Coordinating Multiple Agents in the Supply Chain

Mihai Barbuceanu and Mark S. Fox
mihai,msf@ie.utoronto.ca
Enterprise Integration Laboratory
University of Toronto
4 Taddle Creek Road, Toronto, M5S 3G9 On, Canada

Abstract

The agent view provides a level of abstraction at which we envisage computational systems carrying out cooperative work by interoperating across networked 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. In this paper we propose a conceptualization of the coordination task around the notion of structured "conversation" amongst agents. Based on this notion we build a complete multiagent programming language and system for explicitly representing, applying and capturing coordination knowledge. The language provides KQML-based communication, an agent definition and execution environment, support for describing interactions as multiple structured conversations among agents and rule-based approaches to conversation selection, conversation execution and event handling. 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 across networks linking people, organizations and machines on a single virtual platform. Coordinating the behavior of the individual agents to achieve both the individual and the shared goals of the participants is one major problem to be solved before we can make effective use of agent communities.

Coordination has been defined as the process of *managing dependencies between activities* [8]. 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 (i) 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 (ii) 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 agents usually have incomplete knowledge of the environment and of the consequences of their actions and the environment changes dynamically making it more difficult to evaluate the current situation and the possible outcomes of 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 explore the view that the coordination problem can be tackled by explicitely representing knowledge about the interaction processes taking place among agents. Jennings [5] 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. We investigate this hypothesis by proposing a conceptualization of coordination around the notion of structured "conversations" amongst agents, by building a complete programming language and system for this conceptualization and by applying this system to the coordination of supply chain agents in an enterprise integration domain. The paper is organized as follows. In section 2 we describe the organization of the supply chain as a multiagent system. In section 3 we discuss our coordi-

1080-1383/96 $5.00 © 1996 IEEE
Proceedings of WET ICE '96

nation system. In section 4 we show how the system is applied to manage coordination in the supply chain.

2. Integrating the Supply Chain

The supply chain of a modern enterprise is a worldwide 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 can not 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 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 1 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 companies 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.

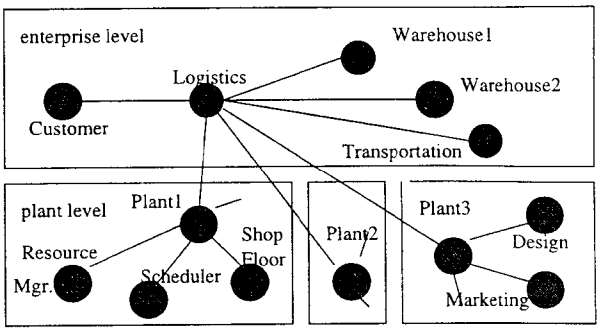

Figure 1. Multi-level supply chain.

3. The Coordination Language

3.1. 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. Also, we do not impose any content language. We have implemented a mail system for KQML messages providing TCP/IP supported transport and services like persistent storage of received KQML messages, visual tools for message browsing, composition, sorting and general pattern matching. The following example illustrates the form of extended KQML we are working with.

```
(propose                    ;; new
 :language KIF              ;; communicative
 :sender A                  ;; action
 :receiver B
 :content (or (produce 200 widgets)
              (produce 400 widgets))
 :conversation C1           ;; new slot
 :intent (explore           ;; new slot
          fabrication possibility))
```

3.2. 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.

```
(def-agent 'customer
  :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. Agents exist in local or remote *environments*. To control agent execution within an environment, we use *conversation managers*. These specify the set of agents they manage, a pluggable control function selecting agents 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.

3.3. 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. 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

```
(def-conversation-class 'customer-conversati
 :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)))
```

Figure 2. Customer-conversation.

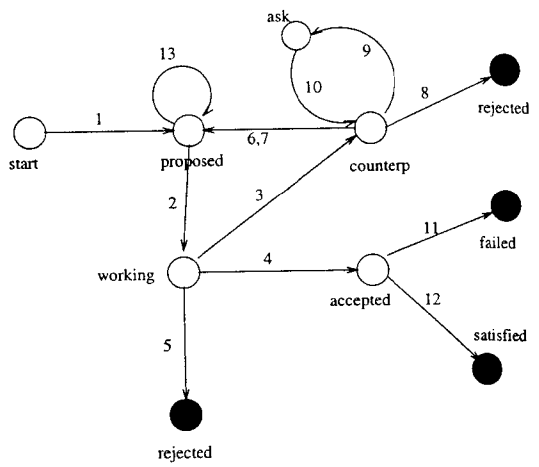

Figure 3. Finite state representation of customer-conversation.

machine representation that is often used for visualiz tion and animation purposes. VonMartial [9] describ techniques for designing consistent asynchronous co versations described by finite state machines. Figure shows the conversation class governing the Custome conversation with Logistics in our supply chain applic tion. Figure 3 shows the associated transition diagra of this conversation class.

Error recovery rules are another component of co versation classes. They specify how incompatibiliti among the state of a conversation and the incomi messages are handled. Such incompatibilities can ha many causes - message delays, message shuffling, lc messages, wrong messages sent out, etc. Error recove rules deal with this by performing any action deem appropriate, such as discarding inputs, initiating cla ification conversations with the interlocutor, changi the state of the conversation or just reporting an erro

```
(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)
```

Figure 4. Conversation rule.

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 figure 2 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.

3.4. Conversation Rules

Conversation rules describe the actions that can be performed when the conversation is in a given state. In figure 2 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 at will. Figure 4 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 of several messages in the same rule.

3.5. 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, it must contain a `:conversation` slot (an extension to KQML) that contains the name of an actual conversation that the recipient already has or will create to handle the interaction. To find out which conversation class to instantiate in response to a message, a receiver agent may use either the value of the `:intent` slot of the message (another extension to KQML) matching it with known intents that its conversation classes can handle, or simply select a conversation class that in the initial state has rules that accept the message just sent.

3.6. Continuing Conversations

Agents are able to specify their policies for selecting the next conversation to work on. Since an agent can have many ongoing conversations, the way it selects conversations reflects its priorities in coordination and problem-solving. The mechanism we use to specify these policies is *continuation rules*. 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.

3.7. 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 or when an agent must dynamically switch focus of attention due to various events.

4. In Context Acquisition and Debugging

Coordination structures 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 contained, they are better acquired and improved in an emergent fashion, during and as part of the interaction process itself rather than by off-line interviewing of users, which for widely distributed systems will be hard to locate and co-locate anyway. Because of this the coordination tool must support (i) *incremental modifications* of the structure of interactions e.g. by adding or modifying knowledge expressed in rules and conversation objects, (ii) system operation with *incompletely specified interaction structures*, in a manner allowing users to intervene and take any action they consider appropriate (iii) 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 conversation 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 don't really know whether the rule matches or not, hence the s tem does not try to match the rule itself. Since t action part may be incomplete, the system can r apply the rule either. All that can be done is to the user handle the situation. Interaction specifi tions may contain both complete and incomplete ru in the same time. Assuming the usual strategy of a plying the first matching rule in the definition order, can have two situations. The first is when a compl rule matches. In this case it is executed in the norn way. The second is when an incomplete rule is enco tered (hence no previous complete rule matched). this case the acquisition/debugging regime is trigger with the user in control over what to do in the resp tive situation, as explained further on.

```
(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)
```

Figure 5. Incomplete conversation rule.

Figure 5 shows an example incomplete rule fr the *customer-conversation* that allows a user intera ing with the Customer agent to answer (indetermina questions from the Logistics agent.

The rule is incomplete in that it does not spec how to answer a question - the `:transmit` part o contains the generic part of the response message. is designed to work under the assumption that onc question is received, the user will formulate the ans interactively, using the graphical interface provided the acquisition tool. When the knowledge acquisit interface is popped up, the user will have access to received message containing the actual question. Us whatever tools are available, the user can determine answer. Then, the user can create a copy of the r and edit the transmitted message to include the swer. This rule can be executed (thus answering question) and then discarded. Alternatively, if the n rule contains reusable knowledge, it can be retain marked as complete and hence made available for tomated application (without bothering the user) n time.

The facilities provided by this service can be ill trated with examples from its graphical interface. view the status of the conversation at the time incomplete rule was encountered, the acquisition s

vice shows the finite state abstraction (like in figure 6). Here we have an instance of the logistics execution process as seen by the Logistics agent.

The rules indexed on the current state (drawn as a larger circle) can be 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 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.

5. Back to the Supply Chain

Going back to the supply chain, we implement the supply chain agents as COOL agents and devise coordination structures appropriate for their tasks. Figure 7 shows the conversation plan 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 figures 2 and 3). Once Logistics receives the order, it tries to decompose it into activities like manufacturing, assembly, transportation, etc. This is done by running an external constraint based logistics scheduler inside a rule attached on the `order-received` state. If this decomposition is not possible, the process ends. If the decomposition is successful, the conversation goes to state `order-decomposed`. Here, Logistics matches the resulted activities with the capabilities of the existing agents, trying to produce a ranked list of contractors that could perform the activities.

If this fails, it will try to negotiate a slightly different contract that could be executed with the available contractors (state `alternative-needed`). If ranking succeeds, Logistics tries to form a 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 that are in principle interested 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

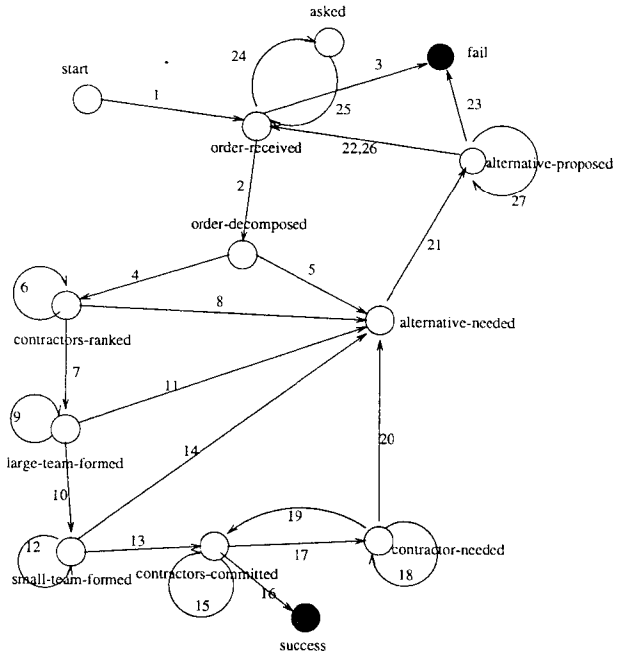

Figure 7. Logistics execution conversation plan

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. When forming the small team, Logistics similarly discusses 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.

In the `small-team-formed` state we continue with other newly spawned conversations with the team members to kick off execution. After having started execution, 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 can not be found, Logistics tries to negotiate an alternative contract (`alternative-needed`) with the Customer. To

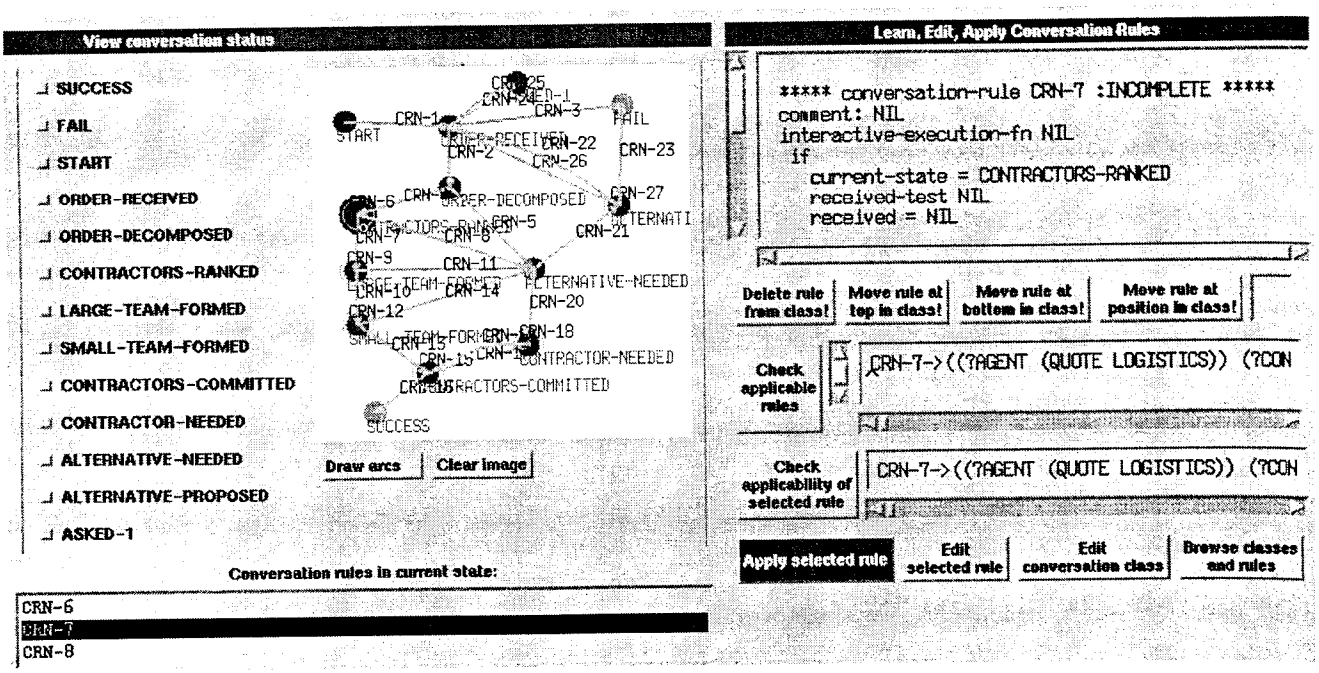

Figure 6. Inspecting, editing and applying rules.

do that. Logistics relaxes various constraints in the initial order (like dates, costs, amounts) and uses its scheduling tool to estimate fesability. 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 conversation goes back to the `order-received` state and resumes execution as illustrated.

The typical execution of the above coordination structure has one or more initial iterations during which things go as planned and agents finish work succesfully. Then, some contractors begin to lack the capacity required to take new orders (again this is determined by the local scheduling engine that considers the accumulated load of activities) and reject Logistics' proposal. In this case, Logistics tries to relax some constraints in the order (e.g. extend the due date to allow contractors to use capacity that becomes available later on). If the Customer accepts that (after negotiation) then the new (relaxed) order is processed and eventually succeeds. We usually run the system with 5-8 agents and 40-60 concurrent conversations. The COOL specification has about 12 conversation plans and 200 rules and utility functions. The Scheduler is an external process used by agents through an API. All this takes less than 2600 lines of COOL code to describe. We remark the conciseness of the COOL representation given the complexity of the interactions and the fact that the size of the COOL code does not depend on the actual number of agents and conversations, showing the flexibility and adaptability of the representation.

6. Conclusions

We believe the major contribution of this work is advancing a complete language design, including high level objects and control structures, of a practical application independent language for describing and carrying out coordination in multi-agent settings. Previous theoretical work investigating related [11, 9] state based representations has not consolidated the theoretical notions into usable language constructs, making it hard to use their ideas into applications. Related previous practical work [6, 10, 7, 12] has not produced truly generic, application independent language structures and constructs, making it difficult to reuse their experience. Agent oriented programming [13] is a notable exception. Being able to consolidate generic concepts and constructs into a language guarantees that developers of multi-agent systems will be able to reuse coordination structures and will be supported in building their own by the high level notions embodied in the language. As another contribution, we believe that recent approaches to agent communication like KQML [3], by focusing exclusively on generic vocabularies of communicative actions, have neglected the planning and execution dimension of the coordination task, requiring users to implement it from scratch. With a language

like COOL, these aspects are well supported and the expresiveness of KQML communicative actions can be taken advantage of. Finally, the language provides the representational foundation for tackling the important problem of acquiring dynamically emerging coordination knowledge. We also report on this aspect in [2].

The coordination language has been now evaluated on several problems, ranging from well-known test problems like n-queens to the supply chain of our TOVE virtual enterprise [4] and to supply chain coordination projects carried out in cooperation with industry. In all situations, the coordination language enabled us to quickly prototype the system and build running versions demonstrating the required behavior. Often, an initial (incomplete) version of the system has been built in a few hours enabling us to immediately demonstrate its functionality. We have built models containing hundreds of conversation rules and tens of conversation plans in several days. Moreover, we have found the approach explainable to industrial engineers interested in modeling manufacturing processes.

Our major priority at the moment continues to be gathering empirical evidence for the adequacy of the approach to industrial applications and for that matter we are jointly working with several industries. In one project for example, we are using the system to produce hard data characterizing how various coordination schemes affect the responsiveness and robustness of supply chains.

Since our approach is in an essential way managing workflow, we have also started addressing organizational workflow modeling and enactment. Last but not least, explaining the decisions and behavior of multi-agent systems will become more and more important as we move into more complex applications. Having explicit representations of coordination mechanisms forms the basis for providing explanations and we are studying the issue as part of another joint effort with industry.

7. 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 and M.S. Fox. COOL: A Language for Describing Coordination in Multi-Agent Systems. In Proceedings of the First International Conference on Multi-Agent Systems(ICMAS-95), pp 17-24, San Francisco, CA, june 1995.

[2] M. Barbuceanu and M.S.Fox. Capturing and Modeling Coordination Knowledge for Multi-Agent Systems. To appear in International Journal on Intelligent and Cooperative Information Systems, 1996.

[3] T. Finin et al. Specification of the KQML Agent Communication Language. The DARPA Knowledge Sharing Initiative, External Interfaces Working Group, 1992.

[4] M. S. Fox. A Common-Sense Model of the Enterprise. In Proceedings of Industrial Engineering Research Conference, 1993.

[5] N. R. Jennings. Towards a Cooperation Knowledge Level for Collaborative Problem Solving. In Proceedings 10-th European Conference on AI, Vienna, Austria, pp 224-228, 1992.

[6] N. R. Jennings. Controlling Cooperative Problem Solving in Industrial Multi-Agent Systems Using Joint Intentions. Artificial Intelligence, 75 (2) pp 195-240, 1995.

[7] S. M. Kaplan, W.J. Tolone, D.P. Bogia, C. Bignoli. Flexible, Active Support for Collaborative Work with ConversationBuilder. In CSCW 92 Proceedings, pp378-385, 1992.

[8] T. W. Malone and K. Crowston. Toward an Interdisciplinary Theory of Coordination. Center for Coordination Science Technical Report 120, MIT Sloan School, 1991

[9] F. vonMartial. Coordinating Plans of Autonomous Agents, Lecture Notes in Artificial Intelligence 610, Springer Verlag Berlin Heidelberg, 1992.

[10] R. Medina-Mora, T. Winograd, R. Flores, F. Flores. The Action Workflow Approach to Workflow Management Technology. In CSCW 92 Proceedings, pp 281-288, 1992.

[11] S. R. Rosenschein and L. P. Kaebling. A Situated View of Representation and Control. Artificial Intelligence 73 (1-2) pp 149-173, 1995.

[12] A. Shepherd, N. Mayer, A. Kuchinsky. Strudel - An Extensible Electronic Conversation Toolkit. In CSCW 90 Proceedings, pp 93-104, 1990.

[13] Y. Shoham. Agent-Oriented Programming. Artificial Intelligence 60, pp 51-92, 1993.