Integrating Communicative Action, Conversations and Decision Theory to Coordinate Agents

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

Abstract

The coordination problem in multi-agent systems is the problem of managing dependencies between the activities of autonomous agents, in conditions of incomplete knowledge about the dynamically changing environment and about the actions, reactions and goals of the agents populating it, such that to achieve the individual and shared goals of the participants and a level of coherence in the behavior of the system as a whole. The paper articulates a precise conceptual model of coordination as structured "conversations" involving communicative actions, amongst agents. The model is extended to a complete language design that provides objects and control structures that substantiate its concepts and allow the construction of real multi-agent systems in industrial domains. To account for the uncertainty of the environment and to capture user's prefernces about the possible actions we integrate decision theoretic elements based on Markov Decision Processes. Finally, to support incremental, in context acquisition and debugging of coordination knowledge we provide an extension of the basic representation and a visual tool allowing users to capture coordination knowledge as it dynamically emerges from the actual interactions. The language has been fully implemented and successfully used in several industrial applications, the most important being the integration of multi-agent supply chains for manufacturing enterprises. This application is used throughout the paper to illustrate the introduced concepts and language constructs.

Introduction

Coordination has been defined as the process of *managing dependencies between activities* (Malone & Crowston 91). 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 adhere to the view that the coordination problem can be tackled by recognizing and explicitely representing the *knowledge about the interaction processes* taking place among agents. Jennings (Jennings 92) 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 also believe that principles and theories must be put to work in real applications, and a major and often neglected way of doing this is by consolidating them into usable languages and tools.

Our contribution in this sense is the articulation of a model of "agent interactions" as knowledge driven structured conversations and its consolidation into a practical language design and implementation. The language, named COOL (from COOrdination Language), has been used in several industrial multi-agent systems, the most important of which is suppply chain integration, thoroughly used in this paper to illustrate the concepts and constructs of our system.

Assumptions and Basic Ideas

While coordination can be defined as above, without making assumptions about the ways to achieve it, building a practical language for representing coordination can not be done without clearly stating such assumptions as its foundation. The assumptions on which our language is built are as follows.

1. Autonomous agents have their own plans according to which they pursue their goals.
2. Being aware of the multi-agent environment they are in, agents plans explicitely represent interactions with other agents. Without loss of generality, we assume that this interaction takes place by exchanging messages.
3. Agents can not predict the exact behavior of other agents, but they can delimitate classes of alternative behaviors that can be expected. As a consequence, agents plans are conditional over possible actions/reactions of other agents.
4. Agents plans may be incomplete or inaccurate and the knowledge to extend or correct them may become available only during execution. For this reason, agents are able to extend and modify their plans during execution.

The most important construct of the language is the *conversation plan*. Conversation plans specify states and associated rules that receive messages, check local conditions, transmit messages and update the local status. COOL agents posess several conversation plans which they instantiate to drive interactions with other agents. Instances of conversation plans, called *conversations*, hold the state of execution with respect to the plan. Agents can have several active conversations in the same time and control mechanisms are provided that allow agents to suspend conversations while waiting for others to reach certain stages and to dynamically create conversation hierarchies in which child conversations are delegated issues by their parents and parents will handle situations that children are not prepared for. Conversation plans represent uncertainty by associating probabilities to the actions represented by rules. Users preferences for various states and actions are represented as rewards associated with actions and states. The theory of Markov Decision Processes (Bellman 57; Puterman 94) is used to determine the optimal actions to execute in order to maximize the expected accumulated rewards of conversation plans.

Multi-agent systems built with this language operate on the assumption of *mutual comprehensibility*. This means that they are designed in such a way that, normally, an agent can retrieve a conversation or a conversation plan that handles a message received from another agent. This guarantees that, normally, conversations would not get stuck because agents can not understand a message. This assumption is weaker than the assumption of cooperative systems, because it does not presuppose any intentional stance of the agents. On the other hand, we are aware of the limitations of this assumption and we provide mechanisms that allow agents to continue even when mutual comprehensibility is not satisfied. These come as recovery rules (which can modify the execution status or the plan) and much more important, as support for direct, in context, user guidance which is used for debugging and knowledge acquisition.

Integrating the Supply Chain

The supply chain of a modern "virtual" 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: 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 supply chain, our ability 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

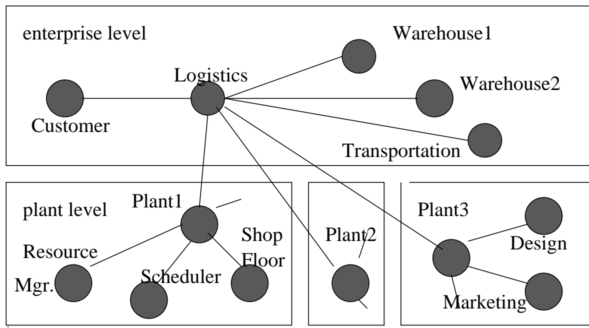


Figure 1: Multi-level supply chain.

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.

The Coordination Language

Communication

COOL has a communication component that uses an extended version of the KQML language (Finin et al 92). 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. This makes our approach practically independent of KQML (any message language with communicative actions would do), although a standard would be a marked advantage. The following example illustrates the form of extended KQML we are working with.

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

Agents and Environments

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, specifying the

```
(def-conversation-plan
  '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.

conversation plan for its *initial conversation* and specifying the variables that form its local persistent data base:

```
(def-agent 'customer
 :initial-conversation-plan
   'initial-conversation-plan).
```

When an agent is created, its initial conversation starts running and while it runs, the agent is "alive". Agents are run as lightweight processes inside *environments* that execute on local or remote sites. TCP/IP is used at the transport level.

Conversations

Conversation plans are rule based descriptions of how an agent acts in certain situations. COOL provides ways to associate conversation plans to agents, thus defining what sorts of interactions each agent can handle. A conversation plan 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 values of a special variable, the *current-state*. Because of that, conversation plans and actual conversations admit a graph representation where nodes represent states and arcs transitions amongst states.

Figure 2 shows the conversation plan governing the Customer's conversation with Logistics in our supply chain application. Figure 3 shows the associated graph of this conversation plan. Arcs indicate the existence of rules that will move the conversation from one state to another.

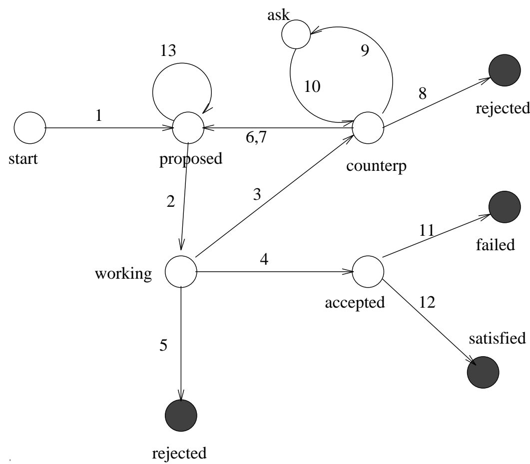


Figure 3: Graph representation of customer-conversation.

Error recovery rules are another component of conversation plans (not illustrated in figure 2). They specify how incompatibilities among the state of a conversation and the incoming messages are handled. Such incompatibilities can be caused by both planning and execution flaws. Error recovery rules are applied when conversation rules can not handle the current situation. They can address the problem either by modifying the execution state - e.g. by discarding inputs, changing the conversation current-state or just reporting an error - or by executing new plans or modifying the current one - e.g. initiating a new clarification conversation with the interlocutor.

Actual conversations instantiate conversation plans 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 plan describes an interaction from the viewpoint of an individual agent (in figure 2 the Customer). For two or several agents to "talk", the executed conversation plans of each agent must generate sequences of messages that the others' conversation plans can process (according to the mutual comprehensibility assumption). Thus, agents that carry out an actual conversation `C` actually instantiate different conversation plans internally. These instances will have unique names (e.g. `Customer-C`) inside each agent, allowing the system to direct messages appropriately.

Conversation Rules

Conversation rules describe the actions that can be performed when the conversation is in a given state.

```
(def-conversation-rule 'lep-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 '(update-var ?conv '?order ?message))
```

Figure 4: Conversation rule.

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 plan 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.

Our typology of rules also includes *timeout*, *on-entry* and *on-exit* rules. Timeout rules have a `:timeout` slot filled with a value representing a number of time units. These rules are tried after the specified number of time units has passed after entering the current state. Such rules enable agents to operate in real time, for example by controlling the time spent waiting for a message or by ensuring actions are executed at well determined time points. On-entry and on-exit rules are always executed when a conversation enters (exits) a state. They are useful for both mundane things like set-ups, clean-

```
(def-conversation-rule 'icc1-1
 :current-state 'process
 :such-that '(exists-runnable-or-waiting
                              ?agent ?conv)
 :next-state 'process
 :do '(progn
       (move-msgs-to-addressee-conv
                      ?conv ?runnable)
       (execute-conversation ?runnable)))
```

Figure 5: Conversation rule of the initial conversation.

ups or instrumentations and non-mundane activities like strategic reasoning, as illustrated in a next section.

The Initial Conversation

When an agent is created, its initial conversation starts running. As long as this conversation is not terminated, the agent is alive and active. All incoming messages are dispatched by the initial conversation. Sometimes they are dispatched to existing conversations, sometimes new conversations are created to handle them (for example we define an `:intent` slot of messages to help identify the conversation plans that can handle messages with given intents). The initial conversation is the ancestor of any conversation in the system. As new conversations are created, they can later create their own child conversations, incrementally building trees of conversations. The message dispatch mechanism allows direct dispatch to known conversations, or various forms of top-down or bottom-up forwarding of the message (possibly with annotations added along the way) to several conversations. This can emulate Brooks-like or hierarchical architectures. Figure 5 illustrates one rule from one initial conversation plan. This rule checks if there exists a conversation (immediately) runnable or waiting for messages and, if so, forwards it its messages and then executes it.

Synchronized Conversation Execution

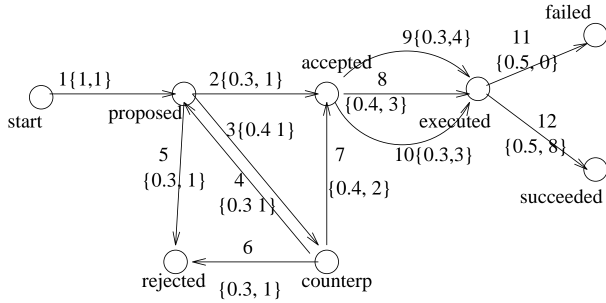
Normally, a conversation may spawn another one and they will continue in parallel. When we need to synchronize their execution, we can do that by freezing the execution of one conversation until several others reach certain states. This is important in situations where an agent can not continue along one path of interaction unless some conditions are achieved. In such cases, the conversation that can not be continued is suspended, the conversations that can bring about the desired state of affairs are created or continued, and the system ensures that the suspended conversation will be resumed as soon as the condition it is waiting for becomes true. The specification of this condition is as an arbitrary predicate over the state of other conversations.

Integrating Decision Theoretic Planning

Decision theoretic planning integrates probabilities and utilities in the planning process, with the goal of producing plans that explicitely consider environment uncertainity and user preferences, guaranteeing certain classes of optimal behavior. The basic observation is that conversations as described in COOL can be clearly mapped to fully-observable, discrete-state Markov decision processes (MDP) (Bellman 57; Puterman 94). In this mapping, COOL states become MDP states (always finite) and conversation rules become MDP actions (again finite) that generate state transitions when executed. Let S be the set of states and A the set of actions of a COOL conversation plan viewed as an MDP. We extend our representation of conversation plans and rules as follows. First, we define for each action (rule) $a \in A$ the probability $P(s,a,t)$ that action a causes a transition to state t when applied in state s. In our framework, this probability quantifies the likelihood of the rule being applicable in state s and that of its execution being successful. Second, we define for each action (rule) the reward (a real number) denoting the immediate utility of going from state s to state t by executing action a, $R(s,a,t)$. (Note that a COOL rule can perform a transition only from one given state to another, which simplifies the computations described bellow). Since COOL conversation plans are ment to operate for indefinite periods of time, we use the theory of infinite horizon MDP-s. A (stationary) policy $\pi : s \rightarrow A$ describes the actions to be taken by the agent in each state. We assume that an agent accumulates the rewards associated with each transition it executes. To compare policies, we use the *expected total discounted reward* as the criterion to optimize. This criterion discounts future rewards by rate $0 \leq \beta < 1$. For any state s, the value of a policy π is defined as:

$$V_\pi(s) = R(s,\pi(s),t) + \beta \sum_{t \in S} P(s,\pi(s),t) V_\pi(t)$$

The value of π at any state s can be computed by solving this system of linear equations. A policy π is optimal if $V_\pi(s) \geq V_{\pi'}(s)$ for all $s \in S$ and all policies π'. A simple algorithm for constructing the optimal policy for a given reward structure is value iteration (Bellman 57). This is an iterative algorithm guaranteed to converge under the assumptions of infinite horizon discounted reward MDP-s. Value iteration produces sequences of *n-step optimal value*



Ordering produced by value iteration: proposed: 2,3,5 accepted: 9, 8, 10
counterp: 7,4,6 executed: 12, 11

Figure 6: Using value iteration to reorder rules.

functions V^n by starting with an arbitrary value for V^0 and computing

$$V^{i+1}(s) = max_{a \in A}\{R(s,a,t) + \beta \sum_{t \in S} P(s,a,t) V^i(t)\}$$

The values V^i converge linearly to the optimal value V^*. After a finite number n of iterations, the chosen action for each state forms an optimal policy π and V^n approximates its value. To stop the iteration, an often used criterion requires termination when

$$|V^{i+1} - V^i| \leq \epsilon(1-\beta)/2\beta$$

This ensures that V^{i+1} is within ϵ of the optimal function V^* at any state.

The application of this theory to conversation plans is illustrated in figure 6. With each rule number we show the probability and the reward associated to the rule. We use the value iteration technique to actually order the rules in a state rather than just computing the best one. This is needed because of the conditional nature of actions in COOL. The result of this is the reordering of rules in each state according to how close they are to the optimal policy. Since COOL tries the rules in the order they are encountered, the optimal reordering guarantees that the system will always try the optimal behavior first. Of course, there are several reward structures corresponding to different criteria, like *cost* or *time*. To account for these, we actually produce a separate ordering for each criterion. Then a weighted combination of criteria is used to produce the final ordering. In COOL we use *on-exit* rules to dynamically estimate how well the system has done with respect to the various criteria. If, for example, we have spent too much *time* in the current plan, these rules will notice that. When entering a new state, *on-entry* rules look at the criteria that are under-achieved and compute a new global criterion that corrects that (e.g. giving *time* a greater weight). This new criterion is used to dynamically reorder the rules in the current state. In this way we achieve adaptive behavior of the agent.

In Context Acquisition and Debugging of Coordination Knowledge

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 system does not try to match the rule itself. Since the action part may be incomplete, the system can not apply the rule either. All that can be done is to let the user handle the situation. Interaction specifications may contain both complete and incomplete rules in the same time. 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 trig-

```
(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 7: Incomplete conversation rule.

gered, with the user in control over what to do in the respective situation, as explained further on.

Figure 7 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 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 its graphical interface. To view the status of the conversation at the time an incomplete rule was encountered, the acquisition service shows the finite state abstraction (like in figure 8). Here we have an instance of the logistics execution process as seen by the Logistics agent. A textual presentation of the conversation and a browser for the conversation variables are also available.

Another aspect of the conversation context is formed by the available rules. This is also shown in figure 8. 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 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 plan. It is also possible to invoke the rule editor, the conversation plan editor or the browser for plans and rules allowing the user to inspect other plans 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. Other services include presentation and browsing of the conversation history and interactive, stepwise modification and execution of rule actions. The modifications performed to the action part can be saved into a new rule that can be "learned" by the system.

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 9 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 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 sus-

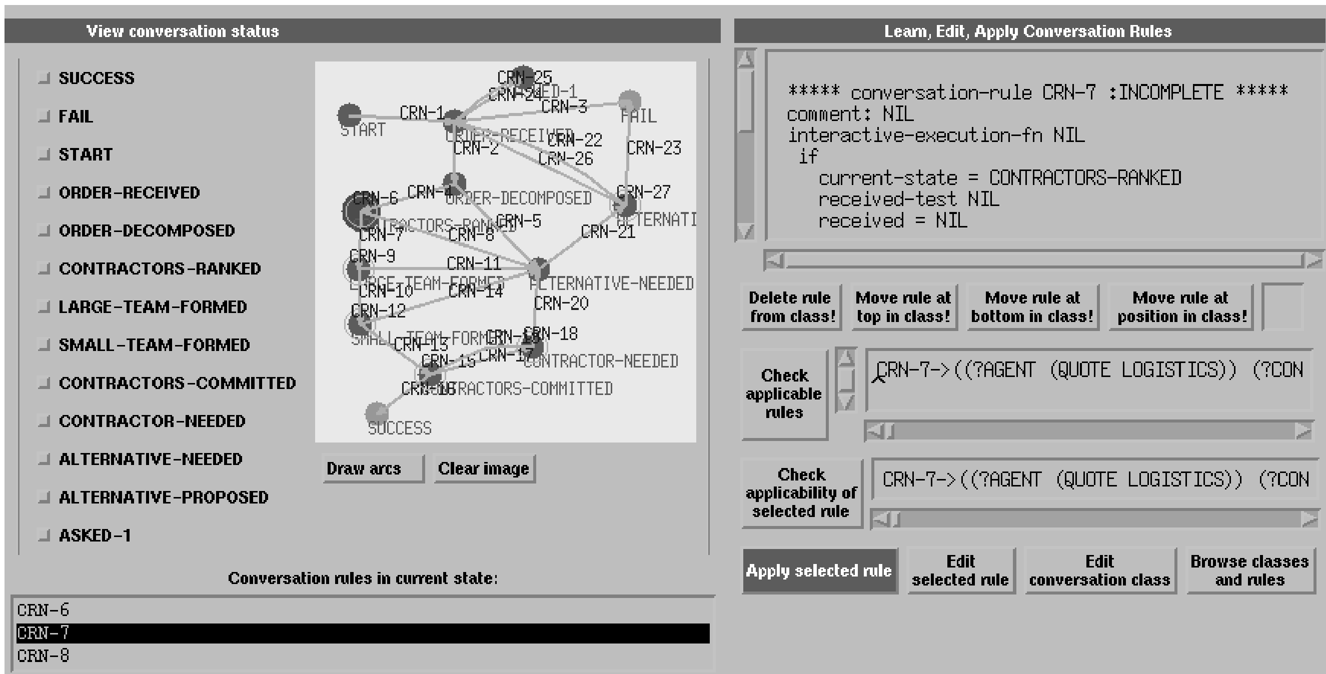


Figure 8: Inspecting, editing and applying rules.

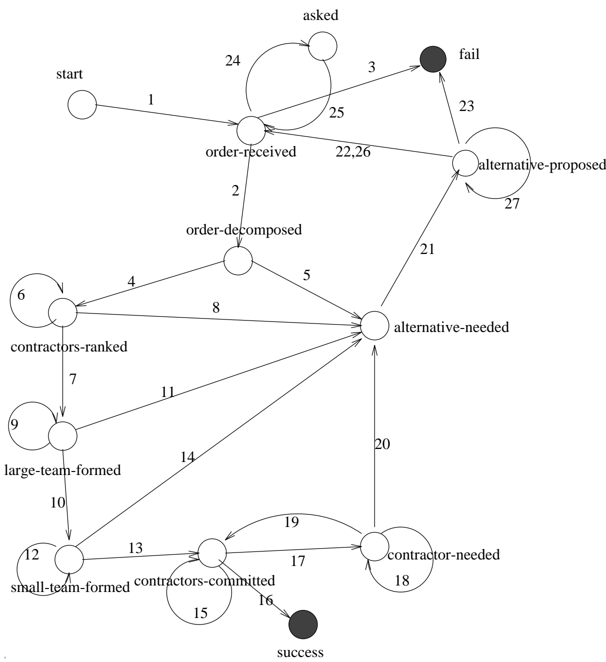


Figure 9: Logistics execution conversation plan

pending 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 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 so-

lution 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 may eventually succeed. The reward structures used give preference to accomplishing work and commitments above anything else, but prefers quick rejections to long negotiations that terminate with rejections. Least preferred is failure of committed work. 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.

Conclusions

We believe the major contribution of this work is advancing a complete language design and an associated programming system for a practical, application independent language for describing and carrying out coordination in multi-agent settings. Previous theoretical work has investigated related state based representations (Rosenschein & Kaebling 95; vonMartial 92) but has not consolidated the theoretical notions into usable language constructs, making it hard to use their ideas into applications. Various formalizations of mental state notions related to agency (Cohen & Levesque 90; Cohen & Levesque 91; Levesque, Cohen & Nunes 90) have provided semantic models that clarify a number of issues, but operate under limiting assumptions that similarly make practical use and consolidation difficult. The work of (Jennings 95; Jennings 92) provided part of the initial motivation for our approach to coordination as a domain of knowledge to be explicitely represented and instrumented. Some conversational concepts have been used by (Kaplan et al 92; Shepherd, Mayer & Kuchinsky 90; Medina-Mora et al 92) in the context of collaborative and workflow applications. We have extended and modified them for use in multi-agent settings and added things like knowledge acquisition, decision theoretic optimization and sophisticated control that led to a more generic, application independent language. Agent oriented programming (Shoham 93) is also related to our work as it similarly uses communicative action, rules and agent representations. Our language differs from AOP in the explicit provision of the conversation notion, the more powerful control structures that emerge from it, the use of decision theretic ideas and the more powerful programming environment including the essential support for knowledge acquisition.

With respect to our own previously reported work on COOL (Barbuceanu & Fox 95), this paper presents important advances related to the decision theoretic elements, the knowledge acquisition component and the industrial application to supply chain management.

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 (Finin et al 92), 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 (which, we repeat, is in fact independent of KQML), these aspects are well supported and the expresiveness of KQML communicative actions can be taken advantage of. With the support for decision theoretic elements, the language explicitely takes into consideration users' probabilities and preferences, guaranteeing a certain notion of optimal behavior w.r.t these quantifications. 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 (Barbuceanu & Fox 96).

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 (Fox 93) 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.

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

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.

M. Barbuceanu and M.S. Fox. Capturing and Modeling Coordination Knowledge for Multi-Agent System. To appear in the International Journal of Intelligent and Cooperative Information Systems, 1996.

Richard E. Bellman. Dynamic Programming. Princeton University Press, Princeton 1957.

P. R. Cohen and H. Levesque. Intention is Choice with Commitment. Artificial Intelligence 42, pp 213-261, 1990.

P. R. Cohen and H. Levesque. Teamwork. Nous 15, pp 487-512, 1991.

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

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

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.

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

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.

H. J. Levesque, P. R. Cohen and J. H. Nunes. On Acting Together. In Proceedings of 8-th National Conference on AI, Boston, pp 94-99, 1990.

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

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

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.

Martin L. Puterman. Markov Decision Processes: Discrete Stochastic Dynamic Programming. Willey, New York, 1994.

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

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

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