International Journal of Cooperative Information Systems
Vol. 5, Nos. 2 & 3 (1996) 275–314

CAPTURING AND MODELING COORDINATION KNOWLEDGE FOR MULTI-AGENT SYSTEMS

MIHAI BARBUCEANU and MARK S. FOX
Enterprise Integration Laboratory, University of Toronto
4, Taddle Creek Road, Rosebrugh Building, Toronto, Ontario, M5S 1A4
{mihai,msf}@ie.utoronto.ca

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. A major challenge in building such systems is coordinating the behavior of the individual agents to achieve the individual and shared goals of the participants. As part of a larger project targeted at developing an Agent Building Shell for multiagent applications, we have designed and implemented a coordination language aimed at explicitly representing, applying and capturing coordination knowledge for multiagent systems. The language provides KQML-based communication, an agent definition and execution environment, support for modeling interactions as multiple structured conversations among agents, rule-based approaches to conversation selection and execution, as well as an interactive tool for in context acquisition and debugging of cooperation knowledge. The paper presents these components in detail and then shows how the coordination language is used in the Agent Building Shell to manage content-based information distribution scenarios among agents and the coordination aspects of conflict management processes that occur when agents encounter inconsistencies. The major application of the system is the construction and integration of multiagent supply chain systems for manufacturing enterprises. This application is used throughout the paper to illustrate the introduced concepts and language constructs.

1. Introduction

The agent view provides a level of abstraction at which we construe computational systems that interoperate globally across networks linking people, organizations and machines on a single virtual platform. We call the computational entities that can operate at this level *agents*. For our purposes, we consider an agent to be a piece of software that:

(1) Is significantly autonomous, goal-oriented and entrusted in performing its functions.
(2) Operates globally on networks by relying on application-independent high-level communication and interaction protocols with other "agents".

Focusing on the agent level of system (de)composition brings to attention a number of specific issues that are not adequately dealt with at other levels of system organization. Some of these are:

- *Agent interaction:* How do agents communicate? How do agents coordinate in joint work, such as to achieve the individual and joint goals of the participants? How are problems stemming from dynamically occuring events and partial knowledge about the environment handled during coordinated behavior? How do we model the patterns of interaction and interoperation that characterize coordinated behavior? How do capture these patterns during the on-line operation of the system?
- *Representation:* How do agents represent their local views of the domain? How is the local view updated or maintained as a consequence of interaction? How are the semantic problems related to conflicting or different meanings of the exchanged terms solved? How do agents revise their beliefs due to exchanged information? How do agents share models and how does the shared model change? How do agents model each other in a cooperative community? How are commonsense issues, e.g. time, action, causality, handled?
- *Reasoning:* How do the requirements for communication and coordination impact the internal reasoning of agents? How do agents handle contradictory information, and how is consistency maintained across agents that may have different goals, views, preferences?
- *Integration:* How can pre-existing (legacy) applications be integrated into agents and thus used in agent communities?

From the practical standpoint, any solutions to the above issues must provide the ability to *reuse* descriptions of coordination mechanisms, system components, services and knowledge bases. Based on this recognition, we are developing an Agent Building Shell that provides reusable languages and services for agent construction, relieving developers from the effort of building agent systems from scratch and guaranteeing that essential interoperation, communication and cooperation services will always be there to support applications.

The layered agent architecture of the Agent Building Shell is shown in Fig. 1. The *knowledge management* layer provides support for general purpose representation and inference. It is used to represent an agent's conceptualizations — of its domain, operation environment and of its own capabilities — as well as the agent's actual beliefs — again about domain, environment and self. It provides support for nonmonotonic reasoning — agents change their beliefs dynamically — and general purpose deductive reasoning. The description logic implementation of this layer provides services for automated concept classification and inconsistency detection, theorem proving through subsumption and truth-maintenance based management of the belief base.

The *ontology layer* consists of the actual conceptualizations agents maintain about their domain, environment and self. Some conceptualizations are shared amongst agents to allow them to communicate in terms that are semantically unified. The environment and self representations use a shared organization ontology that captures the structure of organizations, the roles, goals, actions and empowerment of member agents of the organization.

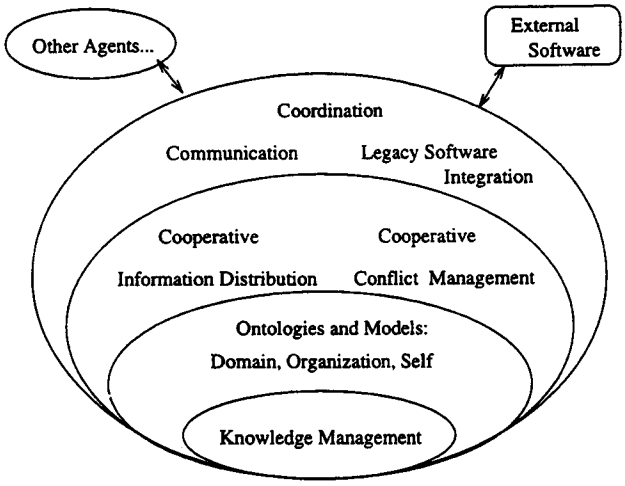

Fig. 1. Architecture of the agent building shell.

The *cooperative information distribution services* provide permanently active information distribution services allowing agents to stay informed about significant events without having to explicitly demand other agents to provide this information each and every time they need it. Agents advertise their long-term topics of interest to the community. Agents that can supply relevant information will do so whenever the information is available and as long as the interest persists. If previously sent information is later invalidated, senders will notify receivers. The service uses subsumption in description logics to "prove" that some information matches an interest expressed as a description logic concept.

The *cooperative conflict management service* provides a general model for reasoning about retraction in a multiagent setting. If an agent receives contradictory information from other agents, it applies this model to retract some beliefs and reinstall consistency both locally and with its neighbors. The model uses:

(1) A measure of the credibility of the beliefs that are potentially retractable.
(2) A measure of the utility of these beliefs in the global current situation.

The latter captures the fact that retracting information may imply undoing previous decisions and actions (e.g. retracting orders for materials) that come with money or other costs. The model reinstates agent-level consistency and, through negotiation with the agents sharing the retracted beliefs, extends the consistent state to surrounding agents. Also, the model stipulates the kind of cooperative behavior that an agent must exhibit towards other agents in the process of reestablishing a consistent state. Both cooperative information distribution and cooperative conflict management require coordinated behavior from the involved agents and thus make essential use of the coordination system.

In this paper we focus on the solutions we are providing for the outer layer of the architecture. They are embedded into a *domain independent COOrdination Language* (COOL) that provides services for defining distributed agent configurations, managing communication, defining and managing structured interactions amongst agents, external software integration and in context acquisition and debugging of coordination knowledge. As these solutions impact on the way agents manage change by information distribution and conflict resolution, we also address these aspects showing how the coordination service supports these tasks.

The paper is structured as follows. In Sec. 2, we review the work in Distributed Artificial Intelligence from several perspectives and define our research goals. As the subsequent presentation of our tools is carried out in the context of our main application, the agent-based integration of the supply chain of manufacturing enterprises, we continue in Sec. 3 with presenting this application domain. Section 4 deals with the main subject of the paper, the components of the coordination language. We illustrate the language throughout with examples from the supply chain. Section 5 then deals with the coordination knowledge acquisition service that allows users to extend and debug coordination knowledge on-line. To show how the coordination system is integrated with other reasoning tasks in the Agent Building Shell, in Sec. 6 we review two other services of the architecture that make use of the coordination framework, cooperative information distribution and cooperative conflict management. In the end, we discuss some related approaches and provide concluding remarks.

2. Coordination Knowledge

Coordination has been defined as the process of *managing dependencies between activities*.[39] An agent that operates in an environment holds some beliefs about the environment and can use a number of actions to affect the environment. Coordination problems arise when:

(1) There are *alternative actions* the agent can choose from, each choice affecting the environment and the agent and resulting in different states of affairs and/or;
(2) The *order and time of executing actions* affects the environment and the agent, resulting in different states of affairs.

The coordination problem is made more difficult as an agent has incomplete knowledge of the environment and of the consequences of its actions and the environment changes dynamically making it more difficult for the agent to evaluate the current situation and the possible outcomes of its actions. In a multi-agent system, the environment is populated by other agents, each pursuing their own goals and each endowed with their own capabilities for action. In this case, the actions performed by one agent constrain and are constrained by the actions of other agents. To achieve their goals, agents will have to manage these constraints by coordination.

In this paper we adopt the view that the coordination problem can be tackled by having knowledge about the interaction processes taking place among agents.

This knowledge is about the problem-solving competence of multi-agent systems as opposed to that of individual agents. As such, Fox[20] has proposed that it be studied as an "organization level" and applied Organization Theory concepts to characterize this level. More recently, Jennings[30] has coined the term "cooperation knowledge level" to separate the social interaction know-how of agents from their individual problem-solving know-how and to help focus efforts on coming with principles, theories and tools for dealing with social interactions for problem solving.

Previous work in DAI can be seen as investigating various facets of this level of knowledge. One direction is concerned with devising useful structures for cooperative problem solving. Thus, the Contract Net protocol[53] provided a way of coordinating agents without global control, by means of a contracting model comprising dynamic task decomposition, negotiation of subtask assignments among agents and the commitment of agents to their assigned subtasks. In the Partial Global Planning method (PGP)[18] and its Generalized PGP form,[16] agents maintain their own subjective views of the tasks, task dependencies and the responsibilities of agents. Various coordination mechanisms (like exchanging private views of tasks, communicating results, handling various types coordination relationships) enable agents to modify their subjective view of the task structure and their commitments to tasks in the task structure, ultimately improving performance. The Joint Responsibility model[32] prescribes when and how agents should form teams and how team members should behave during joint action. The code of conduct imposed by Joint Responibility ensures that the group will operate in a coordinated and efficient manner and that it is robust in face of changing circumstances.

Given the diversity of such cooperation structures, how can we identify, analyze and formalize the essential elements cooperation structures are composed of? This is the focus of a second major direction of work in DAI. We make several distinctions here. The first is between what happens inside an agent when it coordinates with other agents and what happens between agents when cooperative behavior occurs. The second is between explaining how human agents behave and how programmed agents behave. Although in this paper we are solely concerned with artificial agents, insights into human agenthood will help us build agents that are understandable and thus easier to integrate as partners for human users.

Talking about what happens inside human agents, many researchers believe that mental states, like intentions and commitments are the central notion here. Intentions and commitments have been studied for example in Refs. 8, 12 and 49. These studies uncovered a number of essential properties of intentions. Intentions must be consistent with each other and with the beliefs of the agent, the latter meaning that if the intended actions are executed and the agent's beliefs hold in the world, then the desired state of affairs should follow. Also, intentions should have a degree of stability, however without being totally inflexible. Agents should not spend all their time considering and reconsidering intentions. At the same time, they should be able to drop intentions if changes in the situation makes it impossible or undesirable to achieve the intended state of affairs. The re-examination of agents'

intentions should be "regulated" by known policies or conventions[31] stating under what circumstances intentions should be reconsiderd. In the Cohen and Levesque[12] model for example, an agent should reconsider its commitment to a goal G if any of the following happens: G is already satisfied, G will never be satisfied, the motivation for G does not exist any more.

The above approach has been extended to the modeling of inter-agent phenomena. Levesque, Cohen and Nunez[37] have proposed for example necessary and sufficient conditions for having Joint Persistent Goals that would allow agents to form teams:

(1) Agents mutually believe G is currently not true.
(2) They mutually believe they all want G to become true.
(3) Until they all come to mutually believe either that G is true, that G will never be true or that the motivation for G is false, they will continue to mutually believe that they each have G as a weak achievement goal (roughly either a normal goal, or a goal whose achievement status has to be mutually believed by all team members).

The last condition allows agents to undertake actions knowing that if a problem with goal satisfaction occurs, the agents detecting it will inform the others. In order to act cooperatively, a number of other conditions have been discussed, including the mutual desire of agents to cooperate[13] (otherwise agents may for example compete) and the need for a common plan to achieve the goal that will determine the contributions of participants (otherwise inconsistent action may result even if there is a common goal). The latter issue has been dealt with by distributed or multi-agent planning research, including for example, Refs. 17 nd 25. Monitoring the execution of joint action has been investigated as a way of determining what to do when things go wrong or unexpectedly.[33] Another approach to coordinating multiple agents is to restrict their activities in a way that enables them to achieve their goals without interfering with each other. Shoham and Tennenholtz[52] have proposed social laws as the means to specify these restrictions and have studied how such laws can be designed to guarantee certain behaviors from the multi-agent system.

From a sociological perspective, Castelfranchi[10] has shown that internal commitments of agents (commitments of individual agents to certain actions) are not enough to explain social phenomena. He discusses social commitments as basic relations between two or more agents with respect to executing some actions. This is different from having several agents sharing the same internal commitment. This notion uncovers the dependence and power relations among people that form the objective basis of social interaction and has important normative consequences, like obligations and expectations, that pertain to the notions of Group and Organization.

Given work like the above, how do we brigde the gap between the logical, sociological and psychological analysis and the engineering of practical multi-agent systems, performing in real environments and bringing real services to people? There

are not too many answers to this question, but a few of them deserve mentioning. A first answer is represented by the applicative work of Jennings[33] who started with the Cohen and Levesque model for joint intentions, extended it to better fit the need for a common plan and then implemented it with state of the art AI technologies. The result was an industrially applied multi-agent system that comprised the results of theoretical work on joint intentions.

The second answer lies in developing generic agent architectures that integrate the results of theoretical investigations into practical languages and tools. This is the path taken by Agent Oriented Programming[51] where a generic notion of agent was proposed, using speech-act based communication, rule-based behavior and encapsulation into object-like structures. This approach talks about an agentification process in which real systems are casted in terms of mental states and the other concepts provided by the approach. Other work in the same direction focuses on specific aspects that are perceived as important when developing practical systems. The ARPA sponsored Knowledge Sharing Effort[44] attempts to build technologies for inter-agent communication by proposing a language for content communication based on logic, KIF,[23] and a language for intention communication, based on communication acts, KQML.[19] Together, these form an Agent Communication Language (ACL), and approaches like Genesereth's define an agent as anything that communicates using the ACL.[24] Also part of the Knowledge Sharing Effort, work has been devoted to the problem of semantically unifying agent communication by giving common definitions to the terms used by agents. Dictionaries of such shared terms are called ontologies[21,28,29] and a number of tools have been constructed for building and maintaining them. Our work on the generic agent shell falls into this broad second category.

As far as our approach to coordination is concerned, we take the above investigations as revealing the nature of the knowledge that is involved in social behavior and interactions. Our aim is to provide generic tools for the capture, representation and use of this knowledge in multi-agent systems. As previously noted by Jennings,[33] the evolution of applicative DAI systems follows the evolution of applicative knowledge based AI systems in the following sense. Initially, knowledge based systems were encoded in more or less *ad hoc* ways, such that a lot of relevant knowledge about, for example the task structure and problem solving methods were buried into the code once systems were implemented, hence could not be explicitly analyzed and reasoned about. This created growing problems with explanation, reusability and maintainability. In response to these problems, emphasis has later shifted onto explicitly characterizing the problem solving task at a higher level, for example in terms of generic problem solving methods[42] like heuristic classification[11] or distinguishing between the various types of knowledge used to model the domain, the inferences, the task structures and the higher order strategies for resolving impasses.[55] With this emphasis came a new generation of tools that are now able to explicitly represent such higher level types of knowledge and assist users in building systems in more principled and accountable ways.

In an essential way we are trying to do the same for the coordination knowledge agents must possess to interact successfully. In other words we are trying to come up with higher level constructs for describing coordination processes and to fully support these constructs in a programming environment for building multi-agent systems. The insights into the nature of social interaction, from sociological or psychological sources, described semantically in logic systems, give us principles and background knowledge for understanding and modeling interactions. Together with domain and application knowledge, they are used by developers to design the coordination structures that would be actually used by applications. These coordination structures, encoded into our coordination language, then guide the interactions among agents. Even if structures of human social interaction may be a source of inspiration for some agent coordination structures, note that *they are not our object of study and we do not aim in any way at building programs that behave similarly.* Our goal is to build clear, understandable, reusable models of interaction for artificial multi-agent systems and to support their engineering as far as we can.

In this perspective, we are developing coordination technology that:

(1) Provides a conceptualization of the coordination task in terms of agents, generic and actual conversation structures, rule-driven conversation moves, rule-driven exception handling, rule-driven control of agent and conversation execution, multi-agent and multi-conversation management.
(2) Actual programming constructs for the above concepts.
(3) A full visual environment for developing, testing and executing coordination programs.
(4) A full visual environment for non-intrusively capturing coordination knowledge in the execution context. This technology is developed in the framework of a multi-agent system for supply chain integration which provides our experimentation environment. All these elements form the subject of this paper.

3. Integrating the Supply Chain

The supply chain of a modern enterprise is a world-wide network of suppliers, factories, warehouses, distribution centres and retailers through which raw materials are acquired, transformed into products, delivered to customers, serviced and enhanced. In order to operate efficiently, 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, these events cannot be dealt with locally, i.e. within the scope of a single supply chain "agent", requiring several agents to coordinate in order to revise plans, schedules or decisions. In the manufacturing domain, the agility with which the supply chain is managed at the tactical and operational levels in order to enable timely dissemination of information, accurate coordination of decisions and management of actions among people

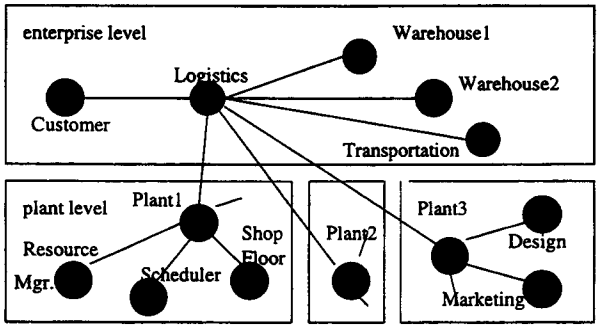

Fig. 2. Multi-level supply chain.

and systems, is what ultimately determines the efficient achievement of enterprise goals and the viability of the enterprise on the world market.

We address these coordination problems by organizing the supply chain as a network of cooperating agents, each performing one or more supply chain functions, and each coordinating their actions with other agents. Figure 2 shows a multi-level supply chain. At the enterprise level, the Logistics agent interacts with the Customer about an order. To achieve the Customer's order, Logistics has to decompose it into activities (including for example manufacturing, assembly, transportation, etc.). Then, it will negotiate with the available plants, suppliers and transportation agents the execution of these activities. If an execution plan is agreed on, the selected participants will commit themselves to carry out their part. If some agents fail to satisfy their commitment, Logistics will try to find a replacement agent or to negotiate a different contract with the Customer. At the plant level, a selected plant will similarly plan its activities including purchasing materials, using existing inventory, scheduling machines on the shop floor, etc. Unexpected events and breakdowns are dealt with through negotiation with plant level agents or, when no solution can be found, submitted to the enterprise level.

A major challenge for such an application is the ubiquity and complexity of coordination aspects. Coordination occurs when agents negotiate future plans or actions, when they execute actions together, when they exchange information, when they respond to events or when they solve conflicts. We address this by building models and tools for the representation, utilization and in context acquisition of this sort of knowledge, as explained in the remainder of this paper.

4. The Coordination Language

As in the situated-automata model,[47] we view an agent as essentially performing a *transduction*. It takes a stream of input messages from the environment (in general composed of other agents) and generates a stream of output messages to the environment, mediated by its internal state. The mediation is described in the coordination language and performed by a conversation management mechanism

enhanced with knowledge acquisition capabilities whose description is the purpose of this and the next section.

Before going into the details of the coordination language, we note that the interaction among agents takes place at several levels. The first level is concerned with the *information content* communicated among agents. A piece of information communicated at this level may be a proposition (fact) like "(produce 200 widgets)". The ARPA Knowledge Sharing Effort[44] has produced the KIF[23] logic language for describing the information content transmitted and the conceptual vocabularies communicating agents must share in order to understand each other.

The second level specifies the *intentions* of agents. 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.

KQML[19] has been designed as a universal language for expressing such intentions such that all agents would interpret them identically. KQML supports communication through explicit linguistic actions, called *performatives*. As such, KQML relies on the speech act[48] framework developed by philosophers and linguists to account for human communication. Work is currently being done[15,36] on endowing KQML with formal semantics based on the speech-act theory as formalized and extended within the fields of Computational Linguistics and Artificial Intelligence.[14]

The third level is concerned with the *conventions* that agents follow when interacting by exchanging messages. The existence of shared conventions makes it possible for agents to coordinate in complex ways, for example by carrying out negotiations[54,56] about their goals and actions. Many such examples can be given in the context of the supply chain. The Customer agent acquires from the customer an order for 200 lamps with a due date for 28 September 1994. It sends this as a *proposal* to the Logistics agent. Knowing that Logistics can only answer with *accepting*, *rejecting* or *counter-proposing*, the Customer agent is able to check that the actual response is one of these and carry out a corrective dialogue with Logistics if this is not the case or if other events occur (such as delays or message shuffling). If Logistics answers with a counter-proposal (e.g. 200 lamps with due date 15 October 1994), the Customer agent may use knowledge about acceptable trade-offs and negotiate with Logistics an amount and a due-date that can be achieved and satisfies the customer. In its turn, upon receiving the order proposal, Logistics will start negotiations with plants and transportation companies to determine the feasibility of scheduling the production and delivery of the order and then will monitor

execution as already shown. This is the level of interaction we are supporting with the COOL language described in this paper.

4.1. *Functions and architecture*

From the viewpoint of the intended functionality, COOL is:

(1) A language for describing the coordination level conventions used by cooperating agents.
(2) A framework for carrying out coordinated activities in multiagent 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.

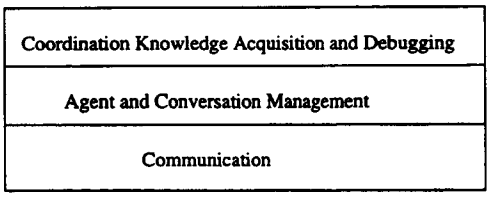

Fig. 3. COOL Architecture.

Architecturally, (Fig. 3) COOL is structured into three layers, the first providing support for communication in KQML, the second providing the main machinery for defining and executing agents and coordination structures and the third supporting the in context acquisition and debugging of coordination knowledge.

4.2. *Communication*

COOL has a communication component that implements an extended version of the KQML language. Essentially, we keep the KQML format for messages, but we leave freedom to developers with respect to the allowed vocabulary of communicative action types (called performatives in KQML, but see Ref. 15). Also, we do not impose any content language. We have implemented a mail system for KQML

```
(propose                                        ;; new performative
  :language KIF
  :sender A
  :receiver B
  :content (produce (or widget gadget) 299)
  :conversation C1                              ;; first new slot
  :intent (explore fabrication possibility)) ;; second new slot.
```

Fig. 4. Message example.

messages providing TCP/IP supported transport and mail services like persistent storage of received KQML messages, visual tools for message browsing, composition, sorting and general pattern based retrieval. The example in Fig. 4 illustrates the form of extended KQML we are working with.

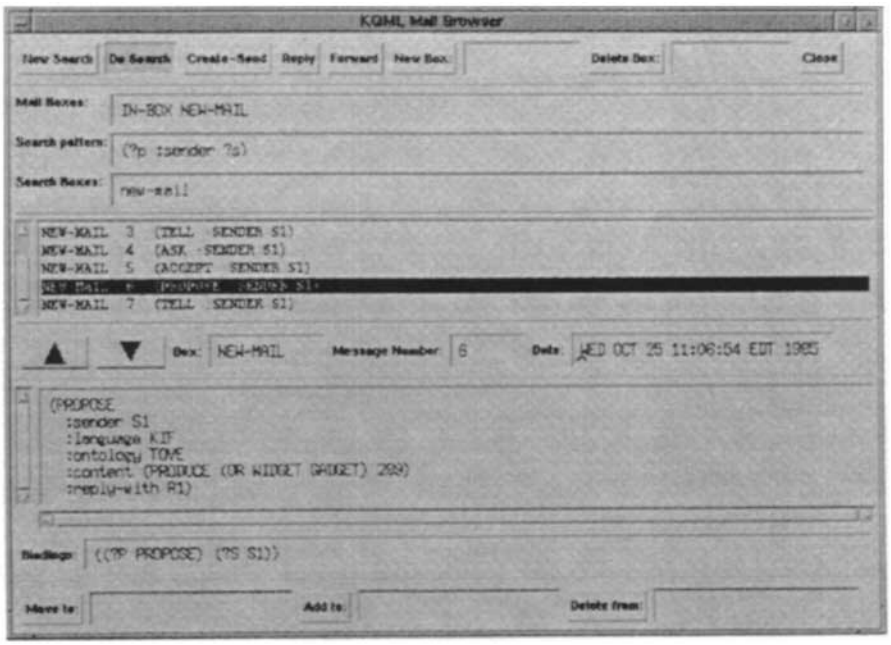

Fig. 5. KQMaiL system interface.

Figure 5 shows the screen for the interface to our KQML mail system (called KQMaiL). Note the use of mail folders and the use of pattern based search to access and visualize the contents of folders. Composing and sending out KQML messages is similarly supported.

4.3. *Agent and conversation management*

The purpose of this layer is to describe and execute coordination protocols, i.e. shared conventions about agent interaction. The main idea here is that agents interact by carrying out structured conversations. To embody this model we provide explicit language constructs for defining the agents we deal with, the structure of the conversations they carry out, the rules that describe conversation progression through states and input/output messages, the ways unexpected events are treated during conversations. As an agent may have multiple conversations at the same time, we also provide constructs for deciding which conversation to continue next and when to suspend or resume a conversation (conversation management).

As agents exist within environments, we also provide constructs for defining the composition of environments and for controlling the activation of the agents in an environment (agent management).

4.3.1. *Agents and environments*

In COOL, an *agent* is a programmable entity that can exchange messages within structured "conversations" with other agents, change state and perform actions. A COOL agent is defined by giving it a name and "plugging in" an interpreter that selects and manages its conversations. The interpreter applies specially defined control rules (called *continuation rules*) to determine which conversation to work on next. In the following example we use an interpreter that also applies the knowledge acquisition and debugging service (`agent-control-ka`) when selecting the next conversation to work on:

```
(def-agent 'customer
  :continuation-control 'agent-control-ka
  :continuation-rules '(cont-1 cont-2 cont-3 cont-4))
(def-agent 'logistics
  :continuation-control 'agent-control-ka
  :continuation-rules '(cont-1 cont-2 cont-3 cont-4))
(def-agent 'plant1
  :continuation-control 'agent-control-ka
  :continuation-rules '(cont-1 cont-2 cont-3 cont-4)).
```

Agents carry out conversations with other agents or perform local actions within their environment. Cooperating agents exist in local or remote *environments*. To control agent execution within an environment, we use *conversation managers*. A conversation manager defines the set of agents it manages, specifies a control function that at each cycle selects an agent for execution and the instrumentation (e.g. tracing, logging, etc.) of agent execution:

```
(def-conversation-manager 'm1
  :agent-control 'execute-agent
  :agents '(customer logistics plant1 ...)).
```

The purpose of the environment is to "run" agents by managing 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. This has the advantage that a set of COOL agents that run in an environment that exists on a single machine will also run without any modification in several environments on several machines. Thus, we can develop and test on a single machine and then deploy with no modification (except for the directory table) on the network. The environment also provides a wealth of tools for visual manipulation — browsing, editing, environment set-up, animated execution.

4.3.2. *Conversations*

Agents interact by carrying out "conversations". Within a conversation, agents exchange messages according to mutually agreed conventions, change state and perform local actions. COOL provides a construct for defining generic conversations, the *conversation class* 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 messages 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, their control mechanism and the local data-base that maintains the state of the conversation. The latter consists of a set of variables whose persistent values (maintained for the entire duration of the conversation) are manipulated by conversation rules. This data-base is the mechanism we currently provide for agents to represent their "mental state" during the interaction represented by the conversation. We do not enforce the use of variables describing explicit mental states like "my current intention" or "my current obligation", but developers can use such a vocabulary if they feel it is appropriate. Conversation rules are indexed on the finite set of values of a special variable, the *current-state.* Because of that conversations admit a finite state machine representation that is often used for visualization and animation purposes. VonMartial[40] describes techniques for designing consistent asynchronous conversations described by finite state machines.

Figure 6 shows the conversation class governing the Customer's conversation with Logistics in our supply chain application. Figure 7 shows the associated transition diagram of this conversation class. Arrows indicate the existence of rules that will move the conversation from one state to another.

```
(def-conversation-class 'customer-conversation
  :name 'customer-conversation
  :content-language 'list
  :speech-act-language 'kqml
  :initial-state 'start
  :final-states '(rejected failed satisfied)
  :control 'interactive-choice-control-ka
  :rules '((start cc-1)
           (proposed cc-13 cc-2)
           (working cc-5 cc-4 cc-3)
           (counterp cc-9 cc-8 cc-7 cc-6)
           (asked cc-10 )
           (accepted cc-12 cc-11))).
```

Fig. 6. Customer-conversation.

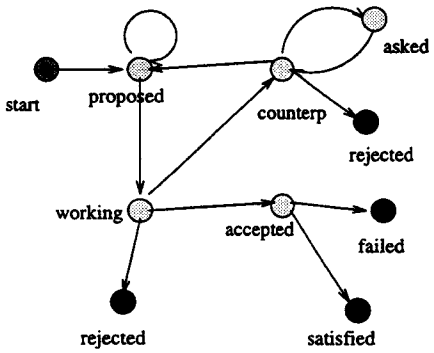

Fig. 7. Finite state representation of customer-conversation.

Fig. 8. Conversation class editor.

Figure 8 illustrates the visual editor for creating or editing conversation classes, with specialized machinery for editing the sets of rules indexed on the conversation state.

Error recovery rules are another component of conversation classes (not illustrated in Fig. 6). They specify how incompatibilities among the state of a conversation and the incoming messages are handled. Such incompatibilities can have many causes — message delays, message shuffling, lost messages, wrong messages sent out, etc. Error recovery rules deal with this by performing any action deemed appropriate, such as discarding inputs, initiating clarification conversations with the interlocutor, changing the state of the conversation or just reporting an error.

Actual conversations instantiate conversation classes and are created whenever agents engage in communication. An actual conversation maintains the current-

state of the conversation, the actual values of the conversation's variables and various historical information accumulated during conversation execution.

Each conversation class describes a conversation from the viewpoint of an individual agent (in Fig. 6 the Customer). For two or several agents to "talk", the executed conversation class of each agent must generate sequences of messages that the others' conversation classes can process. Thus, agents that carry out an actual conversation C actually instantiate different conversation classes internally. These instances will have the same name (C) inside each agent, allowing the system to direct messages appropriately (Fig. 9).

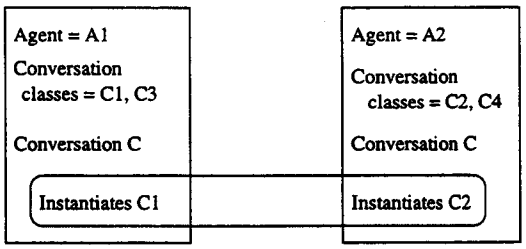

Fig. 9. The same conversation instantiating different conversation classes.

4.3.3. *Conversation rules*

Conversation rules describe the actions that can be performed when the conversation is in a given state. In Fig. 6 for example, when the conversation is in the **working** state, rules `cc-5`, `cc-4` and `cc-3` are the only rules that can be executed. Which of them actually gets executed and how depends on the matching and application strategy of the conversation's control mechanism (the `:control` slot). Typically, we execute the first matching rule in the definition order, but this is easy to change as rule control interpreters are pluggable functions that users can modify

```
(def-conversation-rule 'crn-1
  :current-state 'start
  :received '(propose :sender customer
                      :content(customer-order :has-line-item ?li))
  :next-state 'order-received
  :transmit '(tell :sender ?agent
                   :receiver customer
                   :content '(working on it)
                   :conversation ?convn)
  :do '(put-conv-var ?conv '?order (cadr(member :content ?message)))
  :incomplete nil).
```

Fig. 10. Conversation rule.

at will. Figure 10 illustrates a conversation rule from the conversation class that Logistics uses when talking to Customer about orders.

Essentially, this rule states that when Logistics, in state `start`, receives a proposal for an order (described as a sequence of line-items), it should inform the sender (Customer) that it has started working on the proposal and go to state `order-received`. Note the use of variables like `?li` to bind information from the received message as well as standard variables like `?convn` always bound by the system to the current conversation. Also note a side-effect action that assigns to the `?order` variable of the Logistics' conversation the received order. This will be used later by Logistics to reason about order execution. Among possibilities not illustrated, we mention arbitrary predicates over the received message and the local and environment variables to control rule matching and the checking and transmission several messages in the same rule.

To help users access and modify rules easier, the system provides a browser for conversation classes and a visual editor for the associated conversation rules.

4.3.4. *Initiating conversations*

When an agent wishes to initiate a conversation in which it will have the initiative, it creates an instance of a conversation class. When this conversation instance is executed, messages will be sent and received according to the conversation class. When a message is sent to an agent, the sent performative must contain a `:conversation` slot (an extension to KQML) that contains a conversation name that is shared by the communicating agents. For example, agent `a2` may send to agent `a1` the following message:

```
(propose :sender a2
  :receiver a1
  :content (produce widget 100)
  :reply-with r1
  :conversation c1).
```

Agent `a2` has an actual conversation named `c1` that is managed by the rules of one of `a2`'s conversation classes. If `a1` has an actual conversation named `c1`, then the rules in the conversation class that `a1` associates to its `c1` actual conversation will be used. If receiver `a1` has no conversation `c1`, the message is interpreted as a request for a new conversation made by `a2`. In this case, `a1` must retrieve and instantiate a conversation class able to handle the communication.

Our current mechanism for retrieving the conversation class that will manage a request for a new conversation is based on two elements. First, any message that is a request for conversation may have an additional slot `:intent` (another — and last — extension to KQML) that contains a description of the intent of the requesting agent. The receiving agent tries to find a conversation class that matches the expressed `:intent` of the sender. This is done by having conversation classes

specify an `:intent-test` predicate that will be used with the actual `:intent` as argument. If the test determines that a conversation class can serve the `:intent` of a request, then the second element is used. This is a verification that in the initial state of the selected conversation class there exists at least one rule that can be triggered by the received message. If this is the case, a new (actual) conversation controlled by the retrieved conversation class is created and the receiver agent will use it as its conversation with the sender. Finally, if `:intent` is not specified in the message, the receiver will select a conversation that in the initial state has rules that accept the sent message.

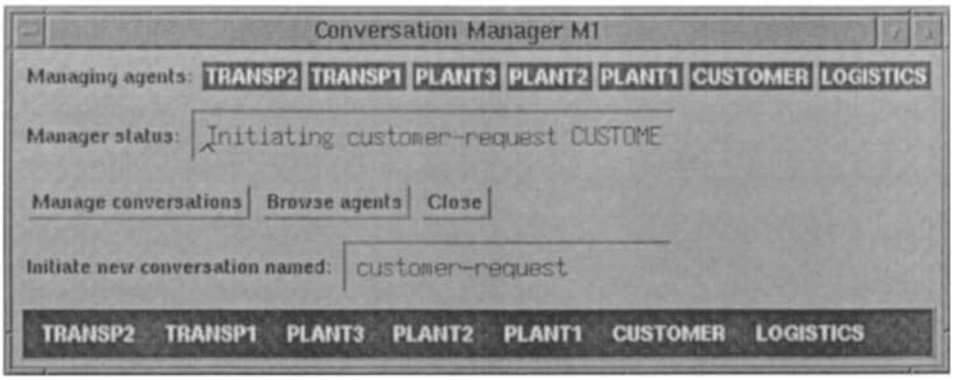

Fig. 11. Setting up the initial conversation.

Figure 11 shows how a set of agents in an environment is interactively set-up for execution within a conversation manager. For each agent, a pull-down menu lists the conversation classes that exist for that agent. The user selects an initial such class and creates an instance of it by simply naming it. Then, if the Manage Conversations button is pressed, the system will start to execute conversations beginning with the created conversation. Usually, we arrange things such that this conversation sends messages to other agents which will then answer, and so on.

4.3.5. *Continuing conversations*

Another element of the framework is the ability of agents to specify their policies of selecting the next conversation to work on. Since an agent can have many ongoing conversations (some may be waiting for input, some may be waiting for other conversations to terminate, others may be ready for execution), the way it selects conversations reflects its priorities in coordination and problem-solving.

The mechanism we use to specify these policies is *continuation rules*. Unlike conversation rules and error recovery rules, which are attached to conversation classes, continuation rules select from among the conversations of an agent and hence are attached to agents.

Continuation rules perform two functions. First, they test the input queue of the agent and apply the conversation class recognition mechanism to initiate new conversations. Second, they test the data base of ongoing conversations and select one existing conversation to execute. Which of these two actions has priority (serving new requests versus continuing existing conversations) and which request or conversation is actually selected, is represented in the set of continuation rules associated to the agent. Our agent definition mechanism allows the specification, for each agent, of both the set of continuation rules and the continuation rule applier.

For illustration, the following continuation rule specifies that a new conversation request is served if there exists a conversation class that accepts the first message in the agent queue:

```
(def-continuation-rule 'cont-1
 :input-queue-test #'exists-conv-class-initially-accepting-1st-msg).
```


4.3.6. *Nested conversation execution*

Nested conversation execution is a conversation execution mode in which the current conversation of an agent is suspended, another conversation is created or continued, with the former conversation being resumed when specified conditions hold (like termination of the spawned conversation). Nested conversation execution of this kind makes it possible to break complex protocols into smaller parts that will be executed much like coroutines in some programming languages. This is important in applications where protocols are complex and need to be broken into manageable pieces.

The mechanism is appropriate in situations like the following. An agent `a`, that has an ongoing conversation with an agent `b`, needs sometime during that conversation to start a new conversation with an agent `c`, for example to acquire information, to achieve a goal or to correct an error. Second, an agent `a` having a conversation with an agent `b` is interrupted during this conversation by a higher priority request from an agent `c`.

To allow for these situations, we let each agent have a set of ongoing conversations. When an agent initiates a new conversation, the new conversation instance is added to this set. When a conversation has to be interrupted because another conversation must take place, the old conversation is suspended, and the system *marks the suspended conversation as waiting for the new conversation to reach a certain state* (in which some condition is true). This creates dependency records among conversations that are used when selecting the next conversation to work on. Because conversations can be 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.

For example, consider again the multiagent supply chain. The Customer agent may have a conversation with the Logistics agent about a new order. Logistics may temporarily suspend this conversation to start a conversation with a Plant agent

to inquire about the feasibility of a due date. Having obtained this information, Logistics will resume the suspended conversation with Customer. This mechanism is intensively used in the logistics execution protocol discussed in Section 7.

4.3.7. *Pluggable rule interpreters and operation regimes*

We have shown that the definitions of both agents and conversation classes have control slots for specifying the interpreters that will handle agent execution and respectively conversation execution. Users can freely develop and use their own interpreters. Up to now we have a number of interpreters for basic operation modes of the system. The basic (default) interpreters carry out conversation selection and execution as explained above. Another set of interpreters support the knowledge acquisition mode of the system explained further in Sec. 4.5. These interpreters have the default behavior as a subset (that is they will function like the default ones if there is no knowledge acquisition to perform) but have the added capability of managing complex graphical interfaces that users employ to dynamically add, refine or debug the coordination knowledge embedded in rules.

Both sorts of interpreters support an operation mode in which once a rule has been found to be applicable, rather than applying it directly, a graphical interface is spawned in which the user can inspect the current execution context, modify the action part of the rule and apply it selectively or as a whole. The purpose of this regime is to offer the foundation for combining rule execution with direct, interactive user action. The actions available when executing a rule may be custom designed to activate external applications, read data or results (from files, databases, etc.) or perform any action in the environment whose effects or results may be needed before or when executing the rule. In particular, active messages that come with attached alternative actions the receiver should choose from in response — as supported by systems like Strudel[50] — can be implemented in this way. The rules that must be executed in this manner are specially tagged in the source form by specifying a function that will manage their execution (actually another pluggable interpreter).

4.3.8. *Legacy software integration*

To integrate legacy software, we simply employ the above conversation mechanism in which we have rules that, rather than checking input messages and sending out responses, activate the legacy application, communicate with it and reason about its operation. This can be done in several ways, ranging from batch execution of an application (by preparing input data, spawning its process, reading the produced outputs) to dynamically interacting through its API functions.

4.4. ***Example: The supply chain***

Going back to the supply chain, we implement the supply chain agents as COOL agents and devise coordination protocols appropriate for their tasks. Figure 12

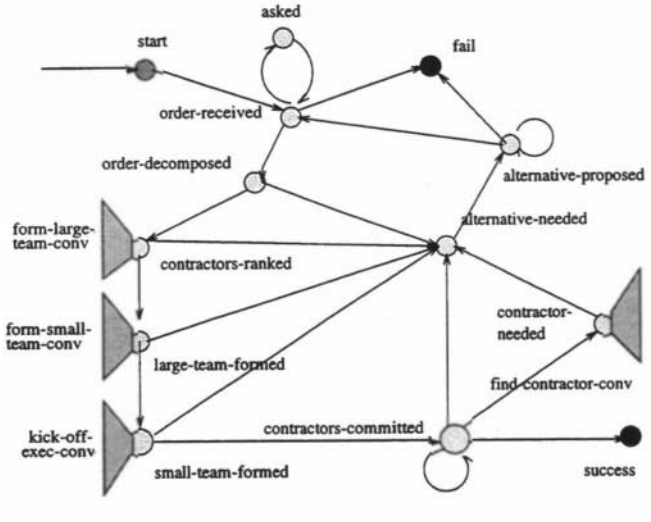

Fig. 12. Logistics execution protocol.

shows the protocol that the Logistics agent executes to coordinate the entire supply chain. The process starts with the Customer agent sending a request for an order (according to `customer-conversation` shown in Figs. 6 and 7). Once Logistics receives the order, it goes to state `order-received`. There, it checks that the order is completely specified, in the sense that it contains all required information. If this is not the case, it will ask specific questions form the Customer to fill in the missing parts. If this is not possible, the conversation ends. When the order is complete, the conversation goes to the `order-decomposed` state. Here, Logistics tries to decompose it into activities like manufacturing, assembly, transportation, etc. and to determine which agents will execute these activities. This is done by running an embedded constraint based logistics scheduler. A rule attached on this state prepares an input file for the scheduler, runs it and then parses the produced output file to extract the activities required to complete the order and the agents that are supposed to carry them out (note that both activities and potential executors are determined by the logistics scheduler — for each activity there can be several potential executors).

If this decomposition fails, Logistics will try to negotiate a slightly different contract with the Customer (by going to state `alternative-needed`). If decomposition succeeds, Logistics tries to form the team of contractors that will execute the activities. This is done in two stages. First, a large team is formed. The large team contains all ranked contractors previously determined by the logistics scheduler that have expressed interest to participate by executing the activity determined previously by Logistics. Membership in the large team does not bind contractors to execute their activity, it only expresses their interest in doing the activity. If the large team was successfully formed (at least one contractor for each activity),

then we move on to forming the small team. This contains exactly one contractor per activity and implies commitment of the contractors to execute the activity. It also implies that contractors will behave cooperatively by informing Logistics as soon as they encounter a problem that makes it impossible for them to satisfy their commitment. In both stages, team forming is achieved by suspending the current conversation and spawning team forming conversations.

```
(def-conversation-rule 'lep-6
  :current-state 'contractors-ranked
  :such-that '(not (get-conv-var ?conv '?forming-large-team))
        ; to prevent multiple spawning of form-large-team-conv
  :next-state 'contractors-ranked
  :do-before '(add-conversation logistics 'form-large-team-class
                                'form-large-team-conv)
  :do-after
   '(progn
      (put-conv-var ?conv '?forming-large-team t)
      (put-conv-var (get-named-conv ?agent 'form-large-team-conv)
                    '?ranked-contractors
                    (get-conv-var ?conv '?ranked-contractors))
      (put-conv-var (get-named-conv ?agent 'form-large-team-conv)
                    '?result nil))
  :wait-for '(form-large-team-conv)
  :incomplete nil)
```

Fig. 13. Rule spawning team forming conversation by Logistics.

For example, in state `contractors-ranked` we form the large team by having Logistics start conversations with each contractor ranked for each activity, in the ranking order. The rule presented in Fig. 13 shows how this happens. First, the rule makes sure that the team forming conversation has not been already spawn (the `:such-that` condition). If this is the case, the action in slot `:do-before` creates a new conversation named `form-large-team-conv`, of class `form-large-team-class`. Then the action in the `:do-after` slot marks team forming as taking place, initializes the `result` variable of the new conversation and transfers some data from the current conversation to the newly created one (this is how new conversations receive the data they need to operate). The transferred data consists of the list of ranked-contractors. The new `form-large-team-conv` conversation will contact each ranked contractor to inquire if they are willing to join the team. The `:wait-for` slot of the rule informs the system that the current conversation will be suspended, waiting for the named conversation to reach some state.

Forming the small team is similar, Logistics will discuss with each member of the large team until finding one contractor for each activity. In this case the negotiation between Logistics and each contractor is more complex in that we can have several rounds of proposals and counter-proposals before reaching an agreement. This is normal, because during these conversations contractual relations are established, while the large team forming conversations had the purpose of only expressing or confirming interest. How the dialogue between Logistics and each contractor takes place when forming the small team is illustrated in Fig. 14 where we show the `form-small-team-class` conversation class used by Logistics and the `answer-form-small-team-class` conversation class used by the contractors.

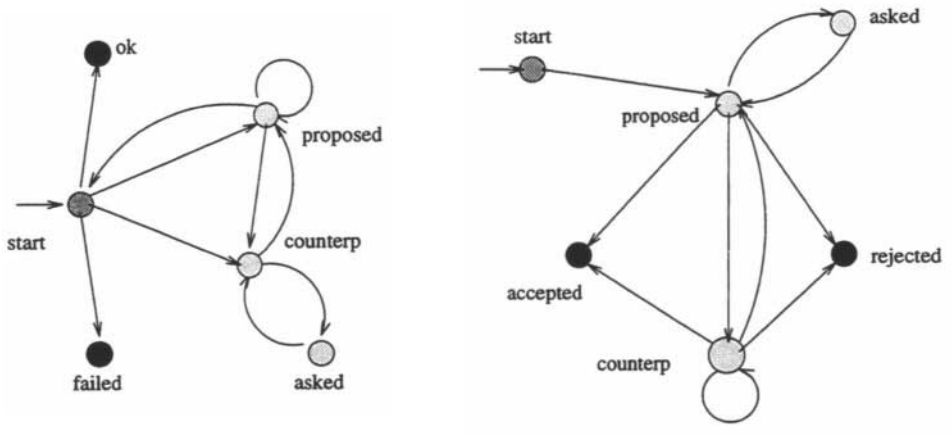

Fig. 14. Team forming conversation classes (logistics and contractors).

An important aspect of the negotiation between Logistics and each contractor in this conversation is that when a contractor counterproposes or rejects an activity, it also reveals to Logistics a list of constraints that it cannot satisfy as the reason for counterproposing or rejecting. Logistics uses these revealed constraints to make a new proposal that would satisfy the revealed constraints or, in case of rejection, remembers them when proposing to a new contractor. This mechanism is important because it focuses negotiations on the problematic issues and thus makes them converge rapidly.

When an activity is assigned to a contractor that will be part of the small team, the other members of the large team are informed about that and asked if they wish to remain in the large team for potential future assignments. If team forming is successful, then we resume the suspended main conversation and move to the next state. This is done by rules like that shown in Fig. 15. This particular rule, tried after the `form-large-team-conv` terminates, applies a test (in slot `:waits-for-test`) on the `form-large-team-conv` conversation. The test checks

```
(def-conversation-rule 'lep-7
  :current-state 'contractors-ranked
  :waits-for-test 'large-team-formed-test
    ;; tests that waited for conversation is terminated and if it
    ;; has produced the team (in one of its variables) it sets the
    ;; ?large-team variable in this conversation and returns t
  :next-state 'large-team-formed
  :incomplete t).
```

Fig. 15. Rule resuming execution of suspended conversation.

that the `form-large-team-conv` has produced the required team in one of its variables. There is a similar rule checking the formation of the small team.

In the `small-team-formed` state we continue with other newly spawned conversations (according to the same pattern) with the team members to kick off execution. After having started execution (which every team member in part acknowledges), we move to state `contractors-committed` where Logistics monitors the activities of the contractors. If contractors exist that fail to complete their activity, Logistics will try to replace them with another contractor from the large team. The large team contains contractors that are interested in the activity and are willingly forming a reserve team, hence it is the right place to look for replacements of failed contractors. If replacements cannot be found, Logistics tries to negotiate an alternative contract (`alternative-needed`) with the Customer. To do that, Logistics relaxes various constraints in the initial order (like dates, costs, amounts) and uses its scheduling tool to estimate feasability. Then, it makes a new proposal to the Customer. Again, we may have a cycle of proposals and counter-proposals before a solution is agreed on. If such a solution is found, the protocol goes back to the `order-received` state and resumes execution as illustrated.

5. In Context Acquisition and Debugging of Coordination Knowledge

Coordination protocols for applications like supply chain integration are generally very complex, hard to specify completely at any time and very likely to change even dramatically during the lifespan of the application. Moreover, due to the social nature of the knowledge they contain, such protocols are better acquired and improved during and as part of the interaction process itself rather than by off-line interviewing of experts. This is especially true in our application context where many agents are supervised by users. Because of this, the coordination tool must support:

(1) *Incremental modifications* of the protocols e.g. by adding or modifying knowledge expressed in rules and conversation objects.
(2) System operation with *incompletely specified protocols*, in a manner allowing users to intervene and take any action they consider appropriate.

(3) System operation in a *user controlled mode* in which the user can inspect the state of the interaction and take alternative actions.

We are satisfying these requirements by providing a subsystem that supports in context acquisition and debugging of coordination knowledge. Using this system execution takes place in a mixed-initiative mode in which the human 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.

Essentially, we allow both conversation and continuation rules to be *incomplete*. An incomplete rule is one that does not contain complete specifications of conditions and actions. Since the condition part may be incomplete we do not really know whether the rule matches or not, hence the system does not try to match the rule itself. Since the action part may be incomplete, the system cannot apply the rule either. All that can be done is to let the user handle the situation. Protocol specifications may contain both complete and incomplete rules in the same time. Incomplete continuation rules are encountered when the system tries to determine the next conversation to work on. Incomplete conversation rules are encountered when executing a given conversation. Assuming the usual strategy of applying the first matching rule in the definition order, we can have two situations. The first is when a complete rule matches. In this case it is executed in the normal way. The second is when an incomplete rule is encountered (hence no previous complete rule matched). In this case the acquisition/debugging regime is triggered. In this regime, a graphical interface is popped up in which the user is presented information about the context of rule execution. For conversation rules, this consists of the current conversation, its status, variables, history, the message queue, the available rules in the current state. The user can browse this information and view each available rule in part. Also, the user can manipulate rules and the message queue by checking rule conditions to determine applicability, editing a rule, creating new rules, moving or removing messages. When the user feels sure about what to do he can execute either an existing, a new, or a modified rule and can instruct the system about retaining the new or modified rules for further use. In this way, the sets of rules are incrementally modified (perhaps becoming complete) as more knowledge is added to the system.

Figure 16 shows an example incomplete rule from the *customer-conversation* that allows a user interacting with the Customer agent to answer (indeterminate) questions from the Logistics agent.

The rule is incomplete in that it does not specify how to answer a question — the `:transmit` part only contains the generic part of the response message. It is designed to work under the assumption that once a question is received, the user will formulate the answer interactively, using the graphical interface provided by the acquisition tool. When the knowledge acquisition interface is popped up, the user will have access to the received message containing the actual question. Using whatever tools are available, the user can determine the answer. Then, the

```
(def-conversation-rule 'cc-13
  :current-state 'proposed
  :received '(ask :sender logistics)
  :next-state 'proposed
  :transmit '(tell :receiver logistics
                   :sender ?agent
                   :conversation ?convn )
  :incomplete t).
```

Fig. 16. Incomplete conversation rule.

user can create a copy of the rule and edit the transmitted message to include the answer. This rule can be executed (thus answering the question) and then discarded. Alternatively, if the new rule contains reusable knowledge, it can be retained, marked as complete and hence made available for automated application (without bothering the user) next time.

The facilities provided by this service can be illustrated with examples from the graphical interface of the service. To view the status of the conversation at the time an incomplete rule was encountered, the acquisition service shows the screen in Fig. 17. Here we have an instance of the logistics execution protocol as seen by the Logistics agent. The finite state abstraction is depicted, together with a textual presentation of the conversation and a browser for the conversation variables.

Another aspect of the conversation context is formed by the available rules. This is shown in Fig. 18. The browser for conversation rules allows the user to inspect the rules indexed on the current state (drawn as a larger circle). Rules can be

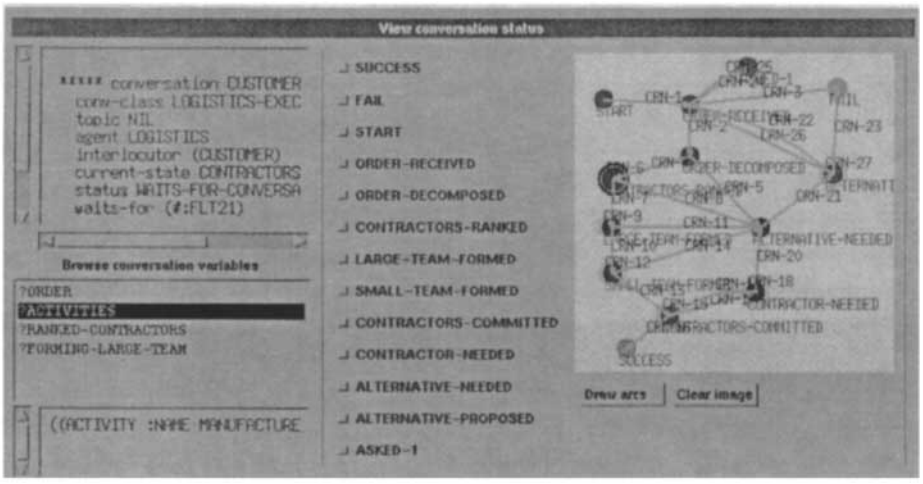

Fig. 17. Viewing the conversation status.

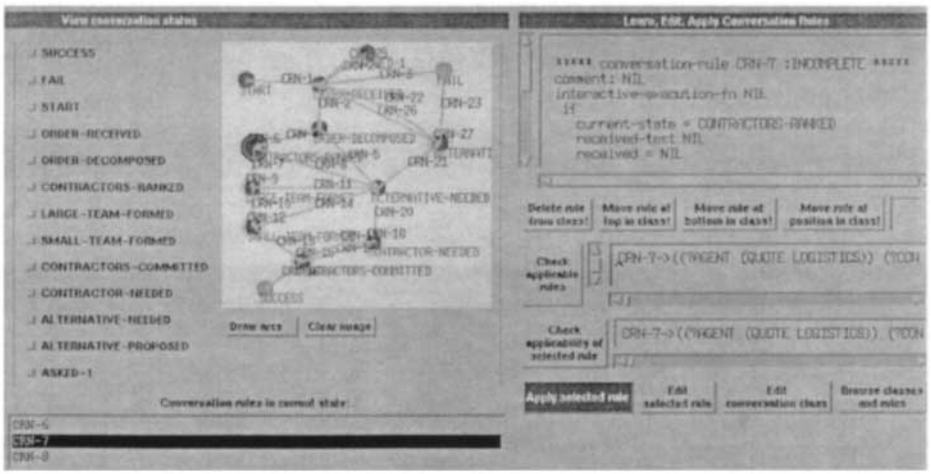

Fig. 18. Inspecting, editing and applying rules.

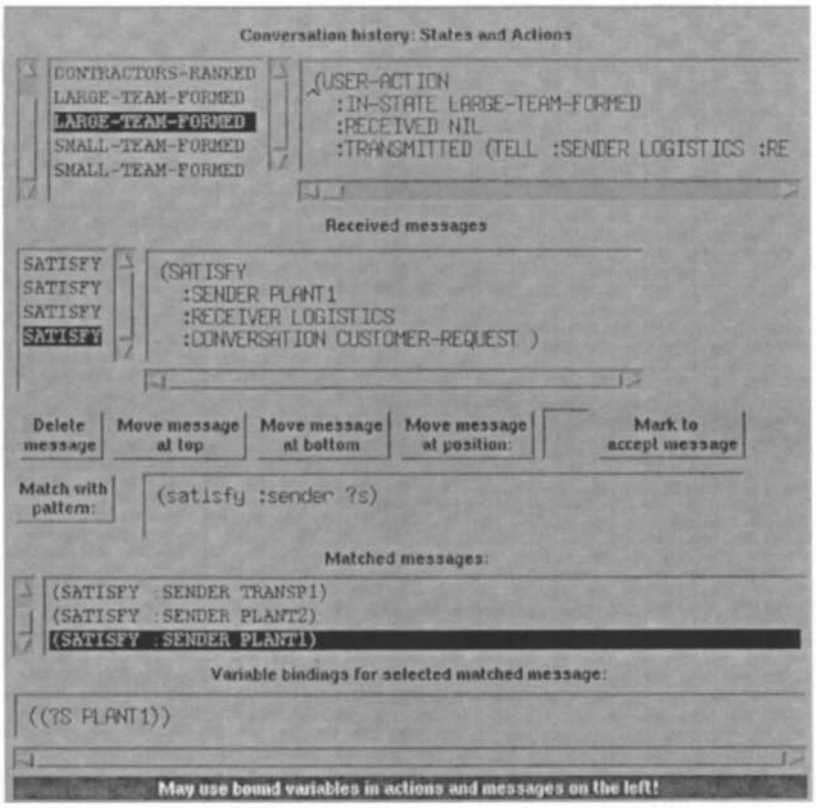

Fig. 19. Inspecting and editing conversation history and messages.

checked for applicability in the current context, with the resulting variable bindings shown so that the user can better assess the impact of each rule. The interface allows the user to perform a number of corrective actions like moving a rule to a different position or removing it from the conversation class. It is also possible to invoke the rule editor, the conversation class editor (if there is something wrong with the conversation class, such as a rule being indexed on the wrong state) or the browser for classes and rules allowing the user to inspect other classes and rules in the system. The effect of any of these modifications will be immediate. Finally, the user can leave the interface and continue execution by applying a specified rule.

When the user needs more information about the history of conversation execution, especially with respect to message exchange, the interface provides presentation and interaction facilities as shown in Fig. 19. First, the history of the conversation can be traced by viewing the sequence of past states and the actions performed in each state (received messages, rule triggered, transmitted message). Second, the messages received (and not yet processed) by the conversation are also displayed. Again, here we provide means for corrective actions, assuming that message transmission is an important source of errors. Amongst them we mention deleting messages and reordering messages in the conversation queue. To better access the content of messages (especially if their number is large or if their content

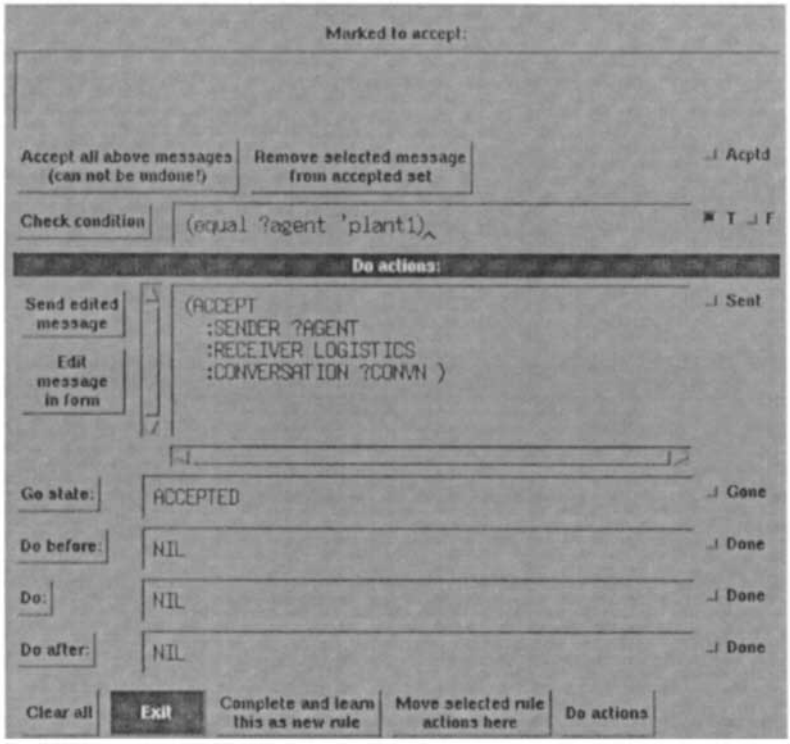

Fig. 20. Editing, executing and learning rule actions.

is complex) we provide pattern based search in which the set of messages is searched for all messages that match a given pattern and we show the variable bindings for the matched messages.

Finally, when the action part of an existing rule is not complete (like in Fig. 16) or is not what the user needs, the service allows the interactive modification of the action part before executing it. This is shown in Fig. 20. First, a set of forms is available for presenting and editing the various slots of the action part (these include the transmitted message, the next state and the side-effect actions). They can be filled automatically from a selected rule. The user can edit these slots and then execute them either separatedly or together. As rule execution may remove messages from the conversation queue, messages shown in the previous part of the interface can be marked as to be removed (or accepted) and actually removed when desired. Arbitrary conditions testing for any conversation variables can be also evaluated in this context to obtain more information. Finally, the modifications performed to the action part can be saved into a new rule that can be "learned" by the system, replacing the original one.

The treatment of incomplete continuation rule is similar, just that this time the system acquires or debuggs control knowledge for conversation selection.

This service provides a unified debugging, development and acquisition environment for cooperation knowledge that has proven invaluable when developing complex multi-agent interactions. Our current policy is that, whenever a new protocol is designed, its first runs are always in this debugging mode (by marking all rules as incomplete). This ensures that the right rules are executed at the right time and helps to quickly detect and correct missing knowledge or other errors.

6. Coordinating Information Distribution and Conflict Management

Coordination is an ubiquitous aspect of multi-agent systems. In this section we review two other services of the shell that require coordination to operate properly — Cooperative Information Distribution and Cooperative Conflict Management. These are complex multi-agent reasoning tasks that play a major role in managing change across the supply chain.

6.1. *Cooperative information distribution*

A first requirement for building agents that are responsive to events is being able to keep agents informed about relevant events. When the number of relevant events is high and their occurrence cannot be predicted in advance — which is the case in the supply chain — we can not propagate information only at explicit request, that is by having every agent asking every other agent every minute if event "X" occured. We need a way to automatically update agents whenever relevant things happen, without relying on explicit demand. *Cooperative information distribution* is a way to achieve this goal. It allows an agent to distribute information to other agents based on the *content* of the information and the expressed *interests* of agents. Agents express an interest by posting a persistent query (once) that will be answered by other

agents whenever relevant information becomes available. If supplied information is ever disbelieved, cooperative behavior requires the producer of the information to send retraction messages to all agents that have received it.

The essential capabilities needed to perform this function are:

(1) Being able to *prove* that a piece of information satisfies an expressed interest of some agent.
(2) Being able to *trace* information that depends on retracted beliefs in order to notify agents affected by retractions.
(3) Being able to *coordinate* information exchange according to this model.

In our system the proof part is performed by the classification and recognition services of the description logic language we use for knowledge management,[1] dependency tracking is supported by the truth-maintenance service of the same description logic and coordination of information exchange is supported by the coordination framework.

In our supply chain architecture we are supporting content based information distribution with the help of Information Agents.[2] These are specialized agents that collect information interests of other agents together with advertisements of what information agents are willing to provide and on this basis mediate information exchange by routing information to interested agents, handling retraction notification and executing the advertise-subscribe protocol that coordinates the process.

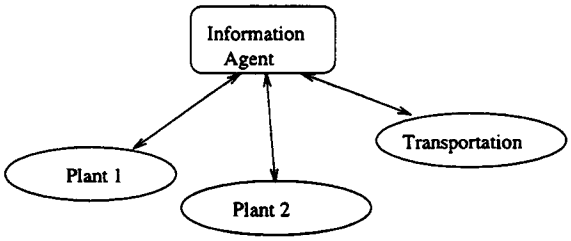

Fig. 21. Information agent servicing functional agents.

```
;; Topic of interest and subscription of Transportation Agent:

(concept heavy-component              ;; any component whose weight
  (:and component (:gt weight 5000)));; is greater than 5000
(subscribe
  :content(stream-about
           :content(query heavy-component
                          [:alltime march-april 94])))
```

Fig. 22. Topics of interest and subscriptions.

```
;; Plant-1 to Information Agent:
(achieve :content (part-of p-111 c-12))
(achieve :content (weight c-12 2700[:starting feb 94]))

;; Plant-2 to Information Agent:
(achieve :content (part-of p-111 c-13))
(achieve :content (weight c-13 3400[:starting jan 94]))

;; Information Agent inferences:
(component p-111)
(weight p-111 6100 [:starting feb 94])
(heavy-component p-111 [:starting feb 94])

;; Information Agent to Transportation Agent:
(tell :content (heavy-component p-111[march-april 94]))

;; Plant-1 to Information Agent:
(deny :content (weight c-12 2700 [:starting april 94]))
(achieve :content (weight c-12 1000 [:starting april 94]))

;; Information Agent to Transportation Agent:
(deny :content
      (tell :content (heavy-component p-111 [march-april 94])))
```

Fig. 23. Content-based information distribution scenario.

The operation of this agent is illustrated in Figs. 21–23. Let us assume that the Transportation Agent is interested to know in advance about produced components that are heavy and thus require special arrangements to transport. It expresses this interest as a KQML subscription to the Information Agent as shown in Fig. 22. As the Information Agent is connected to the plants in the supply chain, it permanently receives information updates about events like production of various parts. Having a definition of what a heavy part means to Transportation and knowing that subparts of the same part will be assembled together to form assemblies whose weight is the sum of parts, the Information Agent is able to infer in which cases heavy components are produced and inform Transportation about that. If information is retracted, affected inferences are retracted as well and recipients of retracted information notified. Figure 23 shows how these things happen, using KQML messages and propositional representations as content. Figuring out the structure of the conversations that model this interaction is an easy exercise left to the reader.

6.2. *Cooperative conflict management*

Another important problem in multi-agent systems that manage dynamic events is

dealing with conflicting information. When the future course of action of an agent depends on conflicting information, the agent must make choices about what to believe/disbelieve in order to continue. Our view is that deciding between what to believe and what to disbelieve must be based on multi-dimensional reasoning about information and the individual and social consequences of adopting/rejecting beliefs. Cooperative behavior requires that decisions that impact other agents be negotiated with them, thus making the conflict management process truly social.

The sort of information conflict that we are dealing with occurs from local reasoning and communication as illustrated by the following general scenario:

(1) Agent-1 believes p and communicates it to Agent-3.
(2) Agent-2 believes q and communicates it to Agent-3.
(3) Agent-3 believes p because it was communicated by Agent-1, q because it was communicated by Agent-2 and has local knowledge stating that $p\&q \Rightarrow false$.

When this contradiction prevents Agent-3 from taking action, it has to be eliminated by retracting some current belief that supports either p or q. In this section we address the issue of how to determine which of the possible supporting beliefs to retract to reinstall consistency and how to behave cooperatively with the agents that are affected by the retraction.

6.2.1. *Credibility and utility of information*

Consider again the supply chain. Assume that one customer expresses an intent to place an order with a certain due date, for example `(due-date 01 (13 march 95))` and involving materials from a certain supplier with which the plant already has an on-going contract. In the same time, another customer expresses an intent to place a second order, for example `(due-date 02 (15 march 95))`, with a due date close to the first one, involving materials from a different supplier (with whom there is no contract yet) but whose production cannot be scheduled in the same time because the plant's capacity would be exceeded. The plant has to make a choice between the two customers before accepting a contract. Solving the problem requires multi-dimensional reasoning involving evaluation of what is gained/lost if either order is accepted. Among the questions to be answered in this evaluation are the following. Which customer is more credible in its intent to place an order? With which customer has the plant a more important relationship? Is any customer more likely to be unable to pay in time? How costly will it be to receive materials from the first supplier without using them or how costly will it be to cancel the first supplier's contract? How important is the relationship with each of the two suppliers?

We classify the aspects dealt with by the above questions into two classes. The first class comprises those properties of information that are relevant to its *credibility*, while the second considers the properties that are relevant to its *utility*. *Credibility* measures the precedence of information coming from various sources. It is based on things like the legal or organizational *authority* of the source, its

reliability, the *specificity* or the *recency* of the communicated information. We assume that we can estimate the aspects related to credibility and on that basis estimate a unique numerical measure of credibility $c_a(b)$, for each belief b of any agent a.

Estimating credibility is not enough to decide what to retract. We also need to reason about the impact of retracting a belief over the entire system, in terms of how much we lose if a belief is retracted in the existing situation. For example, decisions might have been made based on beliefs, and retracting these beliefs may imply undoing the decisions. This may cost money, may incur losing useful relations or may affect one's image. Another loss is that of potential gains that we might had obtained by adopting the retracted belief. Examples in the supply chain include retracting orders which causes undoing lots of planning/scheduling work or retiring manufacturing machines which similarly may have big impacts on scheduling decisions. We quantify these aspects in a measure of information *utility*. The main aspects of information utility comes from include:

- *Money costs.* For example, assume that the plant decides to go for the second customer and cancel the contract with the first supplier. This may incur paying substantial penalties besides losing the first customer's contract.
- *Loss of credibility, relations or image.* In the above example, besides losing money the organization may affect its relations with the first customer and the supplier whose contract was cancelled, and this may create a long range problem especially if cancellation was unexpected or the partners are strongly affected.

We assume we can compute a numerical estimate of information utility — $u_a(b)$ is the estimated measure of utility for information b as computed by agent a. Like in the case of credibility, there can be many aspects that determine utility and their complete delimitation may be possible only on the basis of each application domain in part.

Credibility and utility affect belief retraction in the same sense: beliefs with high credibility/utility are harder to retract. For credibility, this comes from the difficulty of having to contradict a highly credible source. For utility, this comes from the important consequences that retraction may have. The estimation of both utility and credibility requires an agent to put itself into another agent's place. As this may be particularly hard to do, we allow agents to carry out confirmation conversations in which the credibility and utility of beliefs are established. For example, an agent may inquire a producer of information about how credible the information is, or a consumer of information about how useful the information actually is. We assume agents are honest when exchanging such estimates.

For information that is consumed and used by several other agents, the sum of consumers' utilities would normally be used to compute the overall utility. When agents derive new information from existing information, the utility of derived information must be considerd when computing the overall utility.

6.2.2. *What to retract and how to behave when retracting*

Suppose now that we have determined a set $\{p_i\}$ of premises which supports a $p\&q \Rightarrow false$ contradiction, in the sense that each p_i is the starting point of a local derivation chain supporting either p or q. A set $\{p_i\}$ thus defined is called a conflict set for the given contradiction. Without going into details about how to obtain the actual conflict set(s), let us attach to each p_i its credibility and utility measures. We can represent these two values by points in a diagram called a *c-u space*, as illustrated in Fig. 24.

An important aspect of the approach is that when an agent decides to retract a belief, it must exhibit cooperative behavior towards the other agents that share this belief. The kind of required cooperative behavior depends on the credibility and utility of the retracted belief. If these values are low, then it would be enough for the agent to notify the others about the retraction. If these values are higher, the agent may request advice before retracting. Finally, if these values are very high, the agent may become incompetent or unauthorized to decide on retracting, and may need to request permission to retract. We capture these different behaviors by distinguishing among several regions of the c-u space that contain beliefs whose retraction requires different cooperative behavior of the agent. There are three broad such regions (each may have subregions where the nature of cooperative behavior may vary). Figure 24 shows these qualitative regions of the c-u space.

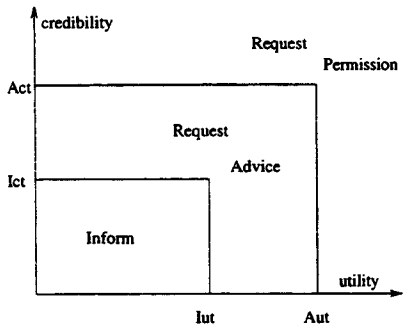

Fig. 24. Cooperative behavior regions in the c-u space.

(1) *Belief is easy to retract.* Propositions with low values of credibility and utility — as defined e.g. by some threshold values Ic_t and Iu_t — are easy to retract because we do not contradict highly credible sources and we are not likely to produce serious consequences. An agent can unilaterally retract such a proposition. Cooperative behavior principles would however require agents performing such retractions to inform the others about the retraction. Since retraction was entirely decided by the agent, the agent also bears the entire responsibility for the act.

(2) *Belief is harder to retract.* Propositions with higher values for credibility or utility — for example up to another set of thresholds like Ac_t and Au_t — are harder to retract because credibility or authority now have significant values. Since retractions from this zone imply either contradicting a credible source or producing significant consequences, an agent will have to negotiate with producers and/or consumers about retraction. At this level the negotiation essentially consists of *Requesting Advice* w.r.t. retraction from producers and consumers. Requesting advice implies that the agent presents its intention to retract a shared belief to a producer or consumer and requests their opinion about the retraction. A producer or consumer can advise on proceeding with the retraction, cancelling it or be neutral. No matter what advice is received, the agent will continue to be responsible for its decision.

(3) *Belief is very hard to retract.* Propositions whose credibility or utility is highest — for example higher than the advise thresholds Ac_t and Au_t — go beyond the decision authority of the agent, either because they come from a very important source or because their consequences are too important for the agent to judge. In this case, the agent's behavior is that of *Requesting Permission* for retraction. The agent will request permission from producers/consumers to retract a shared belief. In this situation the agent is not competent enough to contradict the source and/or has not enough authority in the organization to decide about the consequences of this retraction. In this case, the problem is actually passed on to somebody else and the responsibility of the agent is limited while the responsibiliy of the interlocutors is dominant.

Based on these ideas we have built an interactive framework for contradiction removal where the user makes the final decision about what to retract, how and when and the system assists with tracing, presenting and evaluating the relevant information. A visual interface allows the user to visualize the conflict set in the c-u space and inspect or modify the credibility/utility values. Special conversation classes specify the cooperative behavior of agents when performing retractions. As shown, this behavior ranges from simple notification to more complex negotiation for requesting advice or permission.

Unlike previous work on distributed truth-maintenance,[9,41,45] our model places more emphasis on *reasoning* and *negotiation* about how to solve conflicts and avoids full automation of the process which is often plagued by arbitrary "algorithm-made" decisions. By placing the human user back into the loop, we take advantage from his/her situated reasoning abilities that can make decision making realistically useful. References 54 and 56 are examples of previous work exploring negotiation as a means to mediate among conflicting agents. Some recent approaches to conflict resolution use prioritized defaults[27] or model-theoretic solutions.[38] They provide semantic accounts of conflict resolution based on a single order of precedence of beliefs. This is very useful, but still has to address the issues of computational architectures and negotiation.

7. Related Work

Our coordination system is related to a number of other efforts briefly reviewed here. *KQML*[19] is a high-level agent communication language that provides a message format and a set of communication acts with informally specified semantics. COOL can be seen as providing an operational semantics to KQML, as it defines the structure of agent agreed message exchanges. As KQML does not have a declarative semantics at the time of writing, it is hard to check if the operational use of KQML in COOL conversations complies with the agreed meanings of KQML acts and actually lets one introduce their own communication acts freely. This impedes interoperability and reusability. When declarative semantics will be available, COOL protocols will be easier to check for proper use of KQML and this will improve reusability for both COOL and KQML.

COOL is also related to a number of previous systems originating from the computer supported cooperative work community, like Strudel,[50] Conversation Builder[34] and The Coordinator[43]. In these systems the emphasis is on supporting humans in performing their work. As such, users are required to directly generate communication acts and be aware of the obligations they create. Our work extends these electronic conversation concepts for inter-agent coordination adding standardized communication based on KQML, rule-based specifications at several levels (agent control, conversation execution and exception handling) and knowledge acquisition services. We use coordination structures in a context in which programmed agents rather than human users are in direct contact with communication acts and their implications. This makes it possible for agents to mediate user interaction in more varied and adequate ways.

ARCHON[33] is a general purpose architecture used to develop agent systems in real world domains like electricity distribution and supply. It supports large grain, loosely coupled, and semi-autonomous agents. In ARCHON cooperation knowledge had to be manually coded into a general representation language. We are trying to improve on that by coming with more generic tools like COOL. We continue this trend in our agent building shell by providing reusable, application independent tools for other cooperative services related to information distribution and conflict management.

Agent infrastructures for engineering are aimed at bringing previously isolated design and engineering tools on-line. One set of solutions correspond to the SHADE[35] architecture: KIF as the interlingua, KQML as the speech act language, and the use of facilitator agents (like the SHADE Matchmaker) that match and route advertisements and subscriptions among the set of cooperating agents. Our agent building shell supports facilitator-type services as cooperative information distribution services that any agent can provide if needed. With COOL we have advanced in building an application independent coordination layer on top of KQML, making it much easier to capture, use and reuse complex coordination protocols.

Finally, Ref. 46 is an important precursor of the work described here as it first proposed a multi-layer *enterprise information architecture* that would integrate in-

formation processing from the network communication layer to market based coordination and negotiation.

8. Conclusions

While there exist several meanings of "agents" in the literature, our work builds on agents understood as the high level building blocks of computing architectures designed to interoperate globally on networks forming a virtual, unifying platform. Critical enterprise applications like supply chain integration cannot be developed and fully exploited unless such programming environments become available. From the research perspective, attacking such applications requires the merger of theoretical work on the nature of agenthood and coordination with the development of practical architectures and tools where principles can be tested, evaluated and exploited. Adopting the view that social interactions in artificial multi-agent systems are described by a distinct level of knowledge and noticing the stringent need for conceptualizations and systems operating at this level we have developed generic agent building tools that are able to capture, represent and utilize this level of knowledge. The social nature of coordination knowledge poses special problems, not only because it confers its complexity, but also because this implies that coordination knowledge can be acquired from or through the interaction process itself rather than from off-line interviewing of experts. To add to these difficulties, there are few paradigms to guide modeling of coordination by computer languages.

We have responded to these issues by building an application independent language and programming environment that can be used to model, execute and acquire coordination knowledge. We assume a communication act based model of interaction and we devise a number of constructs for representing coordination conventions. These include a programmatic notion of conversation with distinct rule sets controlling conversation execution, conversation selection and exception handling, constructs for agents and their environments, plug-in interpreters for conversation selection and execution and mechanisms for multiple conversation management. The programming environment we provide supports distributed agent execution and visual instruments for browsing, editing and execution monitoring. To deal with in context acquisition and debugging of coordination knowledge, we introduce incomplete rules and provide an interactive acquisition environment which, in requested circumstances, handles the control to users and supports them in acquiring and modifying rules and conversation descriptions. Using these tools we have developed complex reasoning services for information distribution and conflict management well as complex coordination mechanisms for integrating multi-agent supply chains of manufacturing enterprises.

Acknowledgments

This research is supported, in part, by the Manufacturing Research Corporation of Ontario, Natural Science and Engineering Research Council, Digital Equipment

Corp., Micro Electronics and Computer Research Corp., Spar Aerospace, Carnegie Group and Quintus Corp.

References

1. M. Barbuceanu, Models: Toward integrated knowledge modeling environments, Knowledge Acquisition 5, 1993, 245–304.
2. M. Barbuceanu and M. S. Fox, The information agent: An infrastructure for collaboration in the integrated enterprise, ed. S. M. Deen, Cooperating Knowledge Based Systems, DAKE Centre, University of Keele, 1994, 257–295.
3. M. Barbuceanu and M. S. Fox, The architecture of an agent building shell, *Proc. Workshop on Agent Theories*, Architectures and Languages, IJCAI 95, August 1995.
4. M. Barbuceanu and M. S. Fox, Conflict management with a credibility/deniability model, ed. S. Lander, *Proc. AAAI-94 Workshop on Models of Conflict Management for Cooperative Problem Solving*, AAAI Technical Report, 1994.
5. M. Barbuceanu and M. S. Fox, COOL — A language for describing coordination in multi-agent systems, ed. V. Lesser, *Proc. First Int. Conf. on Multi-Agent Syst.*, AAAI Press/The MIT Press, 17–24.
6. A. Borgida, R. J. Brachman, D. L. McGuiness and L. Resnick, CLASSIC: A structural data model for objects, *Proc. 1989 ACM SIGMOD Int. Conf. on Management of Data*, 1988, 59–67.
7. R. J. Brachman and J. G. Schmolze, An overview of the KL-ONE knowledge representation System, Cognitive Science **9**, 2 (1985) 171–216.
8. M. Bratman, Intentions, plans and practical reason (Harvard University Press, 1987).
9. D. M. Bridgeland and M. N. Huhns, Distributed truth maintenance, *Proc. AAAI-90*, 1990, 72–77.
10. C. Castelfranchi, Commitments: From individual intentions to groups and organizations, *Proc. First Int. Conf. on Multi-Agent Syst.* (AAAI Press/The MIT Press, 1995) 41–48.
11. W. J. Clancey, Heuristic classification, Artificial Intelligence 27, 1985, 289–350.
12. P. R. Cohen and H. Levesque, Intention is choice with commitment, Artificial Intelligence 42, 1990, 213–261.
13. P. R. Cohen and H. Levesque, Teamwork, Nous 15, 1991, 487–512.
14. P. R. Cohen, J. Morgan and M. Pollack, Intentions in communication (MIT Press Cambridge, MA. 1990).
15. P. R. Cohen and H. Levesque, Communicative actions for artificial agents, ed. V. Lesser, *Proc. First Int. Conf. on Multi-Agent Syst.* (AAAI Press/The MIT Press, 1995) 65–72.
16. K. S. Decker and V. Lesser, Designing a family of coordination algorithms, *Proc. First Int. Conf. on Multi-Agent Syst.* (AAAI Press/The MIT Press, San Francisco, 1995) 73–80.
17. E. H. Durfee, Coordination of distributed problem solvers (Kluwer Academic Press, 1988).
18. E. H. Durfee and V. Lesser, Partial global planning: A coordination framework for distributed hypothesis formation, *IEEE Trans. on Syst., Man and Cybernetics* **21**, 6 (1991) 1363–1378.
19. T. Finin *et al.*, Specification of the KQML agent communication language, The DARPA Knowledge Sharing Initiative, External Interfaces Working Group, 1992.
20. M. S. Fox, Beyond the knowledge level, ed. L. Kerschberg, Expert Database Systems (Benjamin/Cummings Publishing Company, 1987) 455–463.
21. M. S. Fox, A common-sense model of the enterprise, *Proc. Industrial Engineering Research Conf.*, 1993.

22. M. S. Fox, M. Barbuceanu and M. Gruninger, An organisation ontology for enterprise modeling: Preliminary concepts for linking structure and behavior, *Proc. Fourth Workshop on Enabling Technologies, Infrastructure for Collaborative Enterprises* (IEEE Computer Society Press, 1995).
23. M. R. Genesereth and R. E. Fikes, Knowledge interchange format, Version 3.0, Reference Manual, Computer Science Department, Stanford University, Technical Report Logic-92-1, 1992.
24. M. R. Genesereth and S. Ketchpel, Software agents, *Comm. ACM* **37**, 7 (1994) 100–105.
25. M. P. Geogeff, A theory of action for multi-agent planning, *Proc. National Conf. on AI*, Austin, 1984, 1250–129.
26. R. McGregor and R. Bates, The LOOM knowledge representation language, ISI-IRS-87-188, USC/ISI Marina Del Rey, CA, 1987.
27. B. N. Grosof, Conflict resolution in advice taking and instruction for learning agents, IBM Research Report RC 20123, T. J. Watson Research Center, June 1995.
28. T. R. Gruber, Toward principles for the design of ontologies used for knowledge sharing, Report KSL 93-04, Stanford University, 1993.
29. R. V. Guha and D. B. Lenat, CYC: A mid term report, AI Magazine **11**, 3 (1990) 32-59.
30. N. R. Jennings, Towards a cooperation knowledge level for collaborative problem solving, *Proc. 10th European Conf. on AI*, Vienna, Austria, 1992, 224–228.
31. N. R. Jennings, Commitments and conventions: The foundation of coordination in multi-agent systems, The Knowledge Engineering Review **8**, 3 (1993) 223–250.
32. N. R. Jennings and E. Mamdani, Using joint responsibility to coordinate collaborative problem solving in dynamic environments, *Proc. 10th National Conf. on AI*, San Jose, CA, 1992, 269–275.
33. N. R. Jennings, Controlling cooperative problem solving in industrial multi-agent systems using joint intentions, Artificial Intelligence **75**, 2 (1995) 195–240.
34. S. M. Kaplan, W. J. Tolone, D. P. Bogia and C. Bignoli, Flexible, active support for collaborative work with conversation builder, *CSCW 92 Proc.*, 1992, 378–385.
35. D. Kuokka, J. McGuire, J. Weber, J. Tenenbaum, T. Gruber and G. Olsen, SHADE: Knowledge based technology for the re-engineering problem, Technical Report, Lockheed Artificial Intelligence Center, 1993.
36. Y. Labrou and T. Finin, A semantics approach for KQML — A general purpose communication language for software agents, University of Maryland, 1993.
37. H. J. Levesque, P. R. Cohen and J. H. Nunes, On acting together, *Proc. Eighth National Conf. on AI*, Boston, 1990, 94–99.
38. J. Lin, Integration of weighted knowledge bases, To appear in Artificial Intelligence Journal.
39. T. W. Malone and K. Crowston, Toward an interdisciplinary theory of coordination, Center for Coordination Science Technical Report 120, MIT Sloan School, 1991.
40. F. vonMartial, Coordinating plans of autonomous agents, Lecture Notes in Artificial Intelligence 610 (Springer-Verlag, Berlin, Heidelberg, 1992).
41. C. Mason and R. R. Johnson, DATMS: A framework for distributed assumption based reasoning, eds. Les Gasser and Michael N. Huhns, Distributed Artificiall Intelligence, Volume II (Pitman Publishing, London, 1989) 293–317.
42. J. McDermott, A taxonomy of problem solving methods, ed. S. Marcus, Automating Knowledge Acquisition for Expert Systems (Kluwer Academic Press, 1988) 225–256.
43. R. Medina-Mora, T. Winograd, R. Flores and F. Flores, The action workflow approach to workflow management technology, *CSCW 92 Proc.*, 1992, 281–288.

44. R. Patil, R. Fikes, P. Patel-Schneider, D. McKay, T. Finin, T. Gruber and R. Neches, The ARPA knowledge sharing effort: Progress report, eds. B. Nebel, C. Rich and W. Swartout, Principles of knowledge representation and reasoning, *Proc. Third Int. Conf. (KR'92)* (Morgan Kaufmann, San Mateo, CA, November, 1992).
45. C. Petrie, Revised dependency-directed backtracking for default reasoning, *Proc. AAAI-87*, 1987, 167–172.
46. M. Roboam and M. S. Fox, Enterprise management network architecture: A tool for manufacturing enterprise integration, *Artificial Intell. Applications in Manufacturing* (AAAI Press/MIT Press, 1992).
47. S. R. Rosenschein and L. P. Kaebling, A situated view of representation and control, *Artificial Intell.* **73**, 1–2 (1995) 149–173.
48. J. Searle, Speech acts: An essay in the philosophy of language (Cambridge University Press, Cambridge, UK, 1969).
49. J. Searle, Collective intentions and actions., eds. P. R. Coehn, J. Morgan and M. E. Pollak, *Intentions in Commun.* (MIT Press, 1991) 401–416.
50. A. Shepherd, N. Mayer and A. Kuchinsky, Strudel — An extensible electronic conversation toolkit, *CSCW 90 Proc.*, 1990, 93–104.
51. Y. Shoham, Agent-oriented programming, *Artificial Intell.* **60** (1993) 51–92.
52. Y. Shoham and M. Tennenholtz, On social laws for artificial agent societies: Off-line design, *Artificial Intell.* **73**, 1–2 (1995) 231–252.
53. R. M. Smith, The contract net protocol: High level communication and control in a distributed problem solver, *IEEE Trans. on Computers* **29**, 12 (1980) 1104–1113.
54. K. Sycara, Multi-agent compromise via negotiation, eds. Les Gasser and Michael N. Huhns, Distributed Artificiall Intelligence, Volume II (Pitman Publishing, London, 1989) 119–137.
55. B. J. Wielinga, A. Th. Schreiber and J. A. Breuker, KADS: A modeling approach to knowledge acquisition, Knowledge Acquisition **4**, 1, 1992.
56. G. Zlotkin and J. S. Rosenschein, Negotiation and task sharing among autonomous agents in cooperative domains, *Proc. IJCAI-89*, Detroit, MI, 1989, 912-917.