corresponding messages. Figure 6.9 illustrates this situation.

In the GUI's response panel, incoming messages will be browsed due to the moment of reception from bottom to the top. The user can select any of those at any point in time, inspect them and may even use informations or conclusions gained for upcoming input requests. As opposed to input requests, messages can be deleted separately or completely from that panel at will.

Just for illustration, we show here the initial message for the user when the Logistics agent has received a new customer order, and the textual result for the scheduled order. As this result follows a specific description for a Gantt Chart (see Content-Type) we are able to transform the message in a graphical representation as well. Some examples for Gantt Charts can be found in chapter 8. The content displayed is self-explanatory. What you can see here is, that we are able to format complex data structures combined with text adequately.

Obtaining Historical Information

During the session with GenUA, the user can retrieve historical information at any point in time, independently from being linked to a particular application. Historical information stored at GenUA encompasses any essential interaction event between the user and GenUA resp. the user and a multi-agent application:

Messages from Agents				
No. + Agent	Message Type	In-Reply-To	Content-Type	Comment
4. LOGISTICS	ANNOUNCE	FORM-LARGE-TEAM	TEXT	"Large team will be formed"
3. LOGISTICS	TELL		TEXT	"Contractors ranked"
2. LOGISTICS	TELL		CHART	"New order successfully scheduled"
1. LOGISTICS	ANNOUNCE		TEXT	"Order received"

Agent Notification Browser		Agent Notification Browser	
Performative:	ANNOUNCE	Performative:	TELL
Sender:	LOGISTICS	Sender:	LOGISTICS
Conversation:	CUS-ANONYMOUS(97)-0	Conversation:	CUS-ANONYMOUS(97)-0
Reply-With:		Reply-With:	
Date:	Sat Oct 16 00:01:07 EDT 1997	Date:	Sat Oct 16 00:01:10 EDT 1997
Comment:	"Order received"	Comment:	"New order successfully scheduled"

<p>"An order from customer"</p> <p>ANONYMOUS(97)</p> <p>"with the following content"</p> <p>CUSTOMER-ORDER</p> <p>:ORDER-NUMBER = "#XYZ/123/1"</p> <p>:DATE = "FRI OCT 17 23:59:38 EDT 1997"</p> <p>:PRIORITY = 1</p> <p>:DESTINATION =</p> <p> STREET = "4 Taddle Creek Rd"</p> <p> CITY = "Toronto"</p> <p> ZIP = "M5S 3G9"</p> <p> PROVINCE = "ON"</p> <p> COUNTRY = "Canada"</p> <p> PHONE = "1-416-978-0910"</p> <p>:CUSTOMER = "Enterprise Integration Laboratory"</p> <p>:LINE-ITEMS = [</p> <p> :ID = INV1</p> <p> :PRODUCT = GARDEN-GNOME</p> <p> :DATE = 0</p> <p> :DUE-DATE = 30</p> <p> :QUANTITY = 1</p> <p> :PRICE = 10</p> <p> :UNIT-TRANSPORTATION-COST = 0.25,</p> <p> :ID = INV2</p> <p> :PRODUCT = NUTCRACKER</p> <p> :DATE = 0</p> <p> :DUE-DATE = 30</p> <p> :QUANTITY = 1</p> <p> :PRICE = 8</p> <p> :UNIT-TRANSPORTATION-COST = 0.25]</p> <p>"... has been received from CUSTOMER. LOGISTICS is trying to determine a scheduling solution for it..."</p>	<p>PROBLEM</p> <p>:Agent = LOGISTICS</p> <p>:Name = PROBL-36</p> <p>:Jobs =</p> <p> JOB</p> <p> :Name = JOB-37</p> <p> :Release-Date = 0</p> <p> :Due-Date = 30</p> <p> :Operations = ASM-38 PAINT-39 TRANSP-40</p> <p> :Attributes =</p> <p> :ID = INV1</p> <p> :PRODUCT = GARDEN-GNOME</p> <p> :DATE = 0</p> <p> :DUE-DATE = 30</p> <p> :QUANTITY = 1</p> <p> :PRICE = 10</p> <p> :UNIT-TRANSPORTATION-COST = 0.25</p> <p> JOB</p> <p> :Name = JOB-41</p> <p> :Release-Date = 0</p> <p> :Due-Date = 30</p> <p> :Operations = ASM-42 PAINT-43 TRANSP-44</p> <p> :Attributes =</p> <p> :ID = INV2</p> <p> :PRODUCT = NUTCRACKER</p> <p> :DATE = 0</p> <p> :DUE-DATE = 30</p> <p> :QUANTITY = 1</p> <p> :PRICE = 8</p> <p> :UNIT-TRANSPORTATION-COST = 0.25</p> <p> :Resources =</p> <p>RESOURCE</p> <p> :Name = ASSEMBLY-PLANT</p>
---	--

Close and Keep Message	Close and Delete Message	Close and Keep Message	Close and Delete Message
------------------------	--------------------------	------------------------	--------------------------

Figure 6.9: Inspecting Received Messages

- begin and end of sessions with GenUA
- creation and shutdown of applications
- linking and unlinking of applications
- navigating through the application
- initiating conversations
- input requests from the application
- completed input sent back to the application
- user conversation updates
- results and notifications for the user

The history for each user is subject to an exorbitant growing through the lifespan of GenUA. At the same time, in a business context there is the requirement that no information should get lost. On the one hand, we have to limit the amount of information transferred between GenUA and the graphical interface and on the other hand we need to provide as much information as possible.

In order to deal with the information flood, GenUA records all historical information for each user and creates dumps on a persistent medium from time to time. Those dumps will be no longer accessible through graphical interfaces but can be inspected if desired. The history is provided only on explicit demand, and we stipulate that during a session, the interface maintains its own list of already retrieved events. Thus, we can avoid to transfer the same information twice.

Anyway, history information is accessible by pressing the corresponding button from the main window. It may take some time to transfer all the information, but in the end, a History Frame will pop up. The regular pattern of interactions between user and GenUA resp. multi-agent application facilitate a clear and handy graphical representation. Figure 6.10 illustrates an example for a History Frame, after several sessions with different applications.

The top panel shows interactions related to the creation and shutdown of multi-agent applications. We assume, that this is not a daily business. However, the person responsible can see here, when a particular application had been started last or how long it had not been used.

The panel in the middle presents user sessions with particular multi-agent applications. Whenever a user decides to leave an application while still participating, one session is completed. (Otherwise, historical information will be dumped only without being accessible through the GUI). We see that as completed lines in the list with a "linking" and "unlinking" time. Multiple sessions with the same application appear numbered.

History

Session

Begin: Tue Nov 04 15:54:43 EST 1997
End:

Application Administration

Time	Application	Notification
Thu Oct 30 16:48:45 EST 1997	SUPPLY-CHAIN+SCHEDULING	CreateApplication
Thu Oct 30 16:48:45 EST 1997	SUPPLY-CHAIN+SCHEDULING	SUCCESSFULLY_CREATED
Fri Oct 31 10:23:25 EST 1997	OFFICE-SYSTEM	CreateApplication
Fri Oct 31 10:23:25 EST 1997	OFFICE-SYSTEM	SUCCESSFULLY_CREATED

Linked Applications

Application	Linked At	Unlinked At
SUPPLY-CHAIN+SCHEDULING #0	Thu Oct 30 16:48:45 EST 1997	Thu Oct 30 16:55:00 EST 1997
OFFICE-SYSTEM #0	Fri Oct 31 10:23:36 EST 1997	Fri Oct 31 11:38:15 EST 1997
SUPPLY-CHAIN+SCHEDULING #1	Fri Oct 31 15:21:03 EST 1997	

Interactions

Application: **SUPPLY-CHAIN+SCHEDULING #0** Interactions Total: **26**

Type: **Response** From: **SPIN-Server**

Time: Thu Oct 30 16:51:59 EST 1997
Concerns: PLANT2
Tag: Re@ActionRequests
Content:

Agent: PLANT2
Conversation: FLT45
State: PROPOSED

Alternative #1

ACCEPT "Join the large team"

SENDER
ANONYMOUS{97}

Current Item: **26** **1** **2** **3** **4** **5** **6** **7** **8** **9** **10** **11** **12** **13** **14** **15** **16** **17** **18** **19** **20** **21** **22** **23** **24** **25** **26**

CloseFrame

Unsigned Java Applet Window

Figure 6.10: Browsing Historical Information

If a user wants to trace back interactions of a particular session, he can select the session from the list upon which the lower panel will be activated. Here, every interaction can be inspected separately. We see the application session, the total number of interactions, and for each interaction the type, the initiator, the current number and the content of the event.

The content of the events does not only reflect the interactions detected. Rather, we nearly copied the content of exchanged messages between the graphical user interface and GenUA into the event object. This allows for easy displaying, and by using the same output structure as during the ongoing interaction, supplies transparency for the user. By means of the control buttons, the user can navigate through the interaction events of a application session at will.

In our example, we have a user who is allowed to create and participate in several applications. He had a session with the "Supply-Chain+Scheduling" application two days before, a session with the "... application yesterday, and today he has resumed his session with the former one. Supposed, he has got a request from the application, and he remembers to have had the same situation two days before. If he now wants to see, how he responded then in order to apply that to the current situation, he just needs to select the event that depicts the completed request and use the information to fill the current one.

Leaving an Application

As mentioned several times, GenUA provides mechanisms for suspending participation in applications temporarily, for resuming them later on and for getting the user informed about crucial changes during his absence via multimedial telecommunication messages. But an application may also be used only once for a particular service, and then never again. Also we don't know, whether a user will be offline only over night or if he is going on a business trip. Considering that, there is a little decision model, whenever a user wants to leave a session with an application

First, the user will be asked first whether he wants to suspend or to terminate his participation. The latter one leads to a removal of all his "traces" meaning all his application tools will be terminated and the historical information concerning this application will be outsourced.

Second, we need to know, whether the user wants to get informed through multimedial telecommunication messages and, if so, how. If the user waives that service, he will get aware of changes during his absence only if he resumes his session again.

Third, we let the user specify, on which medium and where exactly he wants to receive messages. In the small dialog provided, he may select from "Email", "Fax" or "Voice", and specify an email address, a fax number or a phone number. Currently, only the email service is supported directly. This decision will influence ALL the applications the user is still participating in, not only the one he is about to leave. There is no need to assign that to every application separately, before a user closes the graphical interface.

After this process is finished, the top level window remains open, so that the user can link another application right away.

We do not allow to close the graphical interface unless the user has correctly left an application temporarily or finally. Otherwise, we would leave GenUA in an inconsistent state meaning it will keep trying to send requests and notifications from the application to the address of a graphical interface that no longer exists.

Shutdown an Application

Running applications can be shutdown only by those users who are allowed to create them. As only those applications are made available to the GUI at all, where a user has administration and/or access rights, there is no way to affect applications unauthorized from a GUI. The problem of a clean shutdown is that the administrator is, in generally, completely in the dark which users are currently participating in a scenario, where they reside and whether they are online or offline. On the other hand, it should be possible to terminate an application without forcing the administrator to run after each participant.

Thus, the administrator may choose from two modes to shutdown an application. The so-called "user-friendly" mode makes GenUA to check out all users currently interacting with or just being in the scenario and executes the administrators intention only, if no one is participating anymore. Otherwise, he will get at least a hint by providing the user's login name and there current location. However, in the so-called "user-ignorant" mode, the application is terminated right away, but every user will receive at least a notification from GenUA, and the history of interactions remains untouched.

Unlike leaving an application, we allow to close the graphical interface after creating an application right away and to shutdown an application later during a new GenUA session.

6.3 Summary

With the Generic User Agent, people in a collaborative environment can be linked directly from there working place to COOL agent systems. They can immediately participate in the execution process of complex distributed systems without any knowledge about its internal procedures and structures. All user interactions are fully controlled and coordinated by GenUA.

In featuring an interaction mode where users are explicitly and expressively requested to provide input at defined moments, he will be freed from any wondering what to do and when. Automatical relay of execution results and full navigation and inspection support provide a maximum in system transparency.

Any kind of user input expected by an application is incorporated in predefined structures that just need to be filled by the user, when the request has been detected by GenUA. Input

requests can be easily visualized as hierarchically nested modal dialogs. Results come up as messages with a clear context and arbitrary content that just needs to be visualized by passing to a corresponding handler.

GenUA supplies mechanisms for multiple ongoing dialogs, which can be suspended and resumed at any point in time. Users can access historical information and receive multimedial messages about crucial changes in case of non-presence.

Administration and authorization is supplied on both user and application level. Users can only access entities they are allowed to and which one are of immediate need for their interaction.

The architecture of the Generic User Agent allows for a flexible composition of functional elements and a dynamical extension with new capabilities. Interactions among components are mediated by a facilitator on demand. GenUA processes declarative knowledge at a highly abstract and domain-independent level. By means of a driver concept, different graphical interfaces can talk to the Generic User Agent at the same time.

The sample Java applet exposes a convenient way for users to interact to multi-agent applications from anywhere in the world through a Web browser. The user is guided through the entire interaction process while giving him the freedom to select his next action. The all-in-one concept provides transparency and control at any point in time.

The Generic User Agent links people and agents, wherever they may reside, towards a single virtual platform and assists them to solve problems in a collaborative manner.

Chapter 7

Building Agent-Integrated Business Structures

Now that we have a tool for developing systems of cooperating agents and a interface for online user interaction, we have the necessary means to think about a re-design of distributed information systems in the business world. As mentioned in chapter 3, the goal is to come to a virtual platform where information and decision processes are coordinated in a mixed agent-user-initiative across organizational boundaries.

A major advantage of the agent approach is the natural way to conceptualize a system. As most of the entities within a business context at a physical level (machines, transportation systems, inventories, etc) and at an organizational level (plants, departments, offices or even individual people) can be readily conceived in terms of agents, there is only a small step from a traditional description to a multi-agent notion. Relations among the entities, expressed in terms of interactions, information exchange, coordination or negotiation processes can be directly transformed into an agent model.

Building multi-agent applications that involve humans in their execution requires a special software engineering process. For collaborative systems in industrial or business domains, we propose the following steps to be taken:

1. Identification of the agents participating 7.1
2. Determination of their tasks 7.2
3. Analysis of the information and control flow among the agents 7.3
4. Definition of the user's role and interface character 7.4
5. Specification of the user's interactions 7.5

In the following section these steps will be examined more detailed. For better illustration, we will apply them to the domain of supply chain management.

7.1 Step 1: Agent Identification

Which entities in a business application context will be identified as an agent, depends on the granularity of the modeling process and on the complexity of its behavior within the system. As emphasized in the section on agent theory 2.1.2, we use the term “agent” to describe entities in an intentional stance, which means for elements that expose a somewhat autonomous and goal-oriented behavior. In an enterprise context, such a stance can be readily applied to any kind of organizational structure from viewing the company as an “agent” in the market or understanding the production and distribution department as “agents” in a company to defining the role of, say, an individual assembly line worker as an “agent” in the production process or a clerk working on an insurance case. The agent attribute also works perfectly for technological components accompanying the product and service flow: assembly lines, inventories and transport vehicles in manufacturing, switchboards and exchange points in telecommunication and so on.

All of them share the properties to hold some beliefs from a local perspective, to act as a result of an internal or external planning process, to respond to changes in their environment and to coordinate their actions with other entities.

The objective of an agent-based business information system is to provide integrative and coordinative support for those entities by reflecting their course of action and interactions with each other, to facilitate information exchange and to take on automatizable routine activities. Depending on the application purpose and the scope, we will always find a more or less intervoven network of interacting agents.

An important issue in the modeling process is the integration of the plethora of legacy information systems and tools. It should be emphasized that the “agentification” of a business information system does not require the reimplementation of the numerous applications already employed within a company’s network. Software agents can encapsulate databases, decision support system and online services, thus linking them towards a virtual platform with clearly defined interfaces.

In adopting the agent view, we organize the supply chain as a network of heterogenous, cooperating agents, each performing one or more supply chain functions, and each coordinating their actions with other agents. As we will see, a supply chain process is simply a serie of structured interactions among the agents. Given the distributed decision making process of supply chain management, we construe a distributed decision support system with agent technology. An example for a agent-based multi-level supply chain is illustrated in figure 7.1.

We have defined several abstraction layers from the supply chain to the plant level, and

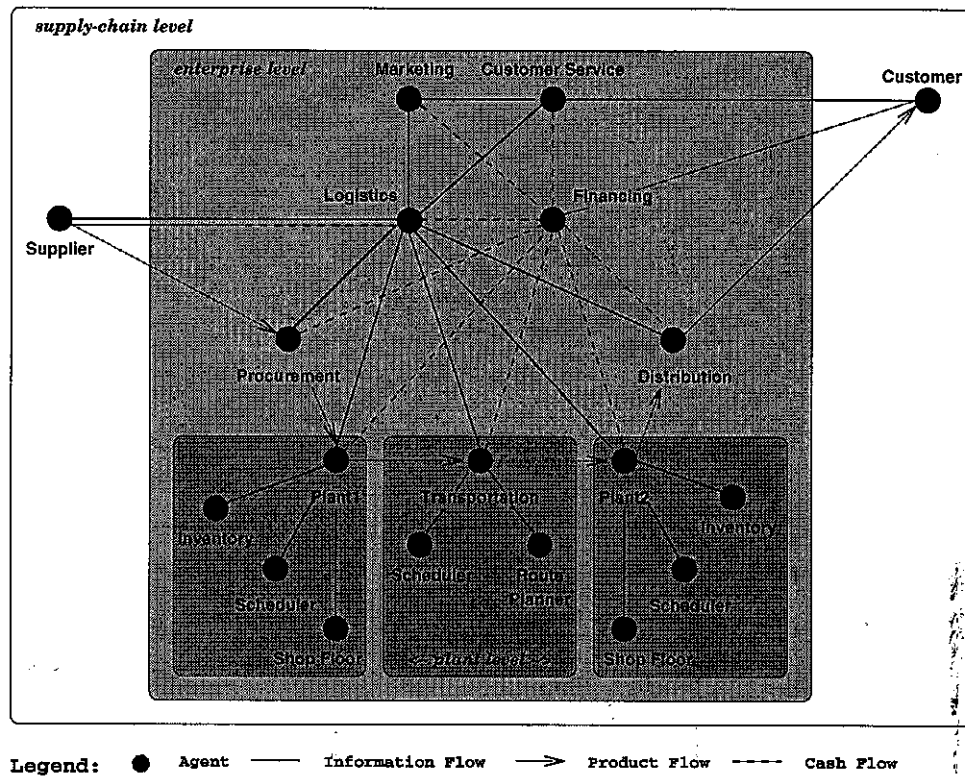


Figure 7.1: Multi-level Supply Chain

for each layer we have identified some agents with essential contributions to supply chain management domain. The interdependencies among the agents in terms of product, information and cash flow arise naturally during the modeling process and will later result in concrete information exchange and coordination protocols.

Most of the agents may represent or wrap corresponding legacy software systems. The Inventory agents may be connected to a bar code scanner in the material stock keeping track about the inventory level. The Scheduling agents may be fed with output of optimal batch size tools, machine scheduling programs and so on. The Shop floor agents may supervise CNC programs for the assembly line. The Finance agent may monitor online stock information such as shareholder values, commodity prices and interests.

7.2 Step 2: Agent Tasks

Once we have identified the agents, we can begin to associate particular tasks they should fulfil in the global execution context. Enterprise settings are characterized by competencies and responsibilities attached to any kind of organizational entity from which actual functional tasks are derived. As a result, we can often find a comprehensive hierarchy of tasks where abstract descriptions are decomposed successively down to individual operational steps.

When defining the agents' tasks in a system, we need to examine the individual operational steps and decide which one (a) are automatizable and (b) will lead to a significant improvement of performance when automatized. Ideally, the introduction of multi-agent systems should be accompanied by a reengineering process, a fundamental reconsideration of the necessity and value of business processes and the efficiency of the organization. It is essential not to fix inefficient structures by tailoring technological systems to it. Only from a global perspective on organizational performance, we are enabled to delegate tasks adequately to a multi-agent system.

Other than that, the association of tasks arises naturally from the scope of actions an agent should have. In the same way, as an organizational or technological entity is responsible to perform certain tasks from a local perspective while interacting with other entities, the agent representing this entity will execute a subset of tasks while communicating with other agents. Competencies and authorities of agents should be defined as strict and clearly as stipulated in the real world.

For the supply chain domain, we can exemplarily identify the following tasks for agents:

Marketing :

- market information acquisition
- demand forecasting

Customer Service :

- order acceptance
- order servicing
- billing and accounting
- delivered goods servicing

Logistics :

- order processing
- identification of necessary facilities
- global material planning
- supplier coordination
- global production planning
- distribution coordination
- global transportation management
- determination and propagation of target values to the respective agents
- handling of unforeseeable events and exceptions

Inventory :

- ensuring safety stock level
- calculating reorder points
- determining the order quantity

Scheduler :

- defining the production sequence
- calculating the batch size
- determining the production schedule

All of these tasks may have planning, decision making or execution elements and most of them are already supported by algorithms, models or tools. The challenge is to capture and integrate these elements in adequate handling routines for agents and to determine interdependencies between tasks in terms of necessary information exchange and coordination requirements.

7.3 Step 3: Information and Control Flow

A business process is composed of a sequence of activities involving several organizations and resources. It links the individual tasks by information and control flow in order to achieve a particular goal. When we have defined a task model for the organization and assigned the individual tasks to agents, we now have to care about the relations among the tasks. Naturally, tasks are characterized by some kind of input coming from one or more previous tasks or the environment and producing some kind of output for one or more successive tasks or for the environment. Moreover, a task associated to an organizational entity may encompass the execution of a plan or protocol which in turn requires interactions with tasks associated to other organizational entities in a particular sequence of steps. The agent view allows to capture information acquisition, information exchange and interaction protocols through

- *Environment interfaces* to acquire information dynamically from all kinds of external sources and to provide informations to all kinds of external destinations
- *Local knowledge* for passing information from one task to another within the represented entity
- *Message passing* for exchanging information between represented entities
- *Coordination protocols* to reflect structured interactions between represented entities

The latter one includes all kinds of cooperative activities such as negotiations, (distributed) decision making processes, or joint task execution beyond the scope of an individual entity.

The described mechanisms go far beyond Electronic Data Interchange between traditional information systems. Agents do not only simply respond, they can autonomously take corrective actions and make routine decisions. As agents can access a variety of sources and give conventional information systems a uniform interface, information becomes abundant throughout the organization. For coordination aspects, people can easily communicate through the agent system and if so desired agent protocols can guide and assist them in cooperating processes.

In the supply chain model, we exemplarily outline the following scenarios:

If the Inventory agent is connected to a bar code scanner for incoming and outgoing material, it can keep track about the maintenance of the safety stock level. The agent knows when material should be reordered and what the optimal order quantity is. At those reorder moments, or when the safety stock decreases below a certain level due to increasing demand from the production facility, the agent automatically sends material orders to the Procurement agent which in turn consults the supplier. If a supplier has problems to satisfy the

order, Procurement notifies the Logistics agent upon which Logistics may start a negotiation process with the supplier and the Plant agent in order to overcome the situation. Logistics may temporarily request another supplier to satisfy the material demand. If even this fails, the Plant agent will make the Shop Floor Agent to turn down the production and the Scheduling Agent to revise the production plan. Logistics will inform the Customer service about postponement of deliveries.

From another perspective, the Shop Floor Agent in plant A monitors the ongoing production process. If breakdowns occur, they may require initially a notification to plant A's Scheduling agent for slight modifications of the production plan. In case the breakdown requires a long time to fix and the Scheduling agent cannot find a solution that matches the global production constraints, it will inform the Plant agent which in turn informs Logistics. Logistics may now try to find a replacement capacities at other plants by bidding elements of plant A's production program to other Plant agents. Depending on their local schedule and inventory, those Plant agents may accept or reject the request. If another plant is "willing" to provide capacity, Logistics will need to ask Transport agent whether the new location can be serviced. Failure of one of the procedures above will lead to a notification to the Customer service agent about delayed or even cancelled orders and to a notification to the Finance agent as cash flow and revenue will be concerned significantly.

Those two scenarios allow only a glimpse to what is possible agent-based information and control flow.

7.4 Step 4: User Role and Interface Character

A multi-agent application in a business context is meant to support a variety of different end users in their daily work: to free them from routine activities and to facilitate communication and coordination across the company's organization. This requires that users are allowed to interact with the agent system directly from their working place through adequate interfaces.

The complexity and functionality of user interfaces to the agent system depends on the role, a specific user or a intended group of end user's are meant to play when dealing with the system. The specific role of each end user in the system is determined by two factors:

- the competency and authority
- the degree of participation

The former one is mostly predefined by the role of the user in the company's organization, stipulated in workplace descriptions, competency models and authority rights. Each user will have its own local perspective to the elements and activities of the agent system. Determining the user's role encompasses the association of end users working in any of the involved organizational entities to specific agents and agent functionalities of the system. The

interface needs to be personalized towards an individual user or group of users so that it can route information flows from and to the agent system, and can ensure authority and access rights.

The latter one describes to what degree a user is supposed to be involved in the system execution process. To some extent, this is stipulated already by the user's competency and authority. The degree of participation will range on the following scale

Monitoring The user obtains specific output during the system's execution.

Information Acquisition The user may send queries to the agent system in order to retrieve specific information.

System Input The user may be asked to fill spreadsheets, to edit documents or to answer questions.

Communication and Information Exchange The user may communicate through the agent system with other users in order to exchange information or to coordinate activities.

Decision Making Either as a result of information gained or by evaluating proposals from the agent system, the user may need to make decisions in a particular situation.

Task Delegation The user may order the agent system explicitly to execute a particular task.

All of these elements can be combined for the specification of a user's role which must be supported by a corresponding design of the user's interface.

Depending on the number of users occupying a specific role in a multi-agent scenario and on the characteristics of such a role, interfaces may range from simple monitors or control terminals to individual interface agents tailored to the needs of a specific user. However, all approaches have one thing in common: The interface should incorporate the user explicitly in the agent system by establishing an own identity. Then, all agents will have an interlocutor whenever user interaction in terms of input to the system, decision making or result presentation is required.

Back to the agent-based Supply Chain, we exemplarily describe two possible end users with different roles in the execution process: the inventory assistant and the logistics manager.

The inventory assistant may be responsible to supervise incoming and outgoing material and products and to ensure the defined stock levels at any point in time. With respect to participation, he will occupy a more passive role in the multi-agent system. Thus, we may link the inventory assistant by providing a simple control monitor to the Inventory agent as an interface. The control monitor may be tailored to the different inventories maintained in a company (purchasing inventory, plant inventory, distribution inventory). On the other hand,

it does not need to be personalized for every inventory assistant individually. We may simply have one or more terminals in the stock, where every user who is known to the system as a inventory assistant may use the same functionality provided by the interface.

The logistics manager will occupy a central and comprehensive role in the agent system as he does in the supply chain organization. His responsibility is to supervise the overall execution of the internal supply chain, to ensure its efficient operation and to intrude when unforeseeable events occur. A logistics manager needs easy access to all kinds of information, models and tools at any point in time, automated information acquisition displayed in a clear format, comprehensive communication facilities, active decision support and to trigger comprehensive evaluation and execution routines affecting the entire agent-based supply chain. Moreover, the logistics manager's way of action is highly determined by an own conceptual models about the operation of a supply chain, personal preferences for useful representation and, traditionally, a scepticism about putting a computer onto his desk. What we need here is an interface that is tailored down to the last detail towards the requirements and needs of the individual logistics manager hiding any system-internal details and coming up with powerful graphical components and the most intuitive interactions.

7.5 Step 5: Specification of the User's Interactions

Once the role of a user is defined, and the interface is endowed with corresponding functionalities, we need to specify all user interactions in detail. This includes to examine all agent activities including local actions, plan execution and coordination protocols step by step and, in close cooperation to the end user, to define what kind of interaction should occur when and how it should be represented. Closely related is the question of activation, which means whether the user or the agent is the trigger for an interaction. This is a procedure where the system's execution process is tailored exactly to the needs and requirements of those end users, the multi-agent system is made for after all.

In generally, while executing a particular activity or protocol, agents may expect user actions (input, decision making, etc.) from any user interface and send messages (notifications, results, etc.) to any user interface, wherever the agent or the user interface physically resides. Vice versa, a user should be enabled to access and utilize the services of an agent at any point in time. The challenge here is to determine the need and the exact moment for user interaction so that on the one hand the system provides a maximum in service and transparency and on the other hand unnecessary overload is eliminated.

Both user actions and messages to users may assume any possible format. User input can be captured in command lines, selection lists, tables, spreadsheets, documents and so on. Messages to users may encompass the entire multi-medial world: text documents, tables, charts, faxes, e-mails, audio and video files. The challenge here is to acquire the interaction knowledge from the users and to incorporate it into the agent activities.

Due to the complexity and constant changes of user-agent interaction, this will require a prototypical and evolutionary approach comprised of continuous deployment, feedback and revising.

In our supply chain model, we continue to focus on involving the inventory assistant and the logistics manager. Suppose we have an Inventory agent being fed with inventory management procedures and reorder mechanisms and being linked to bar code scanners. For the shop floor assistant, the Inventory agent may automatically update lists of incoming and outgoing items, material re-orders, etc. On request, the agent may generate tables and diagrams about that on a daily, weekly and monthly basis. Through a more or less comfortable search mask, the shop floor assistant may send simple queries to the Inventory agent's database for instance about the current amount of a particular entity or about its physical location in the stock.

For our logistics manager, we may envisage an interface to the Logistics agent somehow similar to a executive support system. A monitor may constantly show the periodical development of crucial supply chain parameters such as inventory levels, lead times, transportation and production capacities and so on. A graphical supply chain model may visualize the complete supply chain including facilities, material and information flow. By simply clicking on the elements, the manager may inspect the current parameters. If unforeseeable events occur, the concerned element may blink red, while the manager will be not only informed about the type of event, rather the agent tries to generate proposals for actions to be taken or decisions to be made, for example based on case based reasoning. Through a comprehensive communication module including video conferencing, fax and e-mail, the manager can coordinate corrective actions online with the executives of the facilities concerned.

7.6 Summary

By introducing agent-based software systems into the domain of supply chain management we are able to integrate concepts and tools and to assist *coordination* and *communication* among the individual entities in a flexible manner which are the crucial factors in organizing efficiently manageable structures.

In capturing a supply chain as a set of autonomous, distributed agents with clearly defined interfaces, we can also address the high dynamics a supply chain is subject to. A multi-agent system can be adapted to changes in the environment both (1) locally, by extending or modifying the functionality of one agent without affecting the rest of the community and (2) globally, by adding or removing agents to the system. As a result, we have a robust and flexible structure that can undergo continuous adjustments and improvements without loss of performance.

Chapter 8

The Supply Chain Demonstrator

8.1 Motivation

In order to demonstrate the performance and the high support level of the Generic User Agent system, we've extended an existing COOL prototype for an agent-integrated supply chain which now involves the user actively along the entire execution and decision making processes. The design and implementation procedure concerning the user aspects followed the steps proposed in section 7. The basic supply chain model is deemed to be appropriate to give a notion on both the universality and variety of the COOrdination Language and the essential aspects supported by the GenUA interface. However, it was neither the objective to stipulate a specific supply chain approach nor to dictate when and how end users have to be involved in a real-world scenario. Thus, the following description should be seen as an incentive and encouragement for the interested reader to think about casting models and methodologies on coordinated supply chain management in a multi-agent scenario, and about integrating the everyday activities of executives and employees along the supply chain. COOL provides the mechanisms for reflecting decision and information processes among organizational units adequately, GenUA allows structured and guided interactions of end users as the determining element to control and to influence these processes from any workstation or PC. We envisage a collaborative environment in which users and agents are linked to a single virtual platform across organizational boundaries.

8.2 The Company

For our supply chain model, we've imagined a medium-sized enterprise doing business in arts and crafts in Saxonia. Its famous articles including nutcracker, garden gnomes and wooden Christmas pyramids are sold in many countries, and are said to reflect the native German culture. The "Fichtelberg GmbH" company consists of a headquarter and three

workshops distributed in the local area. The headquarter accommodates central customer service, accounting, logistics and purchasing. The logistics office is responsible to task and to supervise the workshops, and to coordinate the transportation facilities. Purchasing operates the main inventory and contracts suppliers. Top articles can be produced in any of the workshops according to the demand. However, one of the workshops is specialized in assembly, one of them in painting and one of them has facilities for both. The company is the main client of two local forwarding agencies which transport material from the wholesaler and among the facilities, ship final products to the next station and airport terminals, and deliver a number of tourist shops. For marketing purposes, the "Fichtelberg GmbH" consults from time to time a local advertising office.

Through the past decade, the demand for high-quality German arts and crafts has continuously increased, and the company was forced to change its production to semi-automatical manufacturing. They implemented an electronic material and product registration system in every workshop which is directly connected via Electronic Data Interchange with the central logistics and purchasing office. Based on the forecast of the customer service, each workshop sets up weekly and monthly production plans and coordinates them with the other during a meeting in the logistics office. The company has also installed a scheduling tool in the logistics department and in each workshop and forwarding agency. This standard application features a general algorithm for constraint-based scheduling using texture measurements of the problem structure. This means, once being configured for a particular environment with an adequate model, the tool decomposes and schedules activities to be done there.

However, if problems or breakdowns occurred in one of the workshops, it was a troublesome way for the logistics manager to get informed immediately about that in order to discuss replacement capacities with the executives of the other workshops. The company's small intranet allows e-mailing but no video conferencing. Also, the customer service cannot always satisfy inquiries about the state of an order, and does not have adequate information in order to make alternative proposals to the clients if the capacities drop.

Now, the modern boss has read many about the use of agent technology, even in small business. He is looking for a system every employee can work with without being a computer expert, that frees the executives from time-consuming routine activities in coordinating their decisions. It should support the successful concept of team structures, supply workflow elements and integrate the company's software tools, particularly the scheduling applications.

The solution provided by the Enterprise Integration Laboratory was most convincing and turned out an invaluable support. The COOL multi-agent application allows teamwork, planning and execution coordination in a virtual agent-based supply chain at several levels of detail across multiple facilities. Both sophisticated local reasoning tools, like the scheduling application, and non-trivial multi-level and dynamic replanning protocols with response to internal or external changes and events are incorporated. By means of the Generic User Agent, end-users got a maximum in interaction support, transparency and control. The Web interface supplied with GenUA was familiar to most of the end users involved and lead to

a wide acceptance among the employees. The communication expenses could be decreased radically. Breakdowns can now be much better controlled and overcome, resulting in a higher customer satisfaction and increasing competitiveness.

8.3 The Agents

We modeled the supply chain of the “Fichtelberg GmbH” at the initial stage as a set of six COOL agents: a Customer agent, a Logistics agent, three Plant agents and two Transport agents. This approach arised very naturally from the organizational units participating in the production and delivery of goods in the company.

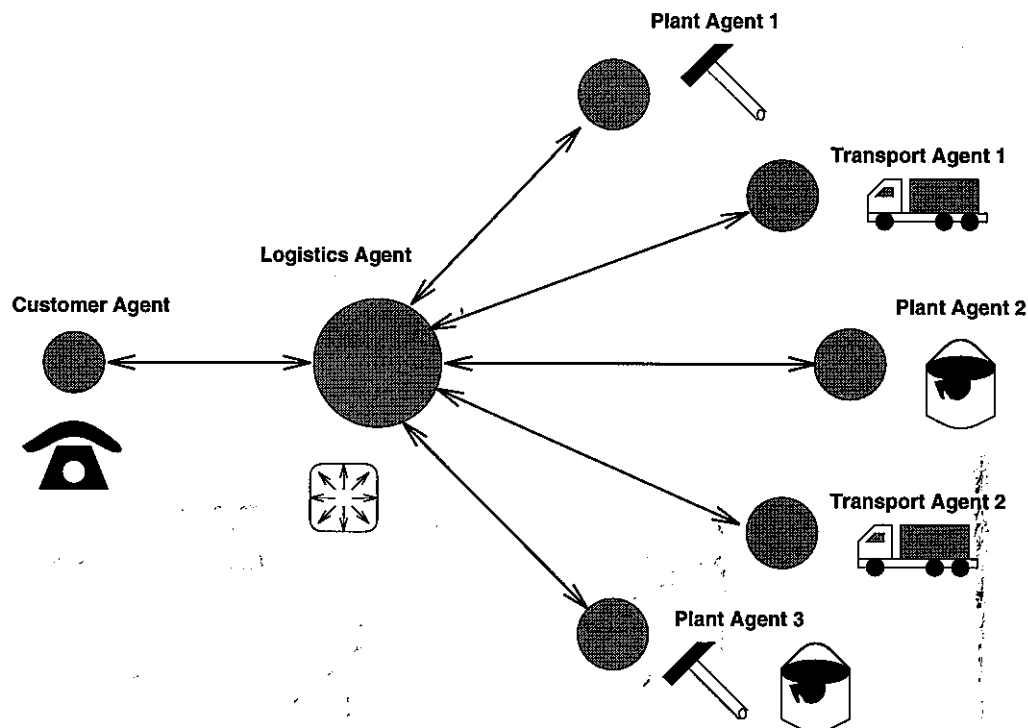


Figure 8.1: Agent-based Supply Chain Model of the Demonstration Company

Figure 8.1 presents the agent model of the supply chain. The agents are physically distributed across the company’s intranet according to the organizational units they are associated to. Arrows indicate coordination and communication relations among the agents. In the following paragraphs, we will omit the attachment “agent” occasionally, as it is obvious what the agent’s names are referring to resp. which organizational unit they represent.

8.3.1 The Customer Agent

The general purpose (or goal) of the *Customer* agent is to achieve optimal client service. The agent is responsible to take and to service customer orders, and is consecutively aimed at supporting the employees in the service center of the "Fichtelberg GmbH".

Customer orders in our approach are meant to be acquired by these employees who enter the data into a generic spreadsheet. One can imagine that the orders arrive the service center in form of written forms, phone calls, faxes or e-mails. Of course, all these orders could be also scanned automatically by the Customer agent, if so desired. The information is then forwarded to the *Logistics agent* 8.3.2 which is responsible to make sure both manufacturing and delivery of the single items ordered.

During that time, Logistics sends constantly information about the processing state of individual orders, so that the service staff can answer client questions, for example about the delivery date. If Logistics detects problems with an order which affect crucial parameters such as delivery time or price, Customer obtains not only notifications but alternative proposals. They can be discussed immediately between the service employees and the clients (in using an adequate medium) resulting in modifications or cancellations of orders.

The procedure described above has been captured in a single interactive conversation plan: the *Customer Conversation*, which is described in paragraph 8.5.1.

8.3.2 The Logistics Agent

In reflection of the central role which one the logistics department occupies in managing the supply chain of the "Fichtelberg GmbH", the *Logistics agent* is also the central player in its agent-based model. It coordinates the decisions and actions of all other supply chain agents, and provides a variety of complex plans to achieve collaborative activities among the organizational units they represent.

Basically, Logistics is responsible to take orders from Customer and, in close cooperation with the workshops and the forwarding agencies, to do everything possible to satisfy them.

The essential mechanisms used for coordinating and integrating all the supply chain agents (resp. organizational units) responsible to satisfy a particular order are (1) *task decomposition*, (2) *level-wise team-forming* and (3) *continuous execution supervision*. Those keywords will be examined in the following more closely.

Task Decomposition: The scheduling tool already used in the logistics department operates on a statical company's model. It knows the general parameters of each facility and works on standardized order and activity objects. Based on this, an incoming order can be decomposed automatically into individual jobs and activities (assembly, painting, transport etc), the activities can be mapped to potential executors (Workshop 1-3, Forwarding agencies 1-2) and pre-scheduled due to the assumptions of the model. In this way, Logistics is able to assign concrete activities to each of the facilities needed to fulfil

a particular order. With the same scheduling tool, but local models and constraints, the facilities can decompose the assigned activities into low-level operations and schedule them as well.

Team Forming We adopt the view, that for optimal satisfaction of every customer order a team of facilities is founded explicitly in order to work on it. The idea of team forming between corporations, profit centers and employees has already become commonplace in most working environment. We seized upon this by having the Logistics agent performing specially structured interaction and negotiations with the agents representing the facilities. The purpose of our level-wise approach is to aggregate a team of committed executors resp. agents out of a loose set of candidates which work on the satisfaction of an individual order in a tightly cooperated manner.

Continuous Supervision: Due to the dynamics of the enterprise' world, many unpredictable events may occur in facilities during the execution of activities. If those events cannot be dealt locally, Logistics needs to be informed about that. For example, if a breakdown occurs at one facility, many customer orders may be affected in terms of delay. In order to detect important problems immediately, we have Logistics explicitly monitoring all the activities being executed at facilities. Logistics will use the information gathered to find alternatives, revise plans or negotiate with the client via the Customer agent.

The users of Logistics are meant to be one or more *logistic managers* working in the company's logistic department. So far, the operation mode is a, say, multiple first-in-first-served queue, which means the first order being received from Customer will be the first in starting to work on, but this does not infer that it will be the first to get finished. The Logistics agent (and hence the associated users) can work on multiple customer orders at the same time being in different or the same states. One can easily imagine to have a team of logistics employees who get orders assigned according to their priority, volume, type of products etc.

Logistics coordination knowledge is encoded in five, closely interrelated conversation plans which are:

- the *Logistics Execution Net* to guide a customer order until satisfaction 8.5.2
- the *Form Large Team Class* to build a team of potential contractors 8.5.3
- the *Form Small Team Class* to build a team of actual contractors 8.5.5
- the *Kickoff Execution Class* to trigger the contractors activities 8.5.7
- the *Find Contractor Class* to find a solution in case of breakdowns 8.5.10

The exact functionality of these plans can be traced in the corresponding paragraphs below.

8.3.3 The Contractor Agents

We accumulated the company's manufacturing facilities for assembling and painting as well as its forwarding agencies under the term *Contractors*. This means that they are known to Logistics as those partners in the supply chain that execute the actual activities to satisfy a customer order. In the "Fichtelberg GmbH", these are the three workshops and the two forwarding agencies, represented by three Plant agents and two Transport agents. Under the assumption of occupying a similar role, we associated in our model the same agent type with the same functionality to any of these facilities. Each agent works on activities being assigned by and negotiated with Logistics, maintains its own database and is subjected to different constraints. The agents are semi-autonomously in taking their actions and decisions: they are controlled by the end users in the facilities and coordinated in their entirety by the protocols of Logistics.

As end users of the Contractor agents, we envisage one or more executives at the contractors place, e.g. a shop floor manager at the assembly workshop or the route scheduling manager at the forwarding agency.

The coordination knowledge needed for the Contractor agents is provided by the following conversation protocols:

- the *Answer Form Large Team Class* to indicate interest in the execution of an activity 8.5.4
- the *Answer Form Small Team Class* to commit itself to execute activities 8.5.6
- the *Answer Kickoff Execution Class* to indicate the beginning of an activity's execution 8.5.8
- the *Monitor Execution Class* to report successful processing of activities or problems with their execution 8.5.9

The exact functionality of these plans can be traced in the corresponding paragraphs below.

8.4 Issues of Multi User Mode

We do not know beforehand who will be the person responsible for interacting with the supply chain agents in each of the departments. Moreover, it must be possible to associate different persons to the "agent-based workplace" as employees may get ill, change positions etc. For that purpose we introduced the concept of roles. The idea is, that a user, after linking to the supply chain application introduces himself while occupying one of the roles "customer-attendant", "logistics-manager" or "execution-supervisor".

We incorporated another agent in the scenario - the Supervisor - which has only one conversation class - the Role Acquisition Class. The only purpose is

1. to check if a user is allowed to occupy a role
2. to assign the role to the user's stub agent
3. to propagate the new identity of the stub agent to every agent concerned

In this way, the agents and conversation plans can be targeted at dealing with users that occupy roles instead of addressing user directly by name.

8.5 The Conversation Plans

According to the COOL philosophy, coordination knowledge among the agents is captured in conversation plans. Conversation plans are used to declare and to execute structured interactions among agents explicitly in order to achieve shared or individual goals. Those plans incorporate user interactions at defined moments with pre-defined decision alternatives, or spreadsheets to be filled, or notification to be displayed. In the following, all the conversation plans used by the agents in the supply chain model of the "Fichtelberg GmbH" are examined successively. It will be shown, how the plans contribute to the agents' and supply chain goals, how they are constructed and inter-related, and how the user is actively involved in their execution.

For the representation of the plans, we use state diagrams in which essential user interactions are indicated explicitly. For all the diagrams, the following legend is applicable:

- *White balls*: indicate the start state of a conversation plan
- *Gray balls*: indicate intermediate states of a conversation plan
- *Black balls*: indicate final states of a conversation plan
- *Arrows*: indicate transitions between conversation states as defined by conversation rules
- *Question Mark Screens*: indicate requests to the user to fill a spreadsheet with values and/or to make a decision
- *Line Screens*: indicate an important notification or (intermediate) result displayed on the user's screen
- *Mixed Screens*: indicate that the user will get additional information related to a request at the same time or that the user's response immediately leads to an important output

8.5.1 The Customer Conversation

We begin our journey along the plans and interactions in the agent-integrated “Fichtelberg GmbH” with the *Customer Conversation*. This protocol can be seen as the trigger of the entire supply chain process, and it is the interface to the “outer world” of clients which is ultimately affected by all the efforts made in optimal supply chain management.

Figure 8.2 illustrates the state diagram of the Customer Conversation where interactions from and to the customer service employee are indicated.

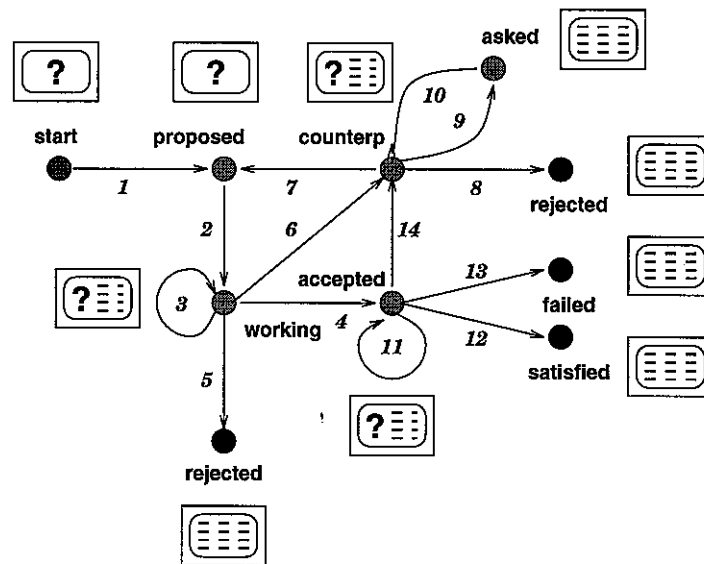


Figure 8.2: The interactive Customer Conversation

The Customer Conversation protocol is listed in detail in table 8.1. For each order, a separate instance is started, which means essentially that every client gets his own “file”. At the very beginning, the service employee will get a standard order spreadsheet onto his screen which one he needs to fill with the information from the actual customer order as being received via one of the media mentioned above (state *start*).

A sample spreadsheet for the customer service is presented in figure 8.3. We see here a hierarchical structure of pop up windows as defined in the by evaluating an instance of the pattern grammar 5.2.4. The upper window deals with general information for a customer order, the middle window shows the single items contained in an order with corresponding manipulation mechanisms, whereas the lower window is used to compose an individual item.

After composing, the order will be sent automatically to Logistics for further processing, and the protocol moves on to state *proposed* [1]. If Logistics has started to work on that particular order, it notifies Customer, and the plan moves “silently” to state *working* [2].

Form Composition

You are supposed to fill the following form:

CUSTOMER-ORDER		
ORDER-NUMBER	#XYZ/123/1	
DATE	FRI OCT 3 19:38:44 EDT 1997	
PRIORITY	1	NUMBER*
DESTINATION	Define Form #0	Press please!
CUSTOMER	Enterprise Integration Laboratory	STRING*
LINE-ITEMS	Define List #0	Press please!

OK Clear Cancel

Single Type List Composition

LINE-ITEMS

You are supposed to provide 2 elements of type 'FORM'.

Incomplete... Add Element Defaults:

(:ID INV1 :PRODUCT GARDEN-GNOME :DATE 0 :DUE-DATE 30 :QUANTITY 1 :PRICE 10 :UNIT-TRANSPORTATION-COST 0.25)

Modify Element Delete Element End of List Clear List Cancel

Form Composition

You are supposed to fill the following form.

ID	INV2	SYMBOL*
PRODUCT	NUTCRACKER	SYMBOL*
DATE	10	INTEGER*
DUE-DATE	30	INTEGER*
QUANTITY	3	INTEGER*
PRICE	5	NUMBER*
UNIT-TRANSPORTATION-COST	0.45	NUMBER*

OK Clear Cancel

Figure 8.3: Composing a Customer Order via Hierarchical Dialogs

After that, Logistics continuously keeps Customer informed about the state of the order's processing so that the information will always be available to the service staff (3) when a client calls.

Logistics may now accept the order (the plan goes to state `accepted`) [4], reject the order (the plan goes to state `rejected`) [5] or make a counter-proposal with alternative order constraints (the plan goes to state `counterp`) [6]. Logistics' decision on a particular order is immediately displayed onto the service staff's screen.

This kind of online notification is especially important in case of a counter-proposal. It occurs when Logistics has determined problems in the execution of the order such as lacking capacity or breakdowns at the Contractors. By means of the propagated alternative, the service employee is able to contact the client, and to discuss solutions with him. The client may decide to accept the alternative as it is [7], to reject it [8] or to propose an own alternative solution [7]. At any rate, the service employee just selects the corresponding template and adds the necessary values to be sent to Logistics while the protocol goes to state `proposed` again, or to state `rejected` where it terminates. Additionally, we allow the client or the service employee to ask Logistics questions about the proposed alternative, e.g. why the delivery time needs to be delayed (state `asked`) [9]. When Logistics responds, the answer will be immediately available [10] and we can go back to state `counterp`.

If we consider the regular case, which is Logistics has accepted the order, it will supervise its execution. Again, during that time it continuously keeps Customer informed about the state of the order's processing [11]. When all the individual items are manufactured and delivered, Logistics notifies Customer about that [12], the plan is finished (state `satisfied`) and the client is hopefully happy with the goods.

If during manufacturing problems occur that even Logistics cannot handle, i.e. a total breakdown of a workshop for weeks without any replacement capacities, it informs Customer that the order has failed [13], and it is the service employees task to break that gently to the client concerned. Otherwise, Logistics may again send a counter-proposal with modified order constraints [14].

8.5.2 The Logistics Execution Net

The *Logistics Execution Net* can be seen as the *master plan* of the logistics department. It interfaces all the other agents on the supply chain: the Customer agent 8.3.1 and the Contractor agents 8.3.3. All further plans and coordination activities of the Logistics agent 8.3.2 are triggered, supervised and evaluated by this protocol. The Logistics Execution Net is also the handler for all unpredictable events that cannot be dealt in sub plans.

However, as the "Fichtelberg" company is only a small business, it could base its entire logistics management on a manageable, effective and detailed concept. Beyond that, all the knowledge had been carefully documented, so that it was possible to automatize many elements of the management process. By means of the agent-based coordination mechanisms,

STATE	CONDITION	REQUEST	ACTION	RESPONSE	GO
start		Fill order form	Submit order to Log.	Confirmation	1
proposed	Rec: "working on it" from Log.				2
working	Rec: "state of order" from Log.		Note state		3
working		State of order (opt)	Retrieve state	Display state	3
working	Rec: "accept order" from Log.			Display info	4
working	Rec: "reject order" from Log.			Display info	5
working	Rec: "counter-proposal" from Log.		Note alternative		6
counterp		Accept alternative	Submit "accept" to Log.	Confirmation	7
counterp		Propose alternative	Submit "propose" to Log.	Confirmation	7
counterp		Reject alternative	Submit "reject" to Log.	Confirmation	8
counterp		Ask on alternative	Submit "ask" to Log.		9
ask	Rec: "answer" from Log.			Display answer	10
accepted	Rec: "state of order" from Log.		Note state		11
accepted		State of order (opt)	Retrieve state	Display state	11
accepted	Rec: "order satisfied" from Log.			Display info	12
accepted	Rec: "order failed" from Log.			Display info	13
accepted	Rec: "counter-proposal" from Log.			Display info	14

Table 8.1: The Course of the Customer Conversation

the logistic employees will act basically as a supervisors for the master plan's execution, and need only to intrude, at defined moments. Much of routine activities could be taken from the logistics department by this kind of support. We will refer to the users of our agent system in the logistics department as "supervisors" along the following description.

Figure 8.4 shows the state diagram of the Logistics Execution Net as well as the user-system-interactions for the logistics department

Table 8.2 presents the entire course of the Logistics Execution Net in detail. The chain starts upon receiving an order from the Customer agent in state *start*. Again, every order gets its own "file", passing the scheme until successfully processed or failed. The new order appears on the supervisor's screen and we move on to state *order-received* [1].

Now, Logistics tries to decompose the order into individual activities needed to fulfil it (such as assembly, painting, transport) and to generate a global schedule. This is done by running the embedded constraint based logistics scheduler, a legacy system which contains a complete model of the static parameters characterizing the company's manufacturing and transport facilities. This tool is fed with the order information by Logistics and will generate results which can be interpreted by the agent. If the scheduler finds a solution, this is displayed in form of a *Gantt chart* onto the supervisor's screen, and the plan goes to *order-decomposed* [2].

Figure 8.5 shows an example for Logistics' Gantt Chart by evaluating the result generated from the scheduler. Looking on the graph, we see three diagrams depicting the occupation of the available resources (assembly-plant, painting-plant, trans) by operations related to the time dimension. A job represents an individual item of a customer order to be manufactured, which in turn is composed of operations. An individual customer may have ordered many items leading to many jobs, each job is symbolized by the same color of its operations. We can

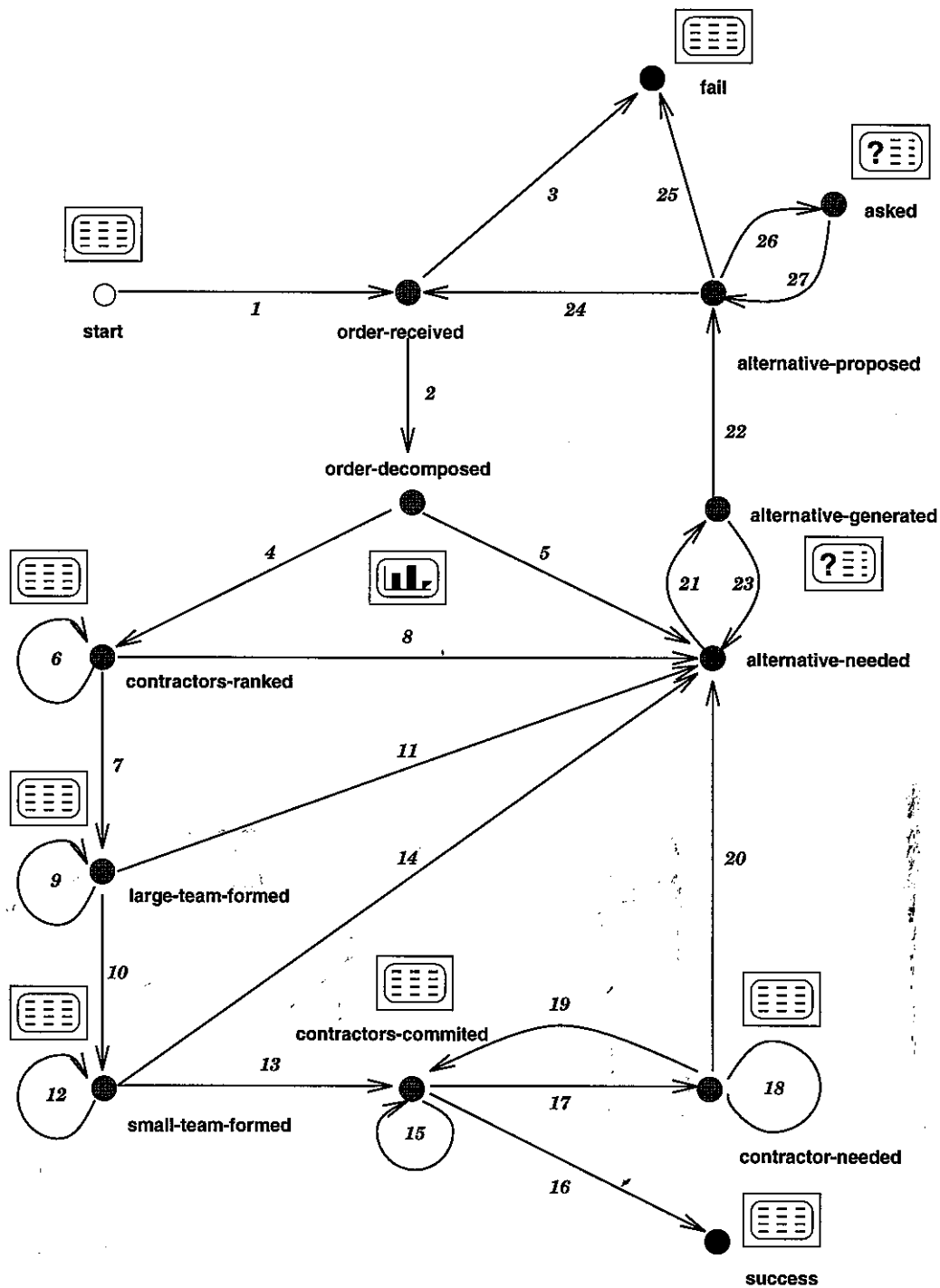


Figure 8.4: The interactive Logistics Execution Net

also see that there is no resource conflict for the operations, and the entire processing chain for each item including shipping. Note, that the scheduler operates on abstract resource names, the mapping to concrete facilities resp. agents will be made afterwards by the plan. The logistics supervisor may browse the information behind any graphical element by clicking on the buttons surrounding the graph.

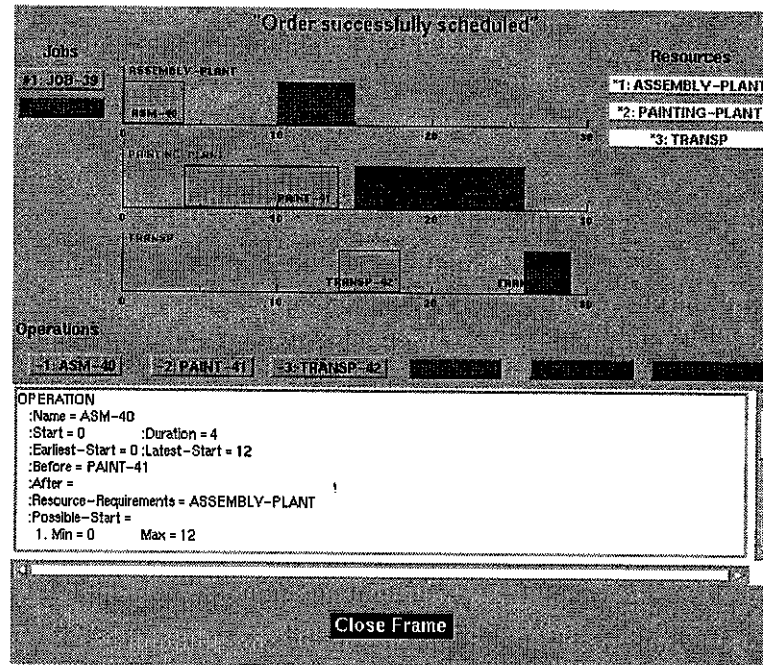


Figure 8.5: A Gantt Chart Example for the Logistics Manager

If the scheduler could not find a solution, the protocol will terminate (state fail) with a corresponding notification to the supervisor and to the Customer agent [3]. In that case, we have either a non-standard order or one that cannot be scheduled at all. How to deal with such orders, should be left to the supervisor's experience.

In state order decomposed, Logistics uses the abstract information proposed by the scheduler to identify concrete facilities, i.e. to identify the agents to contact, and to associate the generated activities to them. Upon success, we go to *contractors-ranked* [4] while displaying the potential executors and scheduled activities textually, otherwise to *alternative needed* [5] where Logistics tries to negotiate a slightly different order with Customer. (See corresponding paragraph below)

The following team forming process deals with the procedure to commit a contractor, i.e. a facility, for executing one of the pre-scheduled activities. This is done in two stages:

First, Logistics tries to form a *large team*. A large team is meant to contain all the contractors which have expressed their principal interest to execute one of the activities. This means, that after this stage, each activity may have more than one candidates for its execution. Indicating interest is not binding for a contractor. The large team is formed by sending a proposal about an activity to any potential contractor and by evaluating their responses. For that purpose, we spawn and initialize a sub plan, the *form-large-team-class* (see below). The Logistics Execution Net itself will be suspended until the sub plan terminates with a result [6]. If the large team forming process has been finished successfully, we move on to state *large-team-formed* [7] and show the composition of the large team to the supervisor. Otherwise, an alternative is needed [8].

The second step is to form a *small team*. Here, the members of the large team are requested to decide whether they want to sign a contract with Logistics on the activity they had signaled interest in. After this stage, we have exactly one activity assigned to one contractor. These are now bound to start the execution of their activity when it is due. The process of small team forming is done in a similar way through the *form-small-team-class*. Upon successful termination [9], we move on to state *small-team-formed* [10] and show the composition of the small team to the supervisor, otherwise to state *alternative-needed* [11].

After having contracts, Logistics triggers the execution of activities explicitly when they are due. At that time, it starts yet another sub plan, the *kickoff-execution-class*. The purpose for Logistics is to know whether its partners, i.e. the signer of activity contracts, are really enabled to execute them at that moment, as their constraints may have changed in the meantime. Successful termination [12] of the sub plan indicates to Logistics, that from then on all the contractors will do their assigned jobs, and we can move on to state *contractors-committed* [13] while notifying the supervisor, otherwise we go again to state *alternative-needed* [14].

In the following, Logistics watches the executors' community by waiting for message about successful job execution [15]. If all have done so, the objective of the Logistics Execution Net for an individual order "file" is achieved, and we go to the final state *success* [16] while notifying the supervisor.

However, during the execution of activities, there might be lots of unpredictable events at the executors: machine breakdowns, material delay, illness among the employees. Whenever such an event occurs, that cannot be handled locally at the contractor's place, it is requested to inform Logistics about that. In that case, the Logistics Execution Net goes to state *contractors-needed* [17] while showing the cause of the failure to the supervisor. Logistics now attempts to find a replacement for the failed activity, which means, it asks successively the other members of the *large team*, whether they are willing to sign a contract on the unfinished activity. Remember that all the large team members once have expressed their principal interest in that activity. This process is done by another sub plan, the *find-contractor-class*. Successful termination [18] indicates to Logistics, that a re-

placement contractor has been found, and Logistics can keep on watching the contractors community [19]. Otherwise, no replacement has been found for an unfinished activity and we need to look for an alternative [20]. The supervisor will get informed by corresponding messages.

The state *alternative-needed* is entered whenever problems occur during the processing of a customer order. Here, Logistics examines the nature of the problem and generates alternative customer orders by relaxing crucial constraints such as delivery time. Those alternatives are displayed in form of editable spreadsheets to the supervisor and we go to state *alternative-generated* [21]. The supervisor can either submit the alternative directly to the Customer or alter its parameters according to his own knowledge and experience before sending [22]. In both cases, the protocol moves to state *alternative-proposed*.

Agent Notification Browser	Agent Notification Browser
Performative: ANNOUNCE	Performative: TELL
Sender: LOGISTICS	Sender: LOGISTICS
Conversation: CUS:ANONYMOUS(102):0	Conversation: CUS:ANONYMOUS(102):0
Reply-With:	Reply-With:
Date: Fri Oct 03 21:35:50 EDT 1997	Date: Fri Oct 03 21:35:51 EDT 1997
Comment:	Comment:
<p>"For order:"</p> <p>CUSTOMER-ORDER</p> <p>:ORDER-NUMBER = "XYZ/123/1"</p> <p>:DATE = "FRI OCT 3 19:38:44 EDT 1997"</p> <p>:PRIORITY = 1</p> <p>:DESTINATION = NOT_DEFINED</p> <p>:CUSTOMER = "Enterprise Integration Laboratory"</p> <p>:LINE-ITEMS = [</p> <p style="padding-left: 40px;">:ID = INV1</p> <p style="padding-left: 40px;">:PRODUCT = GARDEN-GNOME</p> <p style="padding-left: 40px;">:DATE = 0</p> <p style="padding-left: 40px;">:DUE-DATE = 30</p> <p style="padding-left: 40px;">:QUANTITY = 1</p> <p style="padding-left: 40px;">:PRICE = 10</p> <p style="padding-left: 40px;">:UNIT-TRANSPORTATION-COST = 0.25</p> <p style="padding-left: 40px;">:ID = INV2</p> <p style="padding-left: 40px;">:PRODUCT = NUTCRACKER</p> <p style="padding-left: 40px;">:DATE = 10</p> <p style="padding-left: 40px;">:DUE-DATE = 30</p> <p style="padding-left: 40px;">:QUANTITY = 3</p> <p style="padding-left: 40px;">:PRICE = 8</p> <p style="padding-left: 40px;">:UNIT-TRANSPORTATION-COST = 0.45]</p> <p>"from customer"</p> <p>ANONYMOUS(102)</p> <p>"execution kick-off failed. Looking for alternative..."</p>	<p>"The following alternative order has been submitted to CUSTOMER in order to overcome the current problems"</p> <p>CUSTOMER-ORDER</p> <p>:ORDER-NUMBER = "XYZ/123/1"</p> <p>:DATE = "FRI OCT 3 19:38:44 EDT 1997"</p> <p>:PRIORITY = 1</p> <p>:DESTINATION = NOT_DEFINED</p> <p>:CUSTOMER = "Enterprise Integration Laboratory"</p> <p>:LINE-ITEMS = [</p> <p style="padding-left: 40px;">:ID = INV1</p> <p style="padding-left: 40px;">:PRODUCT = GARDEN-GNOME</p> <p style="padding-left: 40px;">:DATE = 2</p> <p style="padding-left: 40px;">:DUE-DATE = 32</p> <p style="padding-left: 40px;">:QUANTITY = 1</p> <p style="padding-left: 40px;">:PRICE = 9</p> <p style="padding-left: 40px;">:UNIT-TRANSPORTATION-COST = 0.25</p> <p style="padding-left: 40px;">:ID = INV2</p> <p style="padding-left: 40px;">:PRODUCT = NUTCRACKER</p> <p style="padding-left: 40px;">:DATE = 12</p> <p style="padding-left: 40px;">:DUE-DATE = 32</p> <p style="padding-left: 40px;">:QUANTITY = 3</p> <p style="padding-left: 40px;">:PRICE = 7.5</p> <p style="padding-left: 40px;">:UNIT-TRANSPORTATION-COST = 0.45]</p>
Close and Keep Message Close and Delete Message	Close and Keep Message Close and Delete Message

Figure 8.6: Problem Detection and Alternative Generation

Figure 8.6 illustrates the informations appearing onto the supervisor's screen in case of failing activity start at a contractor. The left window displays the related customer order

and the right window a generated alternative. If you compare both structures, you can see that the Logistic agent proposes to delay the delivery time and to reduce the prices. This alternative is meant to be sent to Customer. Remember that the customer service now is requested to negotiate with the client whether he wants to accept the alternative, to cancel the order or to make an own proposal.

The supervisor can also reject the generated alternative, which causes the Logistics agent to go back to state **alternative-needed** [23] and to look for another one, and the procedure repeats.

Once an alternative had been submitted to Customer and depending on the Customer's "decision" on behalf of the client, we move to **order-received** which starts the entire process with the modified order again **order-received** [24], or the order is declared as failed which ends its plan instance **fail** [25] with a notification to the supervisor. During the decision making process, the client (or the customer attendant) may ask questions related to the alternative order (state asked) [26] which can be answered by the supervisor [27].

STATE	CONDITION	REQUEST	ACTION	RESPONSE	GO
start	Rec: "Order" from Cust.		File order	Display order	1
order-received	Can schedule order			Gantt chart	2
order-received	Can not schedule order			Display info	3
order-decomposed	Can find contractors			Display contr.	4
order-decomposed	Can not find contractors			Display info	5
contractors-ranked	Large team not formed		Involve next candidate		6
contractors-ranked	Large team formed			Display large team	7
contractors-ranked	Large team failed			Display info	8
large-team-formed	Small team not formed		Involve next candidate		9
large-team-formed	Small team formed			Display small team	10
large-team-formed	Small team failed			Display info	11
small-team-formed	Job execution not triggered		Task next team member		12
small-team-formed	Job execution triggered			Display executors	13
small-team-formed	Trigger process failed			Display info	14
contractors-committed	Still jobs in process				15
contractors-committed	All jobs done			Display info	16
contractors-committed	Breakdown at contractor			Display info	17
contractors-committed	Team member available		Ask it for replacement		18
contractors-committed	Replacement found			Display info	19
contractors-committed	No replacement found			Display info	20
alternative-needed			Generate alternative	Display alternative	21
alternative-generated		Accept alt. Modify alt. Reject alt.	Submit alt. to Cust		22
alternative-generated					23
alternative-proposed	Rec: "accept" from Cust.		Replace order file	Display info	24
alternative-proposed	Rec: "reject" from Cust.		Close order file	Display info	25
alternative-proposed	Rec: "ask" from Cust.			Display question	26
asked		Fill answer	Submit answer to Cust.		27

Table 8.2: The Course of the Logistics Execution Net

8.5.3 The Form Large Team Class

Logistics' *Form Large Team Class* defines the interactions of Logistics with the contractor agents (Plant 1-4 and Transport 1-2) in order to form a large team for each customer order. The large team will contain all the agents that have indicated their principal interest to execute one of the activities needed to satisfy a customer order. The form large team class is a sub plan of the Logistics Execution Net 8.5.2 being activated when that one enters its state **contractors ranked**. The plan has a corresponding *Answer Form Large Team Class* 8.5.4 associated to every contractor agent (see 8.3.3).

The left figure in 8.7 illustrates the state diagram of the Form Large Team Class where interactions from and to the user in the logistics department are indicated.

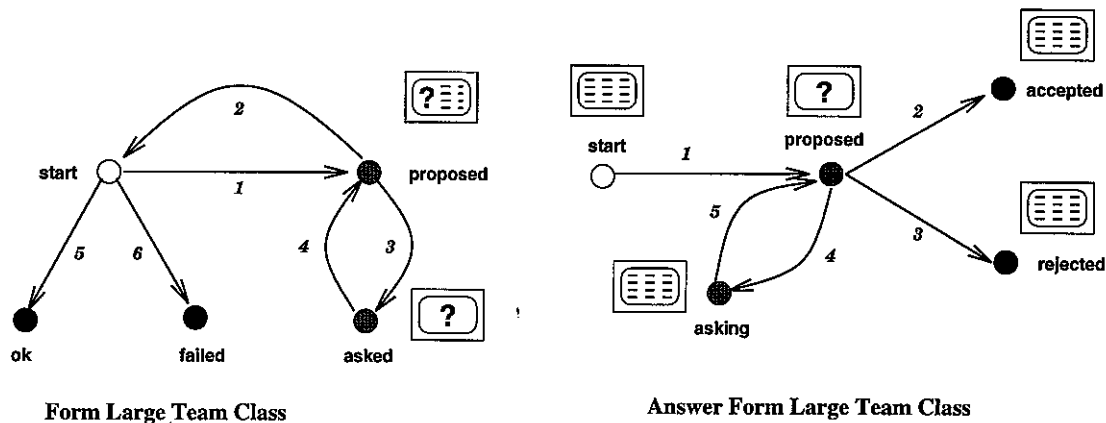


Figure 8.7: The interactive Large Team Forming

The protocol of the Form Large Team Class is listed in table 8.3. The process is very simple and automated to a great extent so that almost no user interaction is necessary. The plan is initialized with the activities needed for an individual customer order and their potential contractors (state **start**). As long as there are still activities to be distributed and candidates for each one, Logistics sends a corresponding proposal to the agent concerned and goes to state **proposed** [1]. Now, the contractor addressed may accept or reject indicating whether he wants to join the large team with the activity proposed or not, and we go back to state **start** [2] for the next contractor resp. the next activity. Upon receiving such a proposal and before decision making, every contractor may ask questions (state **asked**) [3] to Logistics which is the only moment where the supervisor has to become active by sending related answers (back to state **proposed**) [4]. If for every activity at least one contractor has been found in this way, the large team forming is successfully finished and the protocol goes to state **ok** [5] where it terminates and forwards the large team structure back to the master plan. Otherwise, no large team could be formed, the protocol goes to the final state **failed**

[6] and notifies the master plan about the failure.

STATE	CONDITION	REQUEST	ACTION	RESPONSE	GO
start	Contractors left to contact		Submit proposal to contractor		1
proposed	Rec: "accept" from contr.		Mark as team member		2
proposed	Rec: "reject" from contr.		Set next candidate		2
proposed	Rec: "ask" from contr.			Display question	3
asked		Fill answer sheet	Submit answer to contr.		4
start	Large team formed				5
start	Large team failed				6

Table 8.3: The Course of the Form Large Team Class

8.5.4 The Answer Form Large Team Class

The *Answer Form Large Team Class* attached to every Contractor agent defines the interactions with Logistics in order to form a large team for each customer order. Upon receiving a proposal from Logistics, every contractor can indicate its principal interest to execute one of the activities needed to satisfy the customer order. It corresponds to the *Form Large Team Class* 8.5.3.

The right figure in 8.7 depicts the state diagram of the Answer Form Large Team Class where system interactions with the user at the contractor's place are indicated.

In table 8.4 you can trace the course of the Answer Form Large Team Class. The plan involves a decision making from the executive responsible at the facility concerned. It is activated when Logistics has sent a proposal about an activity intending to involve a contractor into a large team (state **start**). The proposal is displayed to the executive, and we move on to state **proposed** [1]. According to his knowledge about the capacities needed for the activity's execution, the executive can decide whether "his" facility wants to "join" the large team (the plan moves to state **accepted** [2] or not (the plan moves to state **rejected** [3]. Accepting to join the large team is not bounding for a later execution of the activity. He may also forward questions to Logistics about the activity proposed (plan goes to **asking**) [4], and can use the answer received in coming to a decision (plan goes back to **proposed**) [5].

STATE	CONDITION	REQUEST	ACTION	RESPONSE	GO
start	Rec: "proposal" from Log.			Display act.	1
proposed		Join large team	Submit "accept" to Log.	Confirmation	2
proposed		Refuse large team	Submit "reject" to Log.	Confirmation	3
proposed		Ask about activity	Submit "ask" to Log.		4
asking	Rec: "answer" from Log.			Display answer	5

Table 8.4: The Course of the Answer Form Large Team Class

Figure 8.8 shows how the decision making process is featured. The executive may choose from any of the items in the upper list. In our example we have selected the "accept"

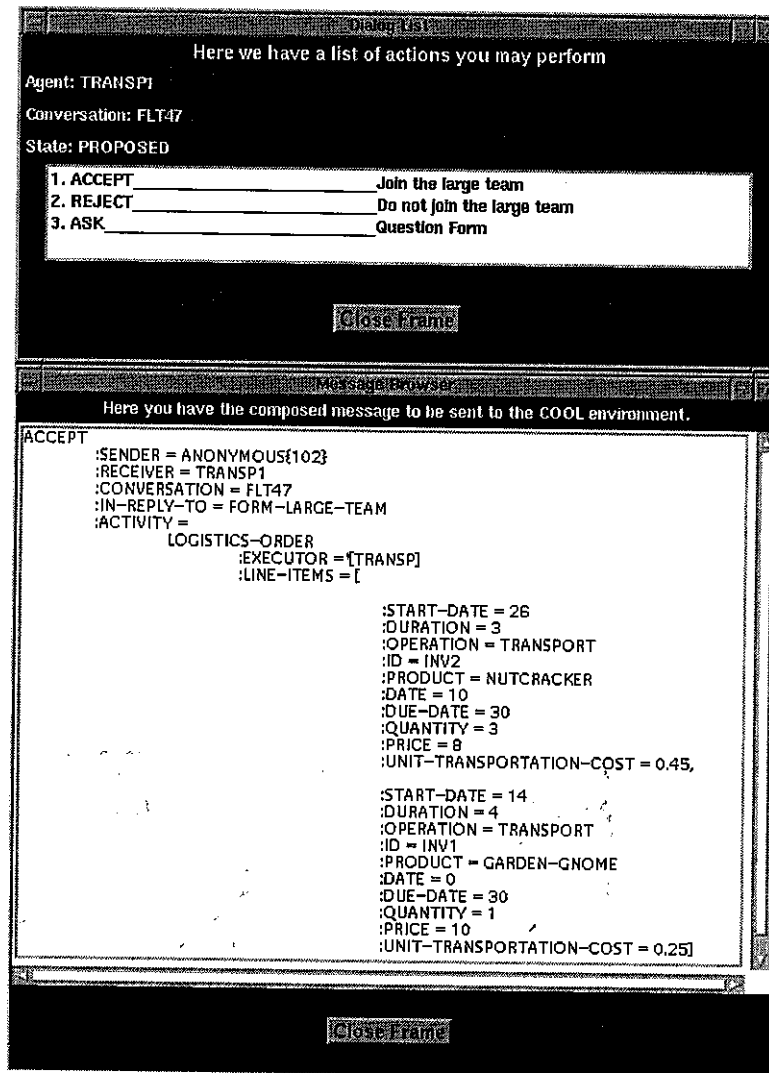


Figure 8.8: Decision on Large Team Participation

template. There is nothing else needed than the decision in itself, and the protocol does not allow manipulation of values in this state whatsoever. So, we just display the information about the activity concerned to the executive and he can decide whether he wants to submit this or rather another message to Logistics.

8.5.5 The Form Small Team Class

Analogous to the Form Large Team Class, Logistics' *Form Small Team Class* defines the interactions of Logistics with the contractor agents (Plant 1-4 and Transport 1-2) in order to form a small team for each customer order. In the small team, each activity will have assigned exactly one executor. The plan is activated by the Logistics Execution Net 8.5.2 when that one enters the state **large-team-formed** (see above). Correspondingly, each contractor provides a *Answer Form Small Team Class* 8.5.6.

The left figure in 8.9 shows the state diagram of the Form Small Team Class where interactions from and to the user in the logistics department are indicated.

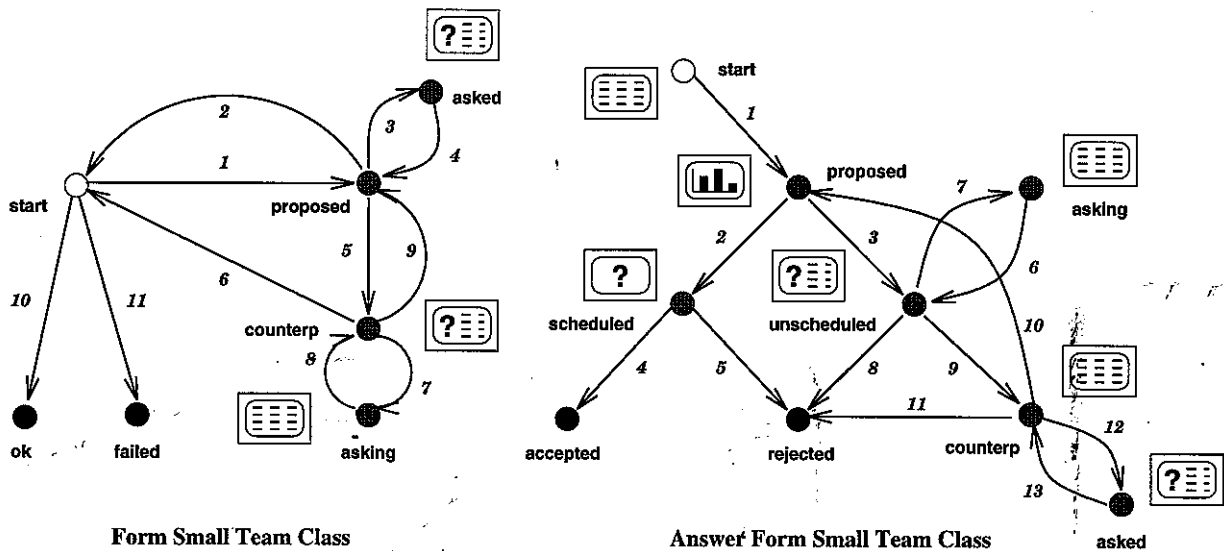


Figure 8.9: The interactive Small Team Forming

Table 8.5 presents the course of the Form Small Team Class. Unlike the large team forming process, this protocol is slightly more complex and requires more user interaction. Thus, we reflect the nature of the contractors' commitments sought. The plan is initialized with the *large team* as generated by the previous process (state **start**). Again, as long as there are still activities to distribute which do not have an executor yet, Logistics sends a proposal to the next member of the large team and goes to state **proposed** [1]. In turn, the contractor addressed may accept or reject the proposal indicating whether he wants to join

the small team with the activity proposed or not, and we go back to state `start` [2] for the next activity. Upon receiving such a proposal and before decision making, every contractor may ask questions (state `asked`) [3] to Logistics and the supervisor is requested to send related answers (back to state `proposed`) [4].

A contractor may also send a counter-proposal in terms of wishing to modify activity parameters that are not consistent with its current constraints. In that case, we start a *negotiation process* between Logistics and the contractor, being composed of several cycles of proposals, counter-proposals and raising questions until one of them has eventually accepted or rejected to get (or to be) involved in the small team. This is the essential part of the user interaction in the Form Small Team Class. Upon receiving a counter-proposal (while moving to state

`counterp` [5], the supervisor can inspect it, and decide what to do. He can choose from the following alternatives: If he feels the (current) counter-proposal is feasible resp. impossible, he can accept resp. reject it, and the negotiation with that contractor is finished. The protocol can move on to deal with the next activity [6]. A supervisor may have questions concerning the counter-proposal to the contractor before coming to a decision (state `asking`) [7] and use the answer provided to come to a decision (back to state `counterp`) [8]. Moreover, the supervisor may use the information provided in the counter-proposal to create an own alternative proposal (state `proposed`) [9], which in turn can be accepted, rejected, questioned or counter-proposed by the contractor.

If for every activity exactly one contractor could be found in this way, the small team forming is successfully finished and the plan goes to state `ok` [10] where it terminates and forwards the small team object to the master plan. Otherwise, no small team could be formed, the plan goes to `failed` [11] where it terminates and notifies the master plan about the failure.

STATE	CONDITION	REQUEST	ACTION	RESPONSE	GO
<code>start</code>	Lt members left to contact		Submit proposal to contr.		1
<code>proposed</code>	Rec: "accept" from contr.		Mark as team member		2
<code>proposed</code>	Rec: "reject" from contr.		Set next candidate		2
<code>proposed</code>	Rec: "ask" from contr.			Display question	3
<code>asked</code>		Fill answer sheet	Submit answer to contr.		4
<code>proposed</code>	Rec: "counterp" from contr.			Display it	5
<code>counterp</code>		Use "accept counterp"	Mark as team member Submit "accept" to contr.		6
<code>counterp</code>		Use "reject counterp"	Set next candidate Submit "reject" to contr.		6
<code>counterp</code>		Use "ask contractor"	Submit question to contr.		7
<code>asking</code>	Rec: "answer" from contr.			Show answer	8
<code>counterp</code>		Use "propose altern."	Submit altern. to contr.		9
<code>start</code>	Small team formed				10
<code>start</code>	Small team failed				11

Table 8.5: The Course of the Form Small Team Class

8.5.6 The Answer Form Large Team Class

The *Answer Form Large Team Class* attached to every Contractor agent defines the interactions with Logistics in order to form a small team for each customer order. Upon receiving a proposal from Logistics, every contractor can commit himself to execute one of the activities needed to satisfy the customer order. It corresponds to the *Form Small Team Class* 8.5.5.

The right figure in 8.9 illustrates the state diagram of the Answer Form Small Team Class where interactions from and to the user at the contractor's place are indicated.

Table 8.6 depicts the course of the Answer Form Large Team Class. The plan involves several decision making processes from the executive responsible at the facility concerned. It is activated when Logistics has sent a proposal about an activity intending to involve a contractor into a small team (state **start**). The proposal is displayed to the executive responsible, and we move on to state **proposed** [1]. Now the Contractor agent runs the embedded scheduler. As mentioned earlier, each contractor has a copy of the scheduling tool configured to its local parameters. The scheduler decomposes the activity proposed into operations and tries to find a schedule due to the facility's resources. If the scheduler finds a solution, a *Gantt Chart* will be displayed to the executive and the protocol goes to state **scheduled** [2]. Otherwise a notification will be shown and we move to state **unscheduled** [3].

Figure 8.10 shows another Gantt Chart example, this time being created for a workshop agent. Each job is related to manufacturing a single item or a margin of the same items (green nutcracker, big Christmas pyramids etc.). The process is decomposed into low-level operations, for example cutting the wood or glueing the components. Again, there are no resource conflicts, the proposed activity could be taken.

If the proposed activity could be scheduled, the executive can now declare to "join" the small team, and the protocol ends up in state **accepted** [4]. In doing so, he practically commits his facility towards Logistics. Nonetheless, he might have reasons to refuse the small team, even if the activity is perfectly scheduled. In this case, we go to state **rejected** [5].

When no schedule could be found at all, the executive needs to decide whether to reject the small team, or to make a counterproposal. But he may want to obtain more details first from Logistics (plan goes to **asking**) [6], and can use the answer received for coming to a decision (plan goes back to **unscheduled**) [7]. Refusing the small team will finish the protocol in state **rejected** [8]. If he decides to submit a counter-proposal, the executive gets a spreadsheet onto his screen, where he can edit the parameters of the original activity. While submitting the counter-proposal to Logistics, we move on to state **counterp** [9].

Now, another *negotiation process* is initiated as Logistics (on behalf of the logistics supervisor) may now decide what to do. If Logistics accepts the counter-proposal, it sends the alternative activity back for being rescheduled with the embedded scheduler (we go back to state **proposed**) [10]. If it rejects the counter-proposal, the decision is displayed to the exec-

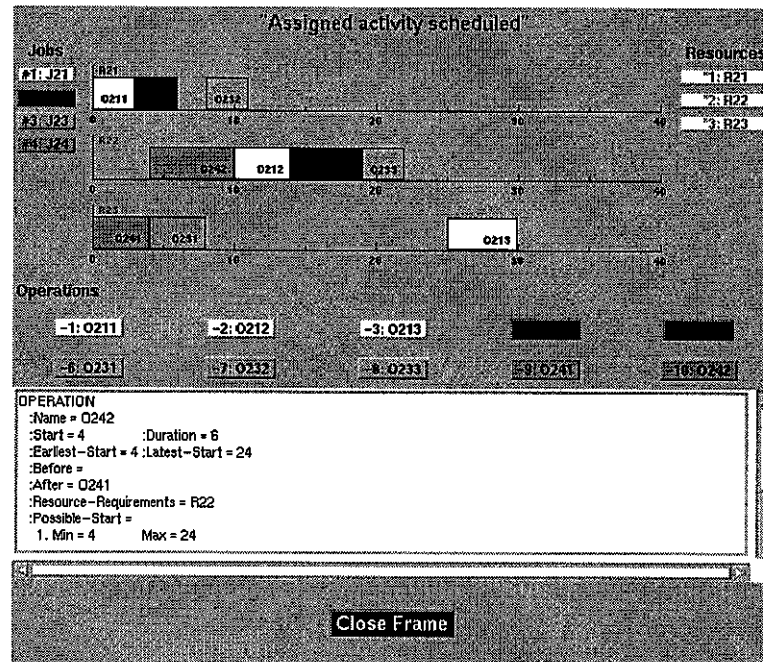


Figure 8.10: A Gantt Chart Example for the Workshop Manager

utive and the protocol terminates in state rejected [11]. Also, Logistics may have questions to the counter-proposal, upon which we end up in state asked [12] where the executive is requested to answer them while going back to counterp [13] waiting for the next action of Logistics.

8.5.7 The Kickoff Execution Class

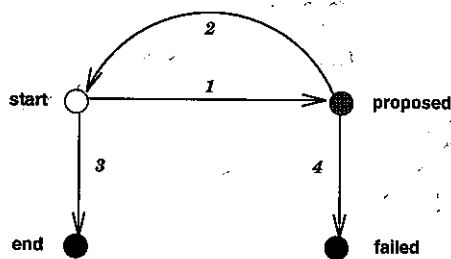
With the *Kickoff Execution Class* Logistics makes every member of a *small team* to start its activity, when the related customer order becomes due. Even though every contractor in the small team has committed himself in the first place, there might have been lots of changes in the meantime. So, Logistics tries to figure out once again, if the items of the customer order concerned can now be produced and delivered. The corresponding plan at the contractors is, surprisingly, the Answer Kickoff Execution Class 8.5.8.

The left figure in 8.11 pictures the state diagram of the Kickoff Execution Class with interactions from and to the user at logistics department.

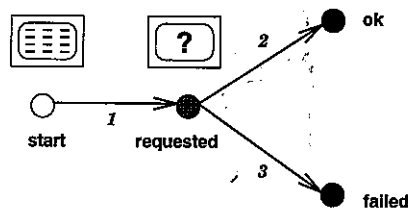
Table 8.7 presents the course of the Kickoff Execution Class. We did not introduce any user interaction here, as the plan can be started and executed automatically. After all, the logistics manager needs only to be informed, whether *all* contractors in the small team have

STATE	CONDITION	REQUEST	ACTION	RESPONSE	GO
start	Rec: "proposal" from Log.			Display act.	1
proposed	Act. can be scheduled			Display Chart	2
proposed	Act. cannot be scheduled			Display info	3
scheduled		Join small team	Submit "accept" to Log.	Confirmation	4
scheduled		Refuse small team	Submit "reject" to Log.	Confirmation	5
unscheduled		Question about act.	Submit "ask" to Log.		6
asking	Rec: "answer" from Log.			Display answer	7
unscheduled		Refuse small team	Submit "reject" to Log.	Confirmation	8
unscheduled		Make counterprop.	Submit "counterp" to Log.	Confirmation	9
counterp	Rec: "accept" from Log.			Display info	10
counterp	Rec: "reject" from Log.			Display info	11
counterp	Rec: "ask" from Log.			Show question	12
asked		Fill answer form	Submit "answer" to Log.		13

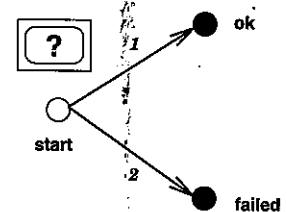
Table 8.6: The Course of the Answer Form Small Team Class



Kickoff Execution Class



Answer Kickoff Execution Class



Monitor Execution Class

Figure 8.11: The interactive Activity Execution

started their activities or not, which is already done in the *Logistics Execution Net*, when this little protocol terminates 8.5.2. Logistics simply contacts every contractor in the small team successively and goes to state **requesting** [1]. When a contractor has indicated the activation of his job, we go back to **start** [2] for asking the next small team member. If all members have responded positively, we go to state **end** [4] and notify the master plan. In case a contractor is unable to start its activity, we go to the final state **failed** [3] and notify the master plan as well.

STATE	CONDITION	REQUEST	ACTION	RESPONSE	GO
start	Team members left to contact		Submit "achieve" to next contr.		1
requesting	Rec: "started" from contr.		Mark as contacted		2
requesting	Rec: "failed" from contr.				3
start	All team members contacted				4

Table 8.7: The Course of the Kickoff Execution Class

8.5.8 The Answer Kickoff Execution Class

The *Answer Kickoff Execution Class* attached to every Contractor agent defines the interactions with Logistics when an activity should be started. It corresponds to the *Kickoff Execution Class* 8.5.7.

The middle figure in 8.11 illustrates the state diagram of the Answer Kickoff Execution Class and the interactions from and to the user at the contractor's place.

The simple protocol of the Answer Kickoff Execution Class is listed in table 8.8, and includes one action of the contractors' executive. He must decide now, if the current capacities really allow to produce resp. to deliver the items concerned. Upon receiving an activation request from Logistics, the activity is displayed once again, and we move to state **requested** [1]. Now the executive can make his decision in selecting either the "started" or the "failed" message to be sent to Logistics which leads to the corresponding states **ok** [2] or **failed** [3]. In the first case a *Monitor Execution Class* (see below) will be started for the contractor, so that he can notify Logistics immediately about problems with the execution or successful processing.

STATE	CONDITION	REQUEST	ACTION	RESPONSE	GO
start	Rec: "achieve" from Log.			Display act.	1
requested		Act successfully started	Submit "started" to Log. Start Monitor Exec Class	Confirmation	2
requested		Act. could not be started	Submit "failed" to Log.	Confirmation	3

Table 8.8: The Course of the Answer Kickoff Execution Class

8.5.9 The Monitor Execution Class

Finally, with the *Monitor Execution Class* attached to each contractors it is possible to notify Logistics about problems or successful processing related to each activity the contractor has started to work on. In case of a problem, Logistics can become active in terms of rescheduling the activity, finding a replacement, contacting the Customer etc. On the other hand, it will know when a customer order has been satisfied and is ready for shipping.

The simple state diagram and user interaction mode of the Monitor Execution Class is shown in the right figure in 8.11. At any time, the executive can inform Logistics about problems during the execution that cannot be handled locally using the corresponding message template (state failed) [2]. And he should indicate when the work on an activity has been done (state ok) [1].

Table 8.9 presents the course of the Monitor Execution Class.

STATE	CONDITION	REQUEST	ACTION	RESPONSE	GO
start		Activity finished	Submit "satisfy" to Log.	Confirmation	1
requested		Problems with act.	Submit "failed" to Log.	Confirmation	2

Table 8.9: The Course of the Monitor Execution Class

8.5.10 The Find Contractor Class

The *Find Contractor Class* is used by Logistics if one contractor has failed to execute an assigned activity, i.e. its executive has announced to Logistics to have grave problems. In that case, Logistics is required to find an adequate replacement to do the job nonetheless, as a client is waiting for his products. If this is possible, the activity is simply re-assigned to the replacement contractor without notifying the client. Otherwise, the related customer order cannot be satisfied in time, and Logistic will need to negotiate via Customer with the client about alternatives.

Figure in 8.12 illustrates the state diagram of the Find Contractor Class where interactions from and to the user in the logistics department are indicated.

The detailed course of the Find Contractor Class can be found in table 8.10. The protocol uses essentially the same interaction mechanisms as the *Form Small Team Class* 8.5.6. Consecutively, the Contractor addressed can use the *Answer Form Small Team Class* 8.5.6 again to interact with Logistics on that matter. The only differences are the initialization with the large team excluding the contractor who has just failed, and the protocol's termination as soon as one contractor has agreed to take over the dangling activity. In state **start**, if there are still potential replacement contractors in the (reduced) large team, Logistics proposes the activity to the next one. The supervisor will be notified and we go to state **proposed** [1]. If all contractors have been contacted in vain (or the reduced large team contains no candidates at all), the supervisor gets informed as well before the protocol terminates in state **failed**

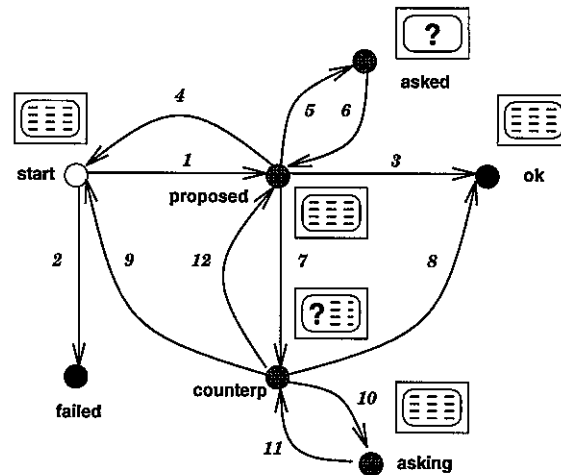


Figure 8.12: The interactive Contractor Replacement

[2]. Now, the contractor contacted has to decide whether he wants to take the dangling activity. As this is done using the *Answer Form Small Team Class*, including scheduling, the executive will know if the resources there are sufficient to add an additional activity. As Logistics does not know the contractor's constraints, it just waits for a particular message and acts according to the following scheme:

- Contractor accepts to take over the activity: We display this information, and the protocol terminates successfully in **ok** [3].
- Contractor rejects to take over the activity: We display this information, and go back to state **start** [4] in order to try the next potential candidate (if he exists).
- Contractor has a question about the activity: We show the question, and go to state **asked** [5] where the Logistics supervisor may fill the answer template, which is submitted to the contractor before going back to **proposed** [6] waiting for its decision again.
- Contractor sends a counter-proposal related to the activity: After displaying the counter-proposal, we go to state **counterp** [7]. Now, the supervisor himself needs to make a decision on whether the alternative proposed makes the contractor an adequate replacement.

According to his knowledge and experience, the supervisor may ...

- accept the counter-proposal: Thus, we have found a replacement contractor for the dangling activity, and end up in **ok** [8].

- reject the counter-proposal: We notify the unwilling contractor and go back to state **start** [9] in order to try the next potential candidate (if he exists).
- ask the contractor about its counter-proposal: After submitting the question, we go to state **asking** [10]. Upon response from the contractor, we display it and go back to state **counterp** [11] where the supervisor can choose again what to do next.
- make an own proposal related to the counter-proposal: The supervisor will obtain a spreadsheet where he can relax or manipulate activity constraints in order to submit it as a new proposal to the same contractor. Remember, Logistics tries at that moment desperately to find an executor for a dangling activity, so that the related customer order can still be shipped on time. In going back to state **proposed** [12], the complete cycle starts all over again, until either one of the candidates has agreed eventually or no candidate is left to contact.

STATE	CONDITION	REQUEST	ACTION	RESPONSE	GO
start	Candidate left to contact		Submit proposal to contr.	Display info	1
start	No cand. left to contact			Display info	2
proposed	Rec: "accept" from contr.			Display info	3
proposed	Rec: "reject" from contr.		Set next candidate	Display info	4
proposed	Rec: "ask" from contr.			Display quest.	5
asked		Fill answer sheet	Submit answer to contr.		6
proposed	Rec: "counterp" from contr.			Display it	7
counterp		Use "accept counterp"	Submit "accept" to contr.	Confirmation	8
counterp		Use "reject counterp"	Set next candidate Submit "reject" to contr.		9
counterp		Use "ask contractor"	Submit question to contr.		10
asking	Rec: "answer" from contr.			Show answer	11
counterp		Use "propose altern."	Submit altern. to contr.	Confirmation	12

Table 8.10: The Course of the Find Contractor Class

8.6 Summary

With the demonstration scenario, we have shown how people can interact with each other in pursuing a complex task like supply chain management. The activities are fully and explicitly coordinated by the underlying COOL agent application. Elements like team-forming, workflow management and the integration of legacy systems exhibit the variety of possible chances to capture the processes of real working environments in agent-based structures.

Chapter 9

Final Remarks

9.1 Summary

9.1.1 Theoretical Background

The theoretical part of the thesis encompassed an introduction to agent technology, the application of agent technology in a business context and the presentation of the COOrdination Language.

The first chapter outlined agent technology as a new paradigm which allows for the development of heterogenous distributed computational systems where the individual components can pursue their goals autonomously, communicate with each other at a highly abstract level, plan and coordinate their actions explicitly and involve end users actively in a collaborative process, thus address a variety of real world problems.

In the following, we identified enterprise operation as a promising application area for multi-agent systems. Focusing on supply chain management, a model has been presented how users and agents can be linked towards a virtual platform across geographical and functional boundaries, leading to a significant improvement of ...

The COOrdination Language, developed at the Enterprise Integration Laboratory has been chosen as the base for building a Generic User Agent, an interface agent to merge users and multi-agent applications onto a universal and abstract interaction platform. With COOL, distributed systems of agents can be defined where agents coordinate their actions explicitly through conversations. Well-founded mechanisms such as state-based agent execution, information sharing through KQML messages and built-in interpreters make COOL an ideal framework to implement applications for business purposes.

9.1.2 Practical Work

The practical work was focussed on identifying basic mechanisms for interactions between users and COOL multi-agent applications. Together with elementary administration functions, by taking a variety of user-related issues into account and exploiting agent-oriented methodology consequently they lead to the design and implementation of a Generic User Agent (GenUA).

The interface agent enables participation of multiple users in multiple applications relying on a interaction model where the user is guided and assisted along the entire execution process. Elements like a comprehensive history and offline information provide an adequate user service. Authorization at several levels and transactions management protect both the application elements and the user against unauthorized access.

GenUA comes up with a highly flexible architecture. New components and functionalities can be easily added without re-implementation of the rest. Working on declarative knowledge structures and incorporating a communication driver concept, GenUA can talk to a variety of different presentation systems at the same time.

The comprehensive Java applet being accessible from anywhere through a Web browser, demonstrates how users at their workplace can be linked to multi-agent applications in an intuitive manner mediated by the Generic User Agent. The applet provides as much transparency as needed, seconds the process of interaction guidance and, nevertheless, allows the user to make decisions about the next action.

Though based on a simple company's structure with a lot of assumptions, the distributed supply chain demonstrator has shown how information exchange, coordination protocols, legacy system integration and social structures can be captured in a promising multi-agent application and how it is used to support end-users in this domain in their daily work. It has been demonstrated how a collaborative environment between users and agents in a real world context may look like.

9.1.3 Meeting the Objective

GenUA allows for building multi-agent applications by means of the COOrdination language that can be directly employed in enterprise scenarios. While COOL reflects the coordinative aspects among organizational units, GenUA links agents and users towards a virtual platform. With a generic and extendible architecture, GenUA can be tailored to specific needs of organizations.

Providing access from the Web to the problem solving competency of agents with guided interactions, a truly collaborative system has been created, where users and agents are working together in order to achieve the enterprise's goals.

The extended supply chain demonstrator also gives the Integrated Supply Chain Management project (ICSM) a real-world prototype for further research on involving users in

agent-integrated enterprises.

With these aspects, the thesis objectives as described in section 1.1 have been met.

9.2 Future Work

9.2.1 Extending the Generic User Agent

The Generic User Agent as presented in this paper can only be seen as a prototype. Further research needs to focus particularly on incorporating explicit models of users in terms of customization, adaptation and preference management. This can be done, for example, by introducing a management layer between presentation system and Generic User Agent instances which maps users or user groups to specifically tailored instances. This involves, that GenUA instances will need to coordinate their activities among each other either directly or facilitated by the manager.

Another key for providing more sophisticated and application specific user interaction lies in the extension of the pattern grammar (for example by concrete graphical descriptions, so that agent input can be provided as diagrams or figures) and transferring HTML specifications for both input requests and execution results between COOL environments and Web browsers.

The idea of multimedial offline messaging may be enriched by a peer to peer communication and document exchange between users. Another factor is the integration of different communication platforms, particularly CORBA and HTTP.

9.2.2 Improving, Varying and Adopting the GenUA Interfaces

The Java applet used to communicate with multi-agent applications through a Web browser is an example how navigation and interaction can be presented in a transparent, abstracting and intuitive way for the benefit of the end user. Assuming that representation descriptions are created directly in the respective agent application, we need to provide corresponding graphical components, particularly in terms of editable charts, tables or images. Instead of or additionally to the textual description, conversation classes should be represented in form of a finite graph, and analogous, ongoing conversations as constantly updated ones. This will provide a better notion to the user about the execution progress. The history may come up with a comprehensive search mask so that the user can retrieve interaction information from the past similar to querying a database.

Also, experiments should be made to emulate and improve the mechanisms given by the standard Java applet on different presentation systems. The specific focus should be directed to evaluate HTML specification through HTTP or CGI, and to build X widget or Windows components that can communicate with GenUA.

The most promising challenge would be to develop a Java applet which allows for development of COOL multi-agent applications from the Web. This includes the specification of

agents and coordination protocols and the definition of user interactions. All what is needed is to implement the corresponding data structures, to ensure a transfer through the Generic User Agent and an evaluation in the COOL environment. As the effects can be directly tested through the user interface applet, we will come to a truly comfortable testbed while the implementation results are immediately deployable into the working environment without any changes.

9.2.3 Extending and Devising Demonstration Scenarios

The presented interactive supply chain scenario is restricted in both width (number of participating agents and users) and depth (capabilities of agents, integration of legacy systems). For a real-world reflection, we'll need to

- multiply the number of agents (plants, customer service)
- add new specialized agents (marketing, inventory, purchasing, production, distribution, finance)
- introduce a number of characteristic features for supply chain management (demand forecasts, inventory management, production supervision, route planning, cash flow management)
- provide interfaces to further legacy systems (inventory databases, bar code scanner, production and quality sensoric, planning and optimization tools, performance analyzer)
- identify prospective end-users in real-world supply chain management, acquire knowledge about their task structure and model adequate interactions for them

Supply chain management is only one of the numerous domains where agents can be employed in a business context. Other scenarios may encompass, for example:

- an insurance office system where agents control the workflow on insurance cases
- a manufacturing system where agents plan and schedule the production, feed CNC machines and assembly lines with corresponding data, supervise the execution and take corrective actions when required
- a transport agency where one or more centralized agents determine route, schedule and batch size proposals for incoming orders, negotiate their feasibility with via radio with agents in the vehicles concerned and acquire feedback from them during the actual transport

Chapter 10

Acknowledgements

This thesis has been written in relation to the Agent Building Shell project at the Enterprise Integration Laboratory (EIL), University of Toronto, Canada, and the current research on intelligent agent architectures at the Distributed Artificial Intelligence Laboratory (DAI lab), Technical University of Berlin, Germany. Parts of the thesis are also related to the Intelligent Supply Chain Management project at the EIL.

Research at the EIL is supported in part by the Manufacturing Research Corporation of Ontario, Natural Science and Engineering Research Council, Digital Equipment Corp., Mitel Corp., Micro Electronics and Computer Research Corp., Spar Aerospace, Carnegie Group, and Quintus Group. The DAI lab is member of the German Research Net and cooperates with the Fraunhofer Institute. Research at the DAI lab is supported by the German Telecom, DeTeBerkom, Sun Microsystems, IBM Germany, Daimler Benz AG, and Siemens AG.

I would like to thank: Dr.-ing. Sahin Albayrak and Prof. Mark S. Fox for giving me the chance to write my thesis at the University of Toronto, Dr. Mihai Barbuceanu for invaluable support and guidance through the work, fellow students and team members for providing a friendly and cooperative work environment, and of course friends and family for motivation and standing by me.

Bibliography

- [1] Nicholas M. Avouris, Marc H. van Liedekerke, Georgios P. Lekkas and Lynne E. Hall. User Interface Design for Cooperating Agents in Industrial Process Supervision and Control Applications. *International Journal of Man-Machine Studies*, 38(5), pp. 873-890, 1993.
- [2] R. M. Baecker (editor). *Readings in Groupware and Computer-Supported Cooperative Work*. Morgan Kaufmann Publishers: San Mateo, CA, 1993.
- [3] Mihai Barbuceanu and Mark S. Fox. Capturing and Modeling Coordination Knowledge for Multi-Agent Systems. *International Journal of Cooperative Information Systems*, Vol.5 Nos.2-3 pp 273-314, 1996.
- [4] Mihai Barbuceanu and Mark S. Fox. The Specification of COOL: A Language for Representing Cooperation Knowledge in Multi-Agent Systems. Enterprise Integration Laboratory, University of Toronto, Internal report, 1996.
- [5] Mihai Barbuceanu and Mark S. Fox. Integrating Communicative Action, Conversations and Decision Theory in a Coordination Language for Multi-Agent Systems, Internal paper, 1996.
- [6] P.R. Cohen and H. Levesque. Intention is Choice with Commitment. *Artificial Intelligence* 42, 1990, pp. 213-261.
- [7] T. Finin et al., Specification of the KQML agent communication language, The DARPA Knowledge Sharing Initiative, External Interfaces Working Group, 1992.
- [8] Mark S. Fox. A Common-Sense Model of the Enterprise, *Proceedings Industrial Engineering Research Conference*, 1993.
- [9] Mark S. Fox. 60 Month Progress Report: NSERC Industrial Research Chair in Enterprise Intergration. Enterprise Integration Laboratories, University of Toronto, 1996.
- [10] Ram Ganeshan and Terry P. Harrison. An Introduction to Supply Chain Management. Penn State University. http://silmaril.smeal.psu.edu/misc/supply_chain_intro.html.

- [11] M. R. Genesereth, R.E. Fikes. Knowledge Interchange Format, Version 3.0, Reference Manual, Computer Science Department, Stanford University, Technical Report Logic-92-1, 1992.
- [12] M. R. Genesereth and S. P. Ketchpel (1994). Software agents. *Communications of the ACM*, 37(7):48-53.
- [13] Aimo Hinkkanen, Ravi Kalakota, Porama Saengcharoenrat, Jan Stallaert and Andrew B. Whinston. Distributed Decision Support Systems for Real Time Supply Chain Management using Agent Technologies. <http://yama.bus.utexas.edu/ejou/articles/art.1.html>.
- [14] Nicholas Robert Jennings. *Cooperation in Industrial Multi-Agent Systems*. Singapore: World Scientific, 1994.
- [15] . Lee and C. Billington. Managing Supply Chain Inventory: Pitfalls and Opportunities. *Sloan Management Review*, pp. 65-73, Spring 1992.
- [16] Lynne E. Hall. User Design Issues for Distributed Artificial Intelligence. In: G.M.P. O'Hare and N.R. Jennings (editors). *Foundations of Distributed Artificial Intelligence*, pp. 543-556, John Wiley & Sons, Chichester, England, 1996.
- [17] P. Maes. Social Interface Agents: Acquiring competence by learning from users and other agents. In: O. Etzoni (editor). *Software Agents - Papers from the 1994 Spring Symposium* (Technical Report SS-94-03). pp. 71-78.
- [18] H. V. D. Parunak. Applications of distributed artificial intelligence in industry. In: O'Hare, G. M. P. and Jennings, N. R., editors, *Foundations of Distributed AI*. John Wiley & Sons: Chichester, England, 1996.
- [19] J.A. Sánchez, F.S. Azevedo and J.J. Leggett. PARAgent: Exploring the Issues in Agent-based User Interfaces. *Proceedings of the First International Conference on Multi-agent Systems (ICMAS'95)*, pp. 320-327, San Francisco, CA, June 1995.
- [20] Philip B. Schary and Tage Skjott-Larsen. *Managing the Global Supply Chain*. Handelshøjskolen Forlag, 1995.
- [21] J. Searle. *Speech acts: An Essay in the Philosophy of Language*. Cambridge University Press. Cambridge, UK, 1969.
- [22] S. R. Rosenschein and L. P. Kaelbling. A Situated View of Representation and Control. *Artificial Intelligence* 73 (1-2) pp 149-173, 1995.
- [23] Janyashankar M. Swaminathan, Norman M. Sadeh, and Stephen F. Smith. *Information Exchange in the Supply Chain*. 1995

- [24] Janyashankar M. Swaminathan, Stephen F. Smith, and Norman M. Sadeh. A Multi Agent Framework for Modeling Supply Chain Dynamics. Technical Report, The Robotics Institute, Carnegie Mellon University, 1996.
- [25] A.S. Rao and M.P. Georgeff. Modeling Rational Agents within a BDI Architecture. In: R. Fikes and E. Sandewall (editors). Proceedings of Knowledge Representation and Reasoning, KR & R-91, pp. 473-484.
- [26] Y. Shoham. Agent-Oriented Programming. Artificial Intelligence 60, 1993, pp. 51-92.
- [27] J.Searle. Speech acts: An Essay in the Philosophy of Language. Cambridge University Press, Cambridge, UK, 1969.
- [28] K. Sycara. Multi-agent compromise via negotiation. In: Les Gasser and Michael N. Huhns, editors, Distributed Artificial Intelligence, Volume II, pp. 119-137, Pitman Publishing, London, 1989.
- [29] Doug Thomas and Paul Griffin. Coordinated Supply Chain Management. European Journal of Operational Research, vol. 94, pp. 1-15, 1996.
- [30] Michael Wooldridge, Nick Jennings. Intelligent Agents: Theory and Practice. Knowledge Engineering Review Volume 10 No 2, June 1995.
- [31] G. Zlotkin, J. S. Rosenschein. Negotiation and task sharing among autonomous agents in cooperative domains. Proceedings of IJCAI-89, pp. 912-917, Detroit, MI, 1989.

Appendix A

Terminology

This section describes the terminology used in the context of COOL and GenUA throughout the thesis paper.

Application: A (distributed) multi-agent application implemented in COOL

Agent: Autonomous entity that uses structured coordination protocols (conversation classes) and interacts with other agents through KQML messages. Agents are associated to a conversation manager and run in an agent execution environment.

Agent Execution Environment: A software environment residing at a site in the network where a set of agent is executed by means of a conversation manager

Conversation: A runnable instance of a conversation class attached to an agent which maintains the current state, a local knowledge base and historical information.

Conversation Class: A state-based plan describing what an agent does in certain situations. Conversation classes spawn a finite graph where the nodes (states) are linked through conversation rules.

Conversation Rule: Describe a state transitions in a conversation class. Composed of a condition and an action part. The conditions may be received messages of a particular structure or certain predicates applicable to the current local knowledge. Actions include transmitting messages to other agents, creating or manipulating other conversations or changes of the local knowledge.

Conversation Manager: A control instance responsible to execute a set of agents and to trace their behavior. Conversation managers and the set of agents they control are running in an agent execution environment.

(Graphical) Presentation System: Platform that supports the development of graphical components, e.g. Java, X widgets, Windows.

Graphical User Interface (GUI): Graphical component that provides access to and reflects the functionality of GenUA to the human

Generic User Agent (GenUA): Mediator between scenarios and graphical user interfaces

Input Request: A request from a conversation to the user to provide input and/or to make a decision. Encoded using the pattern grammar. Automatically detected by GenUA and sent to the GUI.

Scenario: Corresponds to the term application

User: Human who interacts with scenarios through a graphical interface by means of GenUA

User Interaction: Encompasses both interactions with GenUA itself and interactions with a particular scenario

User Message: Any kind of notification or execution result from conversations to users. Automatically detected by GenUA and sent to the GUI.

Appendix B

COOL Syntax

This section describes the syntactical structure of the main COOL objects, used in defining the supply chain demonstrator.

B.1 Agents

The syntax of the agent definitions in COOL:

```
<agent-definition>::=  
(def-agent <name>  
  :conversation-classes <list of conversation classes>  
  :conversations <list of conversations>  
  :continuation-rules <list of continuation-rules>  
  :continuation-rules-incomplete <T or nil>  
  :continuation-control <name of control fn>[default: agent-control]  
)
```

Explanation of the non-terminals:

<name> is the global name of the defined agent in the COOL environment. Different agents must have different names.

<list of conversation classes> is the list of names of the *conversation classes* (see own section below) this agent has. In order for an agent to use a conversation class, it must have been declared in this way. The conversation classes must have been declared before use. Alternatively, to avoid problems of forward referencing, COOL provides a def-associations construct allowing one to declare conversation classes for an agent at the end of the program, after the agent and the conversation classes have been declared.

- <list of conversations>** list of actual conversations of an agent. Must have been defined before use, but to avoid referencing problems the def-associations construct can be used.
- <list of continuation-rules>** list of continuation rules for this agent. Same observations as for conversation classes wrt. prior declaration and type use of def-associations.
- <T or nil>** flag specifying whether the set of continuation rules of this agent is complete or not. Note that this flag marks the entire set as incomplete, so even if every individual rule is complete the knowledge acquisition interface for continuation rules will still be popped up (under the assumption that new rules may be added or existing ones deleted, see section 4.3).
- <name of control fn>** [default: agent-control] This is the name of the pluggable function used as the interpreter of continuation rules. It can be defaulted to a standard interpreter like agent-control, provided with the language implementation.

B.2 Conversation Managers

Declaration of a conversation manager having the unique global name **<name>**:

```
(def-conversation-manager <name>
  :agents <list of agents>
  :trace-agent <list of traced agents>
  :trace-message <list of traced messages>
  :trace-conv <list of traced conversations>
  :trace-conv-rule <list of traced conversation rules>
  :trace-err-rule <list of traced error recovery rules>
  :trace-cont-rule <list of traced continuation rules>
  :trace-conv-class <list of traced conversation classes>
)
```

The other arguments describe various tracing options. These options belong more to the particular implementation, but for orientation should always include message tracing which is of major interest.

B.3 Conversation Classes

The COOL syntax for defining a conversation class:

```
(def-conversation-class <name>
  :content-language <language name>
)
```

```

:speech-act-language <language name>
:ontology <ontology name>
:rules <list of conversation rules>
:rules-incomplete <T or nil>
:control <conversation rule control fn>[default
  interactive-choice-control]
:initial-state <state name>
:final states <list of state names>
:variables <list of variables>
:interactive <T or nil flag>
:recovery-rules <list of recovery rule names>
:recovery-rules-incomplete <T or nil flag>
:recovery-control <recovery rules control fn>[default
  recovery-control]
:intent-check <intent check fn>
)

```

Explanation of the non-terminals:

<name> is the unique name of the conversation class

<language name> is the name of the content language or the speech-act-language. The speech-act-language is usually KQML. The content language is at the discretion of the user. If the :content-language is specified, the system will automatically insert it in the KQML messages that are received or sent by the conversation.

<ontology name> is the ontology used by the conversation. If given, it will be automatically inserted into messages received or sent by any conversation described by this class. This simplifies writing the message patterns for received messages and the output messages.

<list of conversation rules> list of conversation rule unique names for this conversation class. Rules are given associated with the state for which they apply. To avoid problems of forward referencing def-associations can be used.

<T or nil> value of a flag that specifies the incomplete status of a rule. Value T means the rule is incomplete, nil means the rule is complete.

<conversation rule control fn> name of a function used as the interpreter for conversation rules. This is a pluggable interpreter and is defaulted by the system. The default interpreters should be accessible as the value of some parameter that users can set at will.

- <state name>** for the initial state, this is its name. This is the state a conversation starts in.
- <list of state names>** list of states. For the final states, when the conversation reaches any of these states it will terminate.
- <variables>** list of variables of this conversation class. It is not necessary to declare variables in advance, they can be created dynamically, when and if needed. This declaration is useful especially to document the purpose and use of variables in a conversation class. Remember that variable names start with '?' or '??'.
- <interactive>** defines whether this conversation class allows interactive execution by a web user by means of the GenUA interface. A conversation class should be marked as interactive if it includes any *conversation rules* which provide a received-pattern slot. (see 5.2.2 ff.)
- <recovery rules>** list of unique names of the recovery rules used in the conversation.
- <recovery rules control fn>** name of pluggable interpreter of recovery rules. Defaulted by various system configurations.
- <intent check>** the predicate that will be applied to the value of the intent slot of an incoming message to determine if this class can handle the incoming message. Can be function name, lambda expression or something else (depending on what the host language allows).

B.4 Conversation Rules

The COOL syntax for defining conversation rules:

```
(def-conversation-rule <name>
  :name <name>
  :comment <comment>
  :current-state <state>
  :received-pattern <received-pattern>
  :received-test <received-test>
  :received <received>
  :received-many <received-many>
  :received-queue-test <received-queue-test>
  :waits-for-test <waits-for-test>
  :such-that <such-that>
  :next-state <next-state>
```

```

:transmit <KQML message>
:wait-for <wait-for>
:do-before <do-before>
:do-after <do-after>
:do <do>
:interactive-execution-fn <interactive-execution-fn>
:incomplete <T or nil>
)

```

Explanation of the non-terminals:

<name> is the unique global name of the rule.

<comment> is any string used to document the rule.

<state> is a state name.

<received-pattern> is used for interactions of a web user with the COOL environment by means of the GenUA interface only. It consists of an expression which builds a list structure according to the pattern grammar (see 5.2.4). This allows a complete specification of the message structure and its content being expected from the web user. The message structure given in a received-pattern slot of a particular conversation rule should always match one of the following “received” tests. Moreover, web user interaction implies disabling the “incomplete” slot as the standard pop up mechanism is not available on the web (see 5.2.2).

<received-test> is a predicate of one argument, the received message. Used to perform procedural checks on the received message.

<received> is a message pattern against which the actual message will be checked. The use of pattern-matching enables the user to specify declaratively the expected structure of a message in order to apply a rule.

<received-many> list of patterns against corresponding received messages will be matched. This enables us to match several messages in a rule.

<received-queue-test> predicate of one argument, the queue or messages for the conversation. Enables checking the entire queue before applying the rule.

<waits-for-test> predicate of one argument, the list of terminated conversations this conversation is waiting for. This can be used only if the current conversation is waiting for other conversations to terminate, enabling the conversation to be resumed as soon as the test condition is satisfied.

<such-that> predicate of any number of arguments applied on a list of bindings produced by matching the patterns in :received or :received-many. A number of standard variables bound by the system are available.

<next-state> state name, the next state if the rule is applied.

<KQML message> a KQML message to be transmitted as an effect of rule execution. Anywhere in the KQML message, values of the form (<expr>) are replaced by the value of <expr>. Inside <expr>, free variables are first replaced by their values. This gives a way of performing arbitrary computations to determine any components of the message to be sent.

<wait-for> list of conversations this conversation is waiting for to terminate. The conversation is put on wait as a consequence of executing the rule.

<do-before> executable actions to be carried out before transmitting the <KQML message>

<do-after> executable actions to be carried out after transmitting the <KQML message>

<do> executable actions to be carried out at an unspecified moment in relation to when the <KQML message> is transmitted

<interactive-execution-fn> a user written function that has application-specific GUI-s guiding the execution of the action part of the rule.

The semantics of the conversation rule definition above is as follows:

If

current state of the conversation is :current-state
 and :received-test is satisfied by the last message
 and last received message matches :received
 and there exists a set of messages in the queue matching
 :received-many (order matters)
 and the message queue satisfies :received-queue-test
 and :wait-for-test predicate satisfied by the list of
 conversations this conversation is waiting for
 and :such-that predicate satisfied

Then

go to :next-state
 and transmit the :transmit message
 and put the conversation on wait for the mentioned :wait-for
 conversations
 and do the :do-before actions before transmitting the message

and do the :do-after actions after transmitting the message
and do the :do actions anytime
or if interactive-execution-fn exists, execute it
(assume it will carry out all above actions)

The semantic of a conversation rule is not influenced by the way, a user interacts with the system (see 5.2.3).

The use of predicates and other executable functions in places like :such-that and :do clauses assumes that programmers are allowed to place as free variables any variables of the conversation. These will be replaced with their actual values before evaluating the forms they are in. Moreover, the following variables are always bound by the system and can be used as well:

- ?convn - the name of the current conversation
- ?agent - the agent who owns the current conversation
- ?message - the :received message
- ?conv - the current conversation (as object).

Appendix C

User Guide for the Supply Chain Demonstrator

The COOL User Interface is available from the following URL

<http://timmins.ie.utoronto.ca:8800/Welcome.html>

There you click on "COOL User Interface in action".

Start the Generic User Agent When entering the page "COOL User Interface", you'll probably first need to start the Generic User Agent by clicking on the corresponding button. After that, the page will be reloaded and the System Login Applet appears.

Login to the system There you have to specify your login name and your password. Note that if you are a non-registered user, login as "anonymous" and your E-mail address as identifier. Furthermore, you'll need to specify the complete hostname from where you've started the browser. The hostname may be specified as a *complete* symbolic IP address ("timmins.ie.utoronto.ca") or as an IP number ("123.45.67.89").

Creating the application After submitting your login information, the actual COOL User Interface will appear on your screen. The session begins by selecting the application "Supply-Chain+Scheduling" in the upper left choice box and pressing the "Create Application" button.

Linking the application When it has been indicated that the application was successfully created (which may take some time, be patient!), you select the same application from the choice box beside and press "Link Application" in order to register as a user. Successful execution results in filling the list of available agents below.

Initiating the conversation Now you click on the agent "Customer" and retrieve its available conversation classes by pressing the button to the right of the agent list. This results in filling the corresponding list.

Clicking on the "Customer Conversation" pops up a window, where you the class is displayed. When you press "Initiate Conversation" at the bottom of the window, an instance of the plan will be created, indicated by a small notification frame and after that the new instance will appear in the list of "Ongoing Conversations" in the middle of the main window. You may in principle start as many instances as you like, but remember each new instance requires to conduct the entire process upon satisfaction, which may take more time.

Answering requests After some moments, you will recognize a first request from the application displayed in the list "Requests from Agents" right below the "Ongoing Conversations". Clicking on the request pops up a frame, where, in generally, you can choose from a number of possible actions (for the initial request, you will only have one choice).

When selecting a possible action, a top level modal dialog appears which besides the top request elements shows a legend for the graphical elements. It is recommended to make your way through the input specification from top to the bottom, by clicking every green button and inspecting and/or modifying every "inner" value. However you can also leave every inner dialog by pressing "Cancel" and return to it later again.

When composing a form, you should enter values that match the description beside the field. Otherwise you will get a notification about incorrect fields. It is not possible to leave a dialog without correct values! If you get lost or if you wanna restore the original values, press "Clear" and you may leave the dialog. Pressing "OK" (assuming that all the fields are correct) leaves the form dialog and sets the value of the attribute where you started the form dialog from to what has been specified now. You may re-enter and modify this dialog as often as you like.

When composing a list, you'll see always the list of elements specified so far in a list. Default values are automatically inserted. Optional values, if any, can be inserted directly from the green choice box above the list. Below the list you'll find list manipulation functions. "Modify Element" and "Delete Element" requires that you first select the element out of the list and then apply the desired function. "Clear List" removes all elements. "Cancel" leaves the dialog without effects to the parent dialog.

Adding and modifying elements works depending on the complexity of the element. For simple elements such as strings, integers etc., you can enter the element directly in the line above the list of elements and add it by pressing the button. To modify it, select it in the list, press "Modify element" upon which the element disappears from the list and is displayed. Pressing "Accept Element" inserts the (modified) element again in

the list. For complex structures, there is no line to edit. Instead you just press "Add Element" which pops up the structure of the element. Here you will find either another list or a form with the same way of interaction.

Some lists may require a specific number of elements while others accept any number. The former case leads to the effect that you cannot press "End of List" unless the list has the required number. Once this is done, "Add Element" is disabled and you may leave the dialog, and the value of the attribute you've started the list dialog from is set to what you've specified. On the contrary, lists with arbitrary element number can be left at any time. Analogous to form dialogs you may re-enter, add, delete, modify elements as often as you like.

Once you have made your way through the nested dialog structure, you'll end up in the top dialog, where you may "Browse" what you've specified or "Submit" the completed request to the agent. However, you may also leave the top dialog and the selection frame without doing anything.

Note that some requests may be optional indicated by a corresponding attribute on top level. You may but you don't need to fill and submit them. *ALL* other requests must be answered to cause an execution progress.

Effects of submitting a completed request When a completed request has been received, it leads to updates on your screen: answered requests disappear, ongoing conversation change, notifications from the application and/or new input requests arrive. Be patient! This is not meant to be a high speed system.

All notifications or (intermediate) results received from the application appear in the list "Messages from Agents" at the bottom of the main window. By clicking on them, you may inspect the content in a separated frame at any point in time. You may decide whether you wanna keep or delete the message. Note that this decision is final. There is no retrieval!

During the interaction process The interaction process is mostly self explanatory and transparent. However, if you have difficulties to understand the context, take a look at the description of the Supply Chain Demonstrator in chapter 8. Other than that you may feel free to play around with different decisions and values.

At any time, you may download the history including all interactions including requests and responses by pressing "Get History" on top of the main window. Once again, be patient, there might be lots of information to transfer.

You may also inspect how the initiated conversation changes through the execution process by clicking on it in the list of "Ongoing Conversation".

Unlinking the application Once you have gone through the complete customer order process, meaning you've got a bill and delivery confirmation and you wanna leave, press "Unlink Application" on top of the main window.

Choosing "Keep state" will have the effect that you can inspect your current history at a later session again. If you do so, and you'll have to provide a redirection address, which will have no effects at all, as there is no time delay during the execution, hence no offline messages are possible. After that you may close the main window.

However, normally you'll choose "Finalize Interaction" when unlinking.

Shutdown the application With respect to other interested users and if you don't wanna resume your session necessarily, it is better to shutdown the application completely. This is done by pressing "Destroy Application" on top of the main window. After that you may close the main window.

Terminating the Generic User Agent As the continuous listening of GenUA for incoming requests is rather time-consuming, it should not run idly for ages. We recommend to stop it by pressing the corresponding button on the HTML page, which should be still available to you.

Troubleshooting In some cases, after answering a request, nothing seems to happen at all. You'll need to press "Manage Conversations". This forces the COOL protocols to move on. Do it only once at a time and see what's happening.

When errors occur during manage conversations (small red alarm window), you have no other chance than to unlink from the application, to destroy it and to start it all over again.

Other errors (small yellow window) should not occur, if so, retry the action you just have made.

There is an interaction blocking and time-out mechanism from the main window to the Generic User Agent. Time-out is after approximately 20-30 seconds upon which the blocking mechanism is stopped. However, this indicates network communication problems or blocking of the Generic User Agent. There is no other chance than to close the main window (as an X widget or Windows frame), to stop the agent and to start from the beginning.