
The International Journal of Flexible Manufacturing Systems, 12 (2000): 165–188

Agent-Oriented Supply-Chain Management

MARK S. FOX, MIHAI BARBUCEANU, AND RUNE TEIGEN
Enterprise Integration Laboratory University of Toronto, Ontario, Canada

Abstract. The supply chain is a worldwide network of suppliers, factories, warehouses, distribution centers, and retailers through which raw materials are acquired, transformed, and delivered to customers. In recent years, a new software architecture for managing the supply chain at the tactical and operational levels has emerged. It views the supply chain as composed of a set of intelligent software agents, each responsible for one or more activities in the supply chain and each interacting with other agents in the planning and execution of their responsibilities. This paper investigates issues and presents solutions for the construction of such an agent-oriented software architecture. The approach relies on the use of an agent building shell, providing generic, reusable, and guaranteed components and services for communicative-act-based communication, conversational coordination, role-based organization modeling, and others. Using these components, we show two nontrivial agent-based supply-chain architectures able to support complex cooperative work and the management of perturbation caused by stochastic events in the supply chain.

Key Words: software agents, coordination, interaction, supply chain

1. Introduction

The supply chain is a worldwide network of suppliers, factories, warehouses, distribution centers, and retailers through which raw materials are acquired, transformed, and delivered to customers. Supply-chain management is the strategic, tactical, and operational decision making that optimizes supply-chain performance. The strategic level defines the supply chain network; that is, the selection of suppliers, transportation routes, manufacturing facilities, production levels, warehouses, and the like. The tactical level plans and schedules the supply chain to meet actual demand. The operational level executes plans. Tactical- and operational-level decision-making functions are distributed across the supply chain.

To optimize performance, supply-chain functions must operate in a coordinated manner. But the dynamics of the enterprise and the market make this difficult: Materials do not arrive on time, production facilities fail, workers are ill, customers change or cancel orders, and so forth, causing deviations from the plan. In some cases, these events may be dealt with locally; that is, they lie within the scope of a single supply-chain function. In other cases, the problem cannot be "locally contained" and modifications across many functions are required. Consequently, the supply-chain management system must coordinate the revision of plans or schedules across supply-chain functions. The ability to manage the tactical and operational levels of the supply chain so that the timely dissemination of information, accurate coordination of decisions, and management of actions among people and systems is achieved ultimately determines the efficient, coordinated achievement of enterprise goals.

In recent years, a new software architecture for managing the supply chain at the tactical and operational levels has emerged. It views the supply chain as composed of a set of intelligent (software) agents, each responsible for one or more activities in the supply chain and each interacting with other agents in planning and executing their responsibilities. An agent is an autonomous, goal-oriented software process that operates asynchronously, communicating and coordinating with other agents as needed.

This paper investigates issues and solutions in the construction of such a software architecture. Section 2 reviews a number of issues and presents a list of requirements for agent-oriented architectures for the supply chain. Section 3 presents our Agent Building Shell, which provides generic, reusable, and guaranteed components for some of the required elements of the architecture. Section 4 shows how the components provided by the shell have been used to construct nontrivial agent-oriented supply chain architectures and evaluates the solutions advanced. We end with concluding remarks and future work hints.

2. Design issues for a multiagent supply-chain system

Which are the most important issues to address, to effectively build an agent-based software architecture for the supply chain? The first issue we face is deciding how supply-chain activities should be distributed across the agents. Existing decompositions, as found in MRP (Material Resource Planning) systems, arose out of organizational constraints, legacy systems, and limitations on algorithms. For example, the distinction between master production scheduling and detailed scheduling is due primarily to algorithm limitations. The merging of these two functions and the inclusion of some activities found in inventory management and activity planning becomes possible with the availability of more sophisticated planning and scheduling algorithms. With more sophisticated planning, scheduling, and coordination methods, we can build better decompositions, improving the overall quality of supply-chain management. For illustration, here is a typical agent decomposition that we use in our work:

- *Order acquisition agent*. This agent is responsible for acquiring orders from customers; negotiating with customers about prices, due dates, and the like; and handling customer requests for modifying or canceling their orders. When a customer order is changed, that change is communicated to the logistics agent. When plans violate constraints imposed by the customer (such as due date violation), the order acquisition agent negotiates with the customer and the logistics agent for a feasible plan.
- *Logistics agent*. This agent is responsible for coordinating the plants, suppliers, and distribution centers in the enterprise domain to achieve the best possible results in terms of the goals of the supply chain, including on-time delivery, cost minimization, and so forth. It manages the movement of products or materials across the supply chain from the supplier of raw materials to the customer of finished goods.
- *Transportation agent*. This agent is responsible for the assignment and scheduling of transportation resources to satisfy interplant movement requests specified by the logistics agent. It can consider a variety of transportation assets and transportation routes in the construction of its schedules.

- *Scheduling agent*. This agent is responsible for scheduling and rescheduling activities in the factory, exploring hypothetical "what-if" scenarios for potential new orders, and generating schedules that are sent to the dispatching agent for execution. It assigns resources and start times to activities that are feasible while at the same time optimizing certain criteria such as minimizing work in progress or tardiness. It can generate a schedule from scratch or repair an existing schedule that has violated some constraints. In anticipation of domain uncertainties like machine breakdowns or material unavailability, the agent may reduce the precision of a schedule by increasing the degrees of freedom in the schedule for the dispatcher to work with. For example, it may "temporally pad" a schedule by increasing an activity's duration or "resource pad" an operation by either providing a choice of more than one resource or increasing the capacity required so that more is available.
- *Resource agent*. The resource agent merges the functions of inventory management and purchasing. It dynamically manages the availability of resources so that the schedule can be executed. It estimates resource demand and determines resource order quantities. It is responsible for selecting suppliers that minimize costs and maximize delivery. This agent generates purchase orders and monitors the delivery of resources. When resources do not arrive as expected, it assists the scheduler in exploring alternatives to the schedule by generating alternative resource plans.
- *Dispatching agent*. This agent performs the order release and real-time floor control functions as directed by the scheduling agent. It operates autonomously as long as the factory performs within the constraints specified by the scheduling agent. When deviations from schedule occur, the dispatching agent communicates them to the scheduling agent for repair. Given degrees of freedom in the schedule, the dispatcher makes decisions as to what to do next. In deciding what to do next, the dispatcher must balance the cost of performing the activities, the amount of time in performing the activities, and the uncertainty of the factory floor. For example, (1) given that the scheduler specified a time interval for the start time of a task, the dispatcher has the option of either starting the task as soon as possible (just in case) or as late as possible ("just in time"); (2) given that the scheduler did not specify a particular machine for performing the task, the dispatcher may use the most "cost-effective" machine (minimize costs) or the "fastest" machine (minimize processing time).

The second issue is coordination among components. The dynamics of the supply chain makes coordinated behavior an important factor in its integration. To optimize supply-chain decisions, an agent cannot by itself just make a locally optimal decision but must determine the effect its decisions will have on other agents and coordinate with others to choose and execute an alternative that is optimal over the entire supply chain. The problem is exacerbated by the stochastic events generated by the flow of new objects into the supply chain. These include customer orders, new customers, shipments of raw material from suppliers, and new suppliers themselves. Modifications to customer orders (at the customer's request), resource unavailabilities from suppliers, and machine breakdown all drive the system away from any existing predictive schedule. In dealing with stochastic events, the agents must make optimal decisions based on complex global criteria that (1) are not completely known by any one agent and (2) may be contradictory and therefore require trade-offs.

Agents operate within organizations where humans must be recognized as privileged members. This requires knowledge of organization roles and respecting the obligations and authority incurred by the roles. Coordination and negotiation must take these issues into consideration as well, in addition to the computational cost, complexity, and accuracy of the algorithms used in optimization.

Given the dynamics of the supply chain resulting from unplanned for (stochastic) events such as transportation problems or supply problems, what nature of interaction among agents will reduce change-induced perturbations in a coordinated manner? If each agent has more than one way to respond to an event, how do they cooperate in creating a mutually acceptable solution? In other words, how do agents influence or constrain each other's problem-solving behavior?

For two or more agents to cooperate, a "cultural assumption" must exist. The cultural assumption indicates what an agent can expect in terms of another agent's behavior in a problem-solving situation. A possible cultural assumption is that agents are "constraint-based problem solvers." That is, given a set of goals and constraints, they search for a solution that optimizes the goals and satisfies the constraints. Another cultural assumption could be that agents can generate more than one solution, thereby enabling the consideration of alternatives and trade-offs by a set of cooperating agents. A third cultural assumption is that agents have the ability and authority to relax a subset of constraints if the global solution is further optimized.

The third issue is responsiveness. In a dynamic environment, the time available to respond may vary based on the event. An agent must be able to respond within the time allotted. Algorithms that can generate solutions no matter how much time is available are known as *anytime algorithms*. The quality of the solution of anytime algorithms usually is directly related to the time available.

The fourth issue is the availability of knowledge encapsulated within a module. In conventional MRP systems, a module is designed to perform a specific task. The modules may contain certain knowledge (used in the performance of each task) that could be used to answer related questions. Our goal is to "open up" a module's knowledge so that it can be used to answer questions beyond those originally intended.

In summary, the next generation supply chain management system will be all of the following:

1. *Distributed*. The functions of supply chain management are divided among a set of separate, asynchronous software agents.
2. *Dynamic*. Each agent performs its functions asynchronously as required, as opposed to in a batch or periodic mode.
3. *Intelligent*. Each agent is an "expert" in its function. It uses artificial intelligence and operations research problem-solving methods.
4. *Integrated*. Each agent is aware of and can access the functional capabilities of other agents.
5. *Responsive*. Each agent is able to ask for information or a decision from another agent—each agent is both a client and a server.
6. *Reactive*. Each agent is able to respond to events as they occur, modifying its behavior as required, as opposed to responding in a preplanned, rigid, batch approach.

7. *Cooperative*. Each agent can cooperate with other agents in finding a solution to a problem; that is, they do not act independently.
8. *Interactive*. Each agent may work with people to solve a problem.
9. *Anytime*. No matter how much time is available, an agent can respond to a request, but the quality of the response is proportional to the time given to respond.
10. *Complete*. The total functionality of the agents must span the range of functions required to manage the supply chain.
11. *Reconfigurable*. The supply-chain management system itself must be adaptable and support the "relevant subset" of software agents. For example, a user who wants to schedule only a plant should not be required to use or have a logistics component.
12. *General*. Each agent must be adaptable to as broad a set of domains as possible.
13. *Adaptable*. Agents need to quickly adapt to the changing needs of the human organization. For example, adding a resource or changing inventory policy should be quick and easy for the user to do.
14. *Backwards compatible*. Agents need to have a seamless upgrade path so that the release of new or changed features does not compromise existing integration or functionality.

3. The Agent Building Shell

Given the complexity and difficulty of the issues just reviewed, how should we approach the construction of a software architecture able to address these concerns? Our answer is that many of these concerns can be addressed in generic and reusable ways by means of an Agent Building Shell (ABS) (Barbuceanu and Fox, 1996a). The ABS is a collection of reusable software components and interfaces, providing support for application-independent agent services. Using these services, developers can build on a high-level infrastructure, whose abstractions provide a conceptual framework that helps in designing and understanding agent systems; eliminate work duplication; and offer guarantees about the services provided by the tool.

Figure 1 shows the current architecture of the ABS. At the outermost layer, communication services allow agents to exchange messages composed from domain-independent communicative acts and domain-dependent content specifications. The next coordination level provides a full design of a coordination language (COOL) based on the conversation metaphor. Conversations can model peer-to-peer interaction in which autonomous agents make requests, volunteer information, react to events, update their state, and so on. Conversations express the shared conventions that stand at the basis of coordination (Jennings, 1993) and are used as the basic abstraction for capturing the coordination knowledge and social know-how discussed by Fox (1987) and Jennings (1992). The shell provides a full conversational ontology, containing conversation plans, conversation rules, and actual conversations, in terms of which complex interactions are described. An important extension deals with using decision-theoretic planning to make conversation more adaptive. Programming tools supporting conversational interaction also are provided, the most important being a tool for dynamic, on-line acquisition of conversation plans. The next layer of the shell deals with generic models of action and behavior, the representation of obligations and interdictions derived from the authority of agents in the organization, the use of obligations

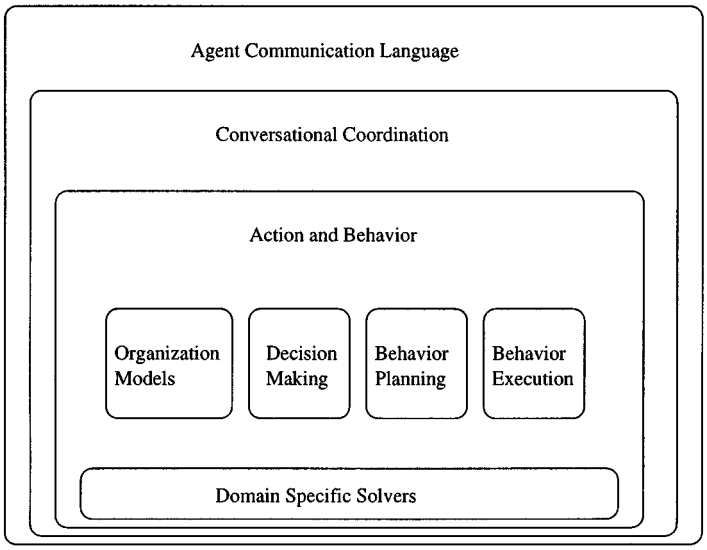

Figure 1. The agent building shell.

in negotiation, and the ways in which agents plan their behavior and execute the planned activities. This is on-going work that will not be addressed in this paper. Published accounts of this work include Barbuceanu (1998) and Barbuceanu, Gray, and Mankovski (1998).

3.1. Conversational interaction language

Communication. We support speech-act-based interagent communication in the style of KQML (Knowledge Query and Manipulation Language) (Finin, Labrou and Mayfield, 1995). Speech acts, as first discussed by Austin (1962) and Searle (1969) or in our context communicative acts (Cohen and Levesque, 1995), have an important advantage of generality and independence across domains and applications, and their relation to human dialogue and discourse makes agent communication understandable and opens the way to better integration of the human user as a privileged member of the agent community. The following example illustrates the message format supported by the service:

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

Here, `propose` is the message type (denoting the speech act), `:sender` and `:receiver` denote the agents involved in the message exchange, `:content` is the content of the message—we make no commitment as to the nature of the content language. (KIF (Knowledge Interchange Format) has been previously proposed as a logic-based content language and the Knowledge Sharing Effort of Patil et al., 1992, was the first proponent of an agent architecture combining KIF and KQML.) The `:conversation` slot gives the name of the conversation of which the message is part and `:intent` is used when initiating a conversation (this case) to indicate to the recipient the purpose of the conversation.

Conversation plans are rule-based descriptions of how an agent acts and reacts in certain situations. Our conversational language 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 database that maintains the state of the conversation. The database 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 among states.

The following is the conversation plan governing the customer's conversation with logistics in one supply-chain application:

```
(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 shows the associated graph of this conversation plan. Arcs indicate the existence of rules that will move the conversation from one state to another. As will became clear immediately, conversation plans are general plan specifications not restricted in any way to exclusively describing interactions among agents by message exchange. They can describe equally any local behavior of the agent that does not involve interaction with other agents. In our applications, we also use conversation plans to describe local decision making; for example, based on using local solvers (e.g. constraint based schedulers) or other decision-making tools available to agents.

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.

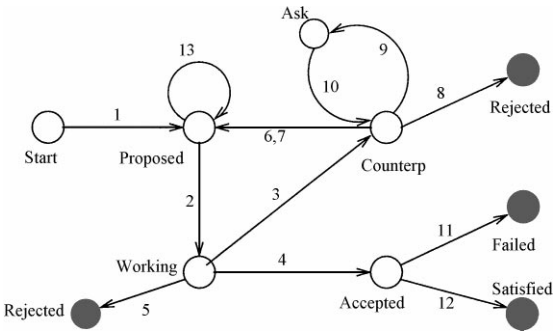

Figure 2. Graph representation of customer conversation.

Each conversation plan describes an interaction from the viewpoint of an individual agent (in the previous example, the customer). As such, an agent's conversation plan operates like a *transducer* (Rosenschein and Kaebling, 1995), converting streams of input messages into streams of output messages and actions. For two or several agents to "talk," the executed conversation plan of each agent must generate sequences of messages that the others' conversation plans can process (according to a mutual comprehensibility assumption). This raises the problem of how an agent that received the first message in a new conversation can select the appropriate conversation plan that will handle this and the next messages in the conversation. We adopt the convention that the first message in a new conversation has to have attached a specification of the purpose of the conversation (in the `:intent` slot). The receiver then will use this specification to find a conversation plan that can sustain a conversation with that purpose. Agents in this way can instantiate different conversation plans internally, without being aware of what plan each other has selected.

Conversation rules describe the actions that can be performed when the conversation is in a given state. In `customer-conversation`, for example, when the conversation is in the `working` state, only rules `cc-5`, `cc-4`, and `cc-3` 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 plugable functions that users can modify. The following is a conversation rule from the conversation plan that logistics uses when talking to the customer about orders:

```
(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)).
```

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.

Error recovery rules (not illustrated in the preceding example) specify how incompatibilities among the state of a conversation and the incoming messages are handled. Such incompatibilities can be caused by planning or execution flaws. Error recovery rules are applied when conversation rules cannot handle the current situation. They can address the problem either by modifying the execution state—such as by discarding inputs, changing the conversation current state, or just reporting an error—or by executing new plans or modifying the current one—such as initiating a new clarification conversation with the interlocutor. Our typology of rules also includes *timeout* rules. These rules are tried when a specified number of time units has passed since 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.

Synchronized conversation execution. Normally, one conversation may spawn another one, and they 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 cannot continue along one path of interaction unless some conditions are achieved. In such cases, the conversation that cannot 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.

Control architecture. Each agent operates in a loop where (1) events are sensed, like the arrival of messages expressing requests from other agents; (2) the current situation is evaluated, updating or creating new beliefs or adding new conversations to the agenda; (3) an agent selects an entry from the agenda. This is either a newly requested conversation, for which a conversation plan is retrieved and initiated, or one that is under processing, in which case its execution continues incrementally.

3.2. Integrating decision theoretic planning

The framework of Markov decision processes (MDP; introduced by Bellman, 1957, and recently reviewed by Puterman, 1994) has been integrated in the coordination system, producing conversation plans that explicitly consider environment uncertainty and user preferences and guarantee certain classes of optimal behavior. The main idea is that conversation plans can be mapped to fully observable, discrete-state Markov decision processes. In this mapping, conversation 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 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, for each action (rule), we define 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 rule can perform a transition only from one given state to another, which simplifies the computations that follow). Since conversation plans are meant 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

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

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, 1957). 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 functions* V^n by starting with an arbitrary value for V^0 and computing

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

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. We stop the iteration by checking that V^{i+1} is within an ϵ of the optimal function V^* at any state.

To use the MDP model in our coordination language, we first extend the representation of conversation plans to include the probabilities and rewards of different actions (rules) in the plan. The representation of rewards allows any number of criteria with their own reward values; for example, a rule can have reward 9 with respect to *time* (therefore, will execute quickly) and reward 1 with respect to the *quality* of the solution produced (hence, produce a poor-quality result).

An illustration of an extended conversation plan is shown in figure 3. With each rule number we show the probability and the reward (according to some criterion) associated to the rule. Second, we use the value iteration technique to actually *order* the rules in a state rather than just computing the best one. The result of this is the reordering of rules in each state according to how close they are to the optimal policy. Since rules are tried in the order they are encountered, this guarantees that the system always will try the optimal behavior first. As mentioned, the several reward structures correspond to different criteria. To account for these, we actually produce a separate ordering for each criterion. Then,

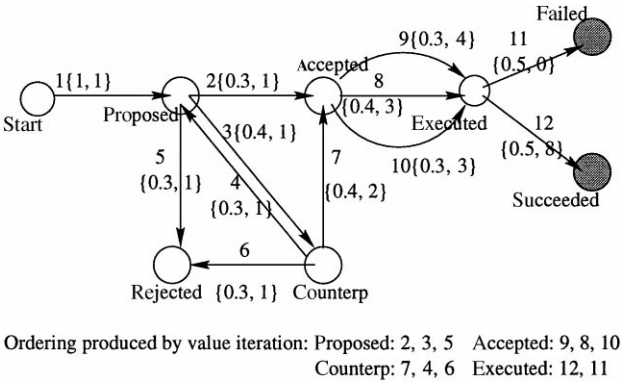

Figure 3. Using value iteration to reorder rules.

a linear combination of criteria (e.g., $w_1 time + w_2 quality$) is used to produce the final ordering. We use special 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 it. When entering a new state, these rules look at the criteria that are underachieved 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.

4. Supply-chain applications

This section talks about two supply chain applications. In the first, we design coordination structures to handle the dynamic formation and operation of teams. This example is relevant for the virtual enterprise approach to manufacturing. The second application captures a realistic supply chain and uses our approach to design and implement a number of coordination mechanisms that account for both the steady-state behavior and, more interestingly, the coordinated behaviors that can be applied to cope with unexpected events that perturbate the operation of the supply chain.

4.1. Coordinating teamwork in a virtual supply chain

The first example basically uses the layout we have been using to illustrate our language up to now. A logistics agent coordinates the work of several plants and transportation agents, while interacting with the customer in the process of negotiating the execution of an order. Figure 4 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`. Once logistics receives the order, it tries to decompose it into activities like manufacturing, assembly, transportation, and the like. This is done by running an external constraint based logistics scheduler

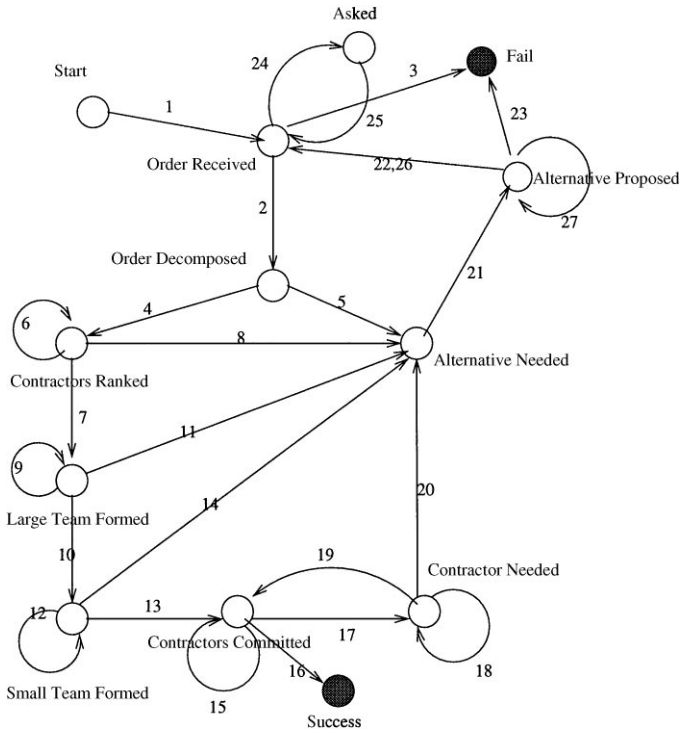

Figure 4. Logistics execution conversation plan.

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, logistics will try to negotiate a slightly different contract, which could be executed with the available contractors (state `alternative-needed`). If ranking succeeds, logistics will try 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, in principle, are interested in participating by executing the activity determined previously by logistics. Membership on the large team does not bind contractors to execute their activity, it only expresses their interest in doing the activity. If the large team is 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. When forming the small team, logistics similarly discusses with each member of the large

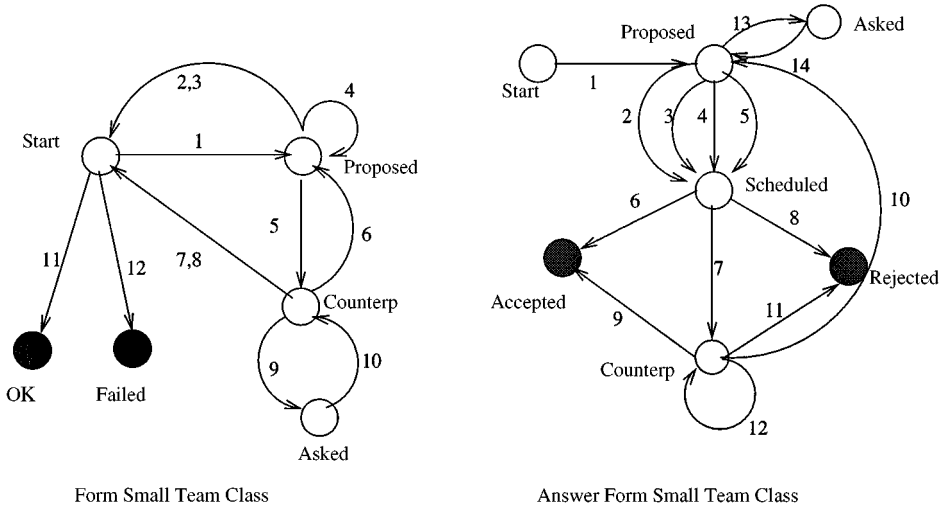

Figure 5. Forming the small team.

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 counterproposals before reaching an agreement. This is normal, because during these conversations contractual relations are established.

Figure 5 shows the conversation plans used by logistics and contractors in this stage. Contractors decide whether or not they will accept the contract by running their own scheduling engine with the input provided by logistics. This takes place in state `proposed` of the `answer-form-small-team-class` plan. The different possible ways to run the scheduler are represented as different rules on this state:

1. Schedule with 10 restarts (rule 2). This will be quick but may miss solutions.
2. Schedule with exhaustive backtracking (rule 3). This will find all solutions but, for larger or difficult problems, may consume much time.
3. Schedule with intelligent backtracking (rule 4). This will not be as quick as way 1 but may find more solutions.
4. Schedule in repair-based mode (rule 5). This requires a good initial approximation of a solution, which may or may not be available.

Each of these actions has its own rewards with respect to criteria like solution quality and execution time, and they are dynamically ordered by the system according to the load and priorities of the agent at the current moment. If a solution is produced, the agent accepts the proposal. If, after devoting enough time, no solution is found and there are many constraint violations or unassigned operations, the proposal is rejected. If a solution is not produced but the number of violations is reduced, the contractor counterproposes by requesting logistics to relax or change some constraints that might lead to a solution. If logistics and the contractor can reach no agreement at this level and this makes forming the small team impossible,

logistics will have to go back to the customer, through the `alternative-needed` state in the main logistics plan (figure 4).

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 fail to complete their activity, logistics will try to replace them with others from the large team. The large team contains contractors interested in the activity and willing to form a reserve team, hence it is the right place to look for replacements for 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 feasibility. Then, it brings a new proposal to the customer. Again, we may have a cycle of proposals and counterproposals 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 successfully. 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 will become available later on). If the customer accepts that (after negotiation), then the new (relaxed) order is processed and eventually may succeed. The reward structures used quantify the different alternative actions available to agents (as shown previously) and their preferences in negotiations. From the latter perspective, we give preference to accomplishing work and commitments above anything else, but prefer quick rejections to long negotiations that terminate with rejection. Least preferred is failure of committed work. We usually run the system with four to seven agents and 30–40 concurrent conversations. The COOL specification has about 12 conversation plans and about 200 rules and utility functions. The scheduler is an external process used by agents through an application program interface. All this takes less than 2600 lines of COOL code to describe. We remark on the conciseness of the COOL representation, given the complexity of the interactions and 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.

The model of teamwork illustrated here can carry out complex interaction structures among dynamically determined partners. Teamwork has been previously studied from a logical specification perspective by Cohen and Levesque (1991), the specification produced being extended and implemented in an industrial setting by Jennings (1995).

4.2. Coordination for dealing with perturbation

The purpose of the second application is to evaluate the different coordination strategies in a supply-chain setup where unexpected events occur. The dynamic behavior of a complex, multiechelon supply chain is hard to model in analytic form. For this reason we follow a simulation approach, where we represent the entities of the supply chain as agents and

the structured interaction processes taking place among them as conversations. In this application, we are specifically interested to see how coordination can be used to reduce the disruption caused by random breakdowns in the supply chain. For this reason, we focus on the management of inventories carried by the agents and, to a lesser extent, on customer satisfaction aspects.

The simulation operates in two modes. In the steady-state mode, we have no breakdowns. In this case, the operation of the supply chain depends on the accuracy of the forecasts. The numerical results obtained in this case show that the system's behavior depends on the ratio of the actual demand versus the forecast demand. If this is close to 1, the supply chain can process orders on time and ensure customer satisfaction. As the ratio grows over 1, more and more orders are delayed. This behavior is what we expect from the supply chain, and although we will not insist on it in the presentation, it gives us reassurance that the simulation model we have built is intuitive in the "normal" case. What we focus on in the following presentation is the situation where random breakdowns occur. In this case, we increase the amount of communication and coordination among the entities in the supply chain in a stepwise manner and measure the consequences of increased coordination after every step, over various parameters of the supply chain. As stated, these parameters deal mostly with carried inventories, the purpose of coordination being to reduce inventory when the processing capacity of supply chain elements is decreased. To obtain numeric results about inventory levels, we adopt a simple, intuitive model for the production capacity of workstations as explained next.

Enterprise structure. The Perfect Minicomputer Corporation (PMC; figure 6) is a small manufacturer of motherboards and personal computers situated in Toronto, Canada. The minicomputers are sold to customers in two markets, Canada and the United States and Germany and Austria. To satisfy the different standards of keyboard and power supply in the two markets, the computers need to be slightly different and are regarded as two distinct products. The motherboard is PMC's third product, sold to the computer industry of the Canadian and U.S. market.

Plants and production. PMC is a vertically integrated company. In addition to the assembly of the finished computer systems (computer, monitor, and keyboard), it assembles the motherboard and the computer boxes (without power supply) in separate plants in Toronto. Each plant has `Planning`, `Materials`, `Production`, and `Dispatching` agents. The `Planning` agent is responsible for production planning. The `Materials` agent handles raw product inventory (RPI), the on-order database for raw products, and all reception of raw products. The `Production` agent handles production and the work-in-progress inventory and has the knowledge of the plant architecture. The `Dispatching` agent handles the finished goods inventory (FGI) and all shipments from the plant. In each plant also is a set of workstations, bins, and stocks. The workstations are production units with a set number of lines (giving the number of units that can be processed simultaneously), a scrap rate (in percent), and a production time for each unit of a given product. The production capacity of the workstation is given by the number of lines times throughput rate (1/production time) minus scrap. Each workstation is modeled as an agent. The storage areas between workstations are modeled as *bins*. Each bin has a maximum inventory level, in which the

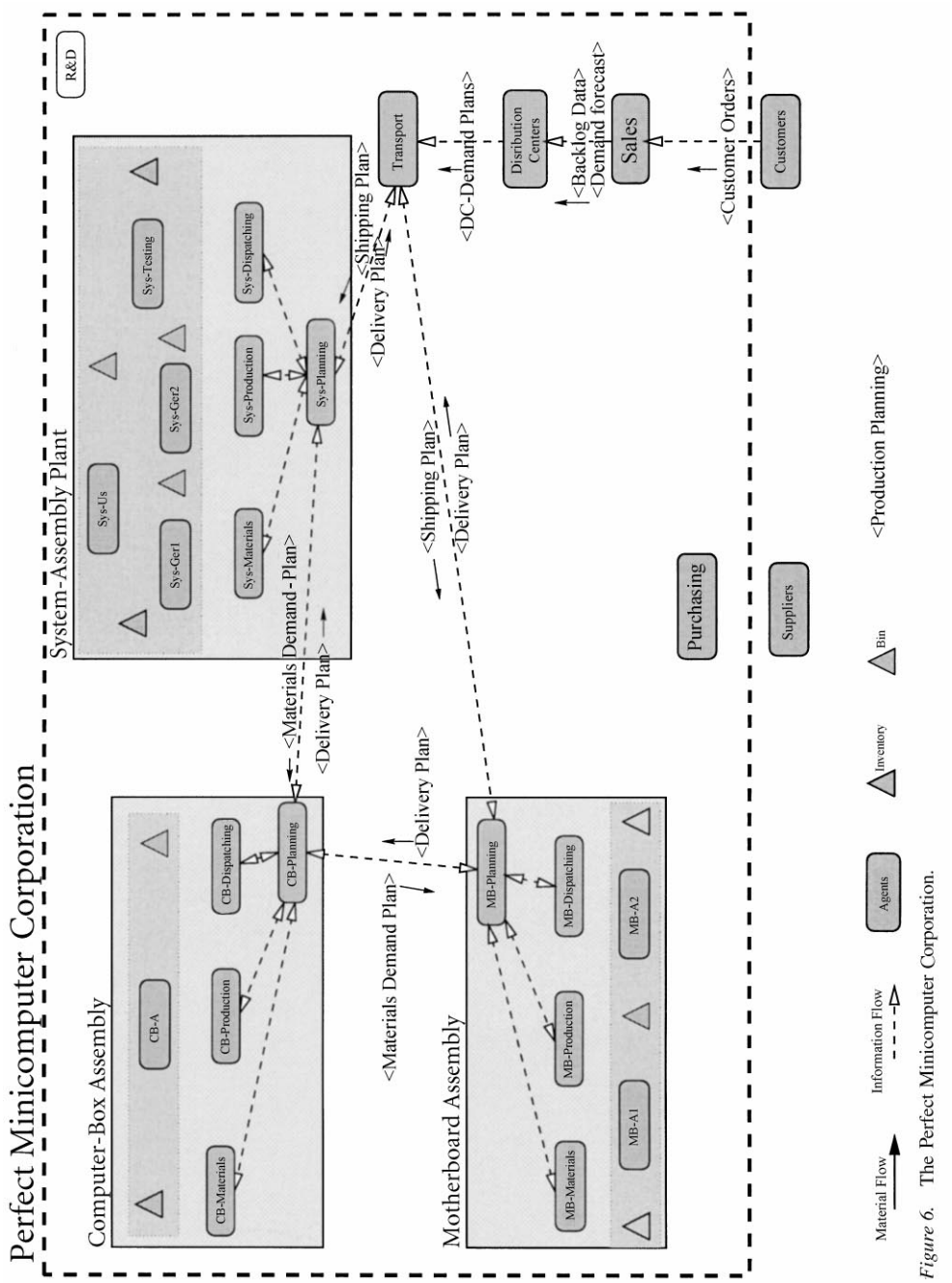

Figure 6. The Perfect Minicomputer Corporation.

bin is filled with inventory, hence no further products can be entered. A single bin agent in each plant is responsible for all bins in the plant. Each plant has two stocks areas, the RPI for in-coming components or raw materials and the FGI on the other end of production. Production is modeled as strictly pull production, where workstations finish products as long as the output bin is not full and start products as long as the input bin is not empty. Production ceases when weekly production goals are achieved.

Markets and distribution centers. PMC also owns and operates two distribution centers (DCs), one in Detroit for the Canadian and U.S. market (dc-us) and one in Hamburg for Germany and Austrian (dc-ger) market. All computers are distributed through these two centers. All motherboards sold to external customers are distributed through the Detroit distribution center. Each DC is modeled as an agent.

Suppliers and customers. Each external supplier is modeled as an agent. PMC has a `Purchasing` agent responsible for communication with suppliers. The `Purchasing` agent has knowledge of which parts to order from which suppliers. Three types of customers are identified for each product in each market: *a*, *b*, and *c* customers, with *a* customers being most important. Customers are modeled in one `Customer` agent for each market. The `Sales` agent in the company is responsible for communication with customers.

Transportation. A `Transport` agent is defined to handle transportation. This agent has knowledge of transportation times and capacities and damage rates, where applicable. It also keeps logs on transports currently underway. Deliveries from plant to distribution centers is modeled with uncertain transportation times (normally distributed) and, in some cases, limited capacity. Three types of carriers are used: boat, truck, and plane. Internal transportation from plant to plant is modeled as instantaneous and with unlimited capacity. All transports from external suppliers are the responsibility of the supplier and therefore not addressed in the model.

Coordination processes

Production planning. Production is planned through lists of goals for this week and a number of future weeks. These plans propagate upstream through the internal supply chain and come back downstream as plans of delivery. On the way upstream, each agent contributes its own knowledge.

To exemplify the use of conversation plans and rules, we look at the issuing of demand forecasts, which start production planning. (The demand forecast gives the expected number of units ordered for this and coming weeks.) The `Sales` agent has a conversation plan for distributing demand forecasts to the distribution centers. When a *demand forecast conversation* is created, the first rule of the conversation plan applies a specific method to compute the demand forecast. The next rule of the plan prepares the data for sending, and rule `dfc-3`, which follows, sends the message. The `?next-dc-forecast` variable contains the demand forecast for the market of the DC agent that is bound to the `?next-dc` variable:

```
(def-conversation-rule 'dfc-3
 :current-state 'sending-forecasts
 :such-that '(and (get-conv-var ?conv '?dc-left)
                  (get-conv-var ?conv '?ready-to-send))
 :transmit '(tell :sender ?agent
                  :receiver ?next-dc
                  :content (:demand-forecast ?next-dc-forecast)
                  :conversation ?convn)
 :do-after '(progn (put-conv-var ?conv '?dc-left
                     (rest (get-conv-var ?conv '?dc-left)))
                   (put-conv-var ?conv '?ready-to-send nil))
 :next-state 'sending-forecasts).
```

A demand-forecast message from `Sales` creates a *demand plan conversation* at the DCs. The rules of these demand plan conversations use knowledge of the DC's inventory levels. *DC-demand plans*, defining the targeted quantity of each product arriving at the DC at the end of this and coming weeks, are made and sent to the `Transport` agent (and similarly creates a corresponding conversation in the `Transport` agent). `Transport` knows how much is on way to the DC and therefore can make *shipping plans*, defining the quantity of each product that should be shipped from a plant to a given DC at the end of this week and a number of future weeks. The shipping plans are sent to the planning agents of the plants concerned.

The aim of a plant's `Planning` agent is to convert the incoming shipping plan (if it has external customers) and *materials-demand plans* from the next downstream plants (if it has internal customers) to the plant's own materials-demand plans for all internally supplied parts. These are sent to the next plants upstream. A materials-demand plan defines the number of units of a given product the plant needs this week and a number of future weeks. To calculate the materials-demand plans the `Planning` agent uses data from the other agents in the plant.

The materials-demand plans move upstream until they meet a last planning agent in the internal supply chain. This agent makes *delivery plans* for each customer (next plants downstream or transport for deliveries to DCs), defining the number of units the plant will deliver this week and in a number of future weeks. This, of course, is the total demand limited by part availability and production capacity. On receiving delivery plans from upstream internal suppliers, a planning agent has the knowledge it needs to decide the *actual building plan* of the plant; that is, the production goals for this and coming weeks. It also makes its own delivery plans, and these plans will flow downstream to the end of the supply chain.

Materials ordering, delivery, and reception. From the actual building plan, via the bill of materials, the materials agent can calculate a *materials-order plan* for externally supplied parts. The plans are sent to the purchasing agent, who transforms them to part orders for the suppliers. The supplier agents sends acknowledgment messages to the materials agents. The materials agents update their on-order database. Materials shipments arriving at the plants are modeled as messages sent by the suppliers to the materials agents. The materials agents update inventory and on-order data.

Product dispatching, transportation, and reception. Product transportation from plant to DC is started through messages from dispatching agents to the `Transport` agent. Arrivals at DC are done by messages from `Transport` to the DC agent.

Dealing with unexpected events. Each agent within the corporation records its own relevant data every week, building a database that is communicated to a `Simulation` agent at the end of the simulation and saved for later analysis. We measure parameters related to inventory levels and customer satisfaction. Examples include the value of all inventories, the company backlog, the in-coming orders, the shipments from plants to DCs, the average time from order arrival until product delivery, and the percentage of shipments delivered on time. We are especially interested in understanding the value of various coordination structures when unexpected disruptions occur in the supply chain and how coordination can be used to reduce the negative consequences of these disruptions. A typical situation is a machine breakdown during normal operation. Such an event tends to increase the level of raw product inventory in the plant where the breakdown occurs because the plant's ability to consume inventory is diminished. The carried inventories of the upstream and downstream plants also are affected and specific coordination is needed to attenuate these effects.

To see how coordination can be used to deal with this problem, we perform a series of experiments involving breakdown of workstations in several plants and using various coordination mechanisms for dealing with them. We can use coordination at two levels to attenuate the disruptions produced by breakdowns: intraplant coordination (that is, coordination among the agents within a plant) and interplant coordination (that is, coordination among different plants).

Within a plant we implement two alternative coordination protocols for handling machine breakdown. The first protocol is simply do nothing. Knowledge of the breakdown remains with the workstation agent and the planning agent continues to plan production as if capacity was unchanged. This setup is referred to as *no notifications*. The second protocol allows workstation agents to notify the production agent on machine breakdown. A production agent that is notified will pass the message on to the plant's planning agent. The planning agent now knows of its plant's reduced production capacity and uses this knowledge in setting the weekly production plans. This setup is referred to as *notifications*.

For interplant coordination, we first remember that, in our model, production planning is coordinated by having demand plans flow upstream the supply chain, while delivery plans flow downstream. The purpose of delivery plans is to provide downstream agents knowledge of the state of the upstream supply chain. Downstream planning agents thus know whether the upstream ability to deliver will constrain their future production. If this is the case, the plant may reduce the ordering of other parts and thereby avoid filling its RPI with unused materials. At the interplant level we have two more setups: no delivery plans and delivery plans. We expect that, in a steady-state situation, with no breakdown, the absence of delivery plans has no effect since downstream production is not constrained by upstream production. When we introduce machine breakdowns, however, the effects should be clear.

To analyze the effects of these coordination strategies we simulate breakdowns in various plants and then run the system with the four possible combinations of internal notification and delivery plans: (1) no delivery plans and no notification, (2) no delivery plans and

notification, (3) delivery plans and no notification, (4) delivery plans and notification. In all cases, we assume the breakdown occurs in week 35 and takes 12 weeks to repair. The severity of the breakdown is assumed to be high, 80% of the plant's capacity being lost.

Some results of these simulations are shown in figures 7, 8, and 9. In figure 7, we assume that the breakdown occurs in the system test and assembly plant (the last plant in the chain) and we show the change in the RPI level in cases 3, on the left, and 4, on the right. The results show that the simple notification introduced reduces the average value of the raw product inventory at the system test plant (where the breakdown occurred) by 26%. It also shows that, for the upstream plants, there is a noticeable increase of the same inventory, because they have to keep more inventory in their own stock. Globally, however, the total inventory decreases about 4% in average. The most important consequence is avoiding the sudden take-off of the system test plant's stock. In the nonnotification case, the stock is more than tripled in the 10-week period following the breakdown. The notification reduces the magnitude of the peak by almost half.

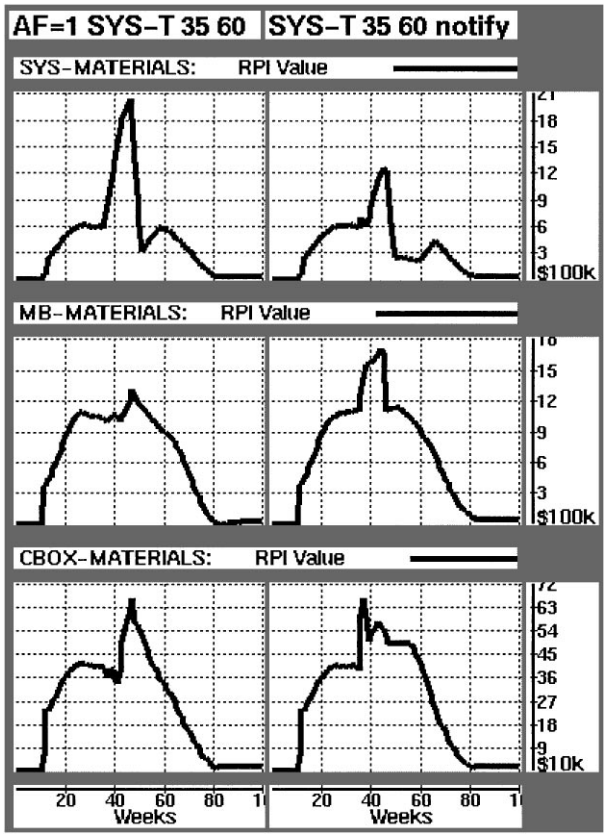

Figure 7. Effect of system test plant breakdown: left; delivery plans and no notification; right; delivery plans and notification.

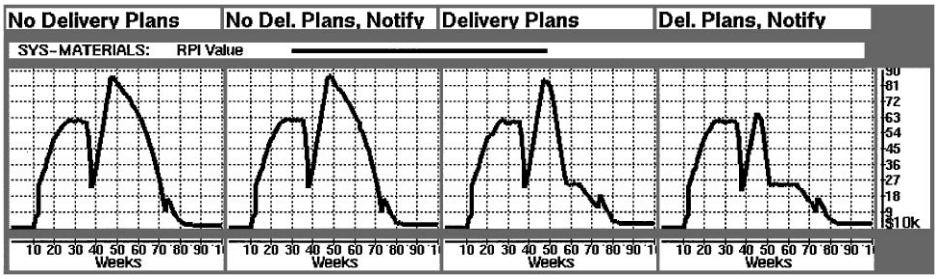

Figure 8. Effects of computer box plant breakdown, all cases.

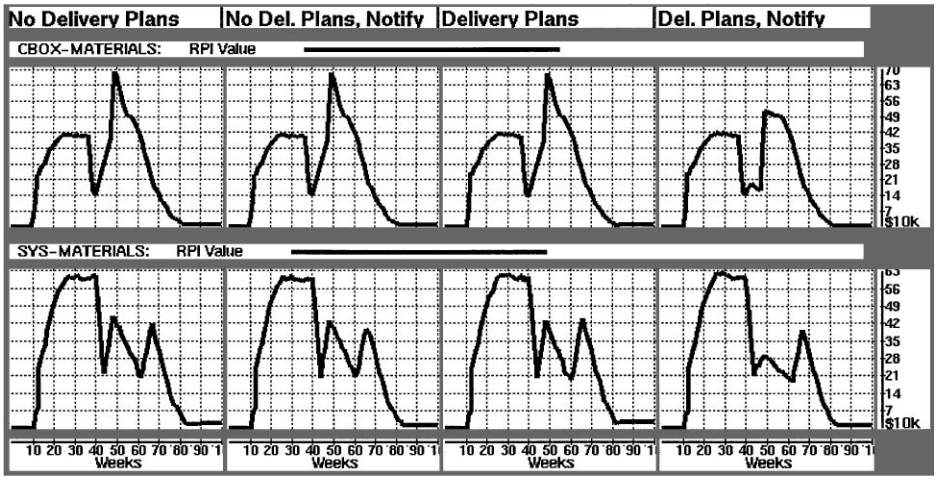

Figure 9. Effects of motherboard plant breakdown, all cases.

In figure 8, the breakdown occurs in the computer box assembly plant (second in the chain), and we show the inventories in all four cases for the next plant downstream (system assembly and test). First, we notice that the local notification has virtually no effect when delivery plans are not sent downstream. This is to be expected, since in this case the breakdown knowledge is not propagated to downstream planning agents. However, when delivery plans are used, even without notification, there is a clear gain in RPI reduction of about 16%. If notification also is used, the gain is as high as 26%.

Finally, in figure 9, we assume the breakdown occurs in the motherboard plant (first in the chain). We see that, for the other plants, only the combined use of notification and delivery plans significantly reduces the level of inventory.

On the customer satisfaction side, although a loss of production is inevitable, the notifications allow the enterprise to update the delivery time quotations sent the customer in advance and thus maintain the customer's trust.

Evaluation. The preceding supply-chain system has 40 agents and just about the same number of conversation plans. The entire specification takes about 7500 lines of COOL code, plus about 2000 lines for GUIs. A typical simulation run over 100 weeks generates thousands of message exchanges and takes less than one hour to complete (no optimizations attempted, and the system runs in an interpreted mode). The system was written by one author, who has no computer science background, in less than three months. Learning the underlying agent and coordination technology was done in another two months, during which time a simpler supply chain was built. (Some limited code sharing between these systems occurred.) We take these data as early indications that the agent coordination model is natural, understandable, and adequate to modeling distributed agent systems like the supply chain. We are aware that such evidence, collected from a reduced number of applications, is only partial. Since we are dealing with evaluating a computer language, more compelling evidence requires much more experimentation and many more users than we could afford. We believe, however, that, incomplete as they are, our results show promise that our plan-action-oriented coordination language addresses the problem of multiagent coordination in a practically relevant manner.

In terms of how far we have gone with the understanding of coordination as a way to cope with disruptions in a dynamic supply chain system, the answer is that we are in an early stage. Although we have an appropriate experimental setup for studying coordination in face of unexpected events, we have modeled only very simple situations of this kind. We expect to go deeper into the problem once we integrate into our setup more powerful scheduling solvers that agents would use to plan production locally. These would allow agents to develop a precise understanding of the options they have when responding to an unexpected event and the consequences of these options. Globally, agents would be in a position to manage change by negotiating about the actions and objectives of each of them. As in the teamwork application described previously, this negotiation would consist of proposing and trying to agree on constraints about agents goals and actions.

5. Conclusions

We believe we have contributed in several ways to the goal of constructing models and tools enabling multiagent systems to carry out coordinated work in real-world applications. We have contributed a model of the new type of coordination knowledge as complex, coordination enhanced plans involving interactions by communicative action. The execution by agents of these plans results in multiple structured conversations taking place among agents. These ideas have been substantiated into a practical, application-independent coordination language that provides constructs for specifying the coordination-enhanced plans as well as the interpreter supporting their execution. Our interpreter supports multiple conversation management, a diverse rule typology that, among others, provides for handling exceptional or unexpected situations, conversation synchronization, and optimization of plan execution by decision-theoretic mechanisms.

In cooperation with industry partners, we applied these models and tools to industrially relevant problems to keep our work in touch with reality and "falsify" our solutions as early as possible based on feedback from reality.

With respect to the coordination model, previous work has investigated related state-based representations (von Martial, 1992) but has not consolidated the theoretical notions into usable language constructs, making it hard to use these ideas in applications. Formalization of mental state notions related to agency (like Cohen and Levesque, 1990) have provided semantic models that clarify a number of issues but operate under limiting assumptions that similarly make practical use and consolidation difficult. Some conversational concepts have been used by Shepherd, Mayer, and Kuchinsky (1990) and Medina-Mora et al. (1992) in the context of collaborative and workflow applications. We extend and modify them for use in multiagent settings and add knowledge acquisition, sophisticated control, and decision-theoretic elements that lead to a more generic, application-independent language. Agent-oriented programming (Shoham, 1993) similarly uses communicative action, rules, and agent representations. Our language differs from that in the explicit provision of plans and conversations, the more powerful control structures that emerge from them, and the decision-theoretic enhancements.

The coordination language and the shell have been evaluated on several problems, including supply chain coordination projects carried out in cooperation with industry. Although the number of applications we built as well as the number of users of our system both are limited, the evidence we have so far shows that our approach is promising in terms of naturalness of the coordination model, effectiveness of the representation and power, and usability of the provided programming tools. 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 or days, enabling us to immediately demonstrate its functionality. Moreover, we have found the approach explainable to and usable by industrial engineers who do not necessarily have a computer science background.

A number of other system capabilities mentioned in the paper are work in progress and have not played an important role in the presented applications. This is the case with role-based organization models of obligation and authority among agents. Some of our research results in this direction are described in Barbuceanu (1998) and Barbuceanu, Gray, and Mankovski (1998). As these become more mature we will integrate them in our supply chain work as well.

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

Austin, John L., *How to Do Things with Words*, Clarendon Press, Oxford, England (1962).

Barbuceanu, Mihai, "Agents That Work in Harmony by Knowing and Fulfilling Their Obligations," *Proceedings of AAAI-98*, pp. 89–96, AAAI Press, Madison, WI (July 1998).

Barbuceanu, Mihai and Fox, Mark S., "The Architecture of an Agent Building Shell," in *Intelligent Agents II*, Michael Wooldridge, Joerg P. Mueller, and Milind Tambe (Eds.), Lecture Notes in Artificial Intelligence, Springer-Verlag, Berlin, Vol. 1037, pp. 235–250 (March 1996a).

Barbuceanu, Mihai and Fox, Mark S., "Capturing and Modeling Coordination Knowledge for Multi-Agent Systems," *International Journal of Cooperative Information Systems*, Vol. 5, Nos. 2 & 3 pp. 275–314 (1996).

Barbuceanu, Mihai and Fox, Mark S., "COOL: A Language for Describing Coordination in Multi-Agent Systems," in *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS-95)*, Victor Lesser (Ed.), AAAI Press/MIT Press, San Francisco, CA, pp. 17–24 (June 1995).

Barbuceanu, Mihai, Gray, Tom, and Mankovski, Serge, "The Role of Obligations in Multi-Agent Coordination," *International Journal on Applied Artificial Intelligence*, Vol. 13, Nos. 1–2 pp. 11–39 (January–March 1999).

Bellman, Richard E., *Dynamic Programming*, Princeton University Press, Princeton, NJ (1957).

Cohen, Phil, R. and Levesque, Hector, "Intention Is Choice with Commitment," *Artificial Intelligence*, Vol. 42, pp. 213–261 (1990).

Cohen, Phil, R. and Levesque, Hector, "Teamwork," *Nous*, Vol. 15, pp. 487–512 (1991).

Cohen, Phil, R. and Levesque, Hector, "Communicative Actions for Artificial Agents," in *Proceedings of the First International Conference on Multi-Agent Systems*, Victor Lesser (Ed.), AAAI Press and MIT Press, San Francisco, pp. 65–72 (June 1995).

Finin, Tim, Labrou, Yannis, and Mayfield, James, "KQML as an Agent Communication Language," in *Software Agents*, Jeffrey M. Bradshaw (Ed.), MIT Press, Cambridge, MA (1995).

Fox, Mark S., "Beyond the Knowledge Level," in *Expert Database Systems*, Larry Kerschberg (Ed.), Benjamin/Cummings, pp. 455–463 (1987).

Jennings, Nicholas R., "Towards a Cooperation Knowledge Level for Collaborative Problem Solving," in *Proceedings 10th European Conference on Artificial Intelligence*, pp. 224–228, Vienna (1992).

Jennings, Nicholas R., "Commitments and Conventions: The Foundation of Coordination in Multi-Agent Systems," *The Knowledge Engineering Review*, Vol. 8, No. 3, pp. 223–250 (1993).

Jennings, Nicholas R., "Controlling Cooperative Problem Solving in Industrial Multi-Agent Systems Using Joint Intentions," *Artificial Intelligence*, Vol. 75, No. 2, pp. 195–240 (1995).

Medina-Mora, Raoul, Winograd, Terry, Flores, Roberto, and Flores, Fernando, "The Action Workflow Approach to Workflow Management Technolgy," in *Proceedings of Computer Supported Cooperative Work 1992*, pp. 281–288, (1992).

Patil, Richard, Fikes, Richard, Patel-Schneider, Peter, McKay, Don, Finin, Tim, Gruber, Tom, and Neches, Robert, "The ARPA Knowledge Sharing Effort: Progress Report," in *Principles of Knowledge Representation and Reasoning: Proceedings of the Third International Conference*, Bertrand Nebel, Charles Rich, and William Swartout, (Eds.), Morgan Kaufman, San Mateo, CA (November 1992).

Puterman, Martin L., *Markov Decision Processes: Discrete Stochastic Dynamic Programming*, Wiley, New York (1994).

Rosenschein, Stan, R. and Kaebling, Leslie, P., "A Situated View of Representation and Control," *Artificial Intelligence* Vol. 73 Nos. 1–2, pp. 149–173 (1995).

Searle, John, R., *Speech Acts*, Cambridge University Press, Cambridge, England (1969).

Shepherd, Allan, Mayer, Niels, and Kuchinsky, Alex, "Strudel—An Extensible Electronic Conversation Toolkit," in *Proceedings of Computer Supported Cooperative Work 1990*, pp. 93–104 (1990).

Shoham, Yoav, "Agent-Oriented Programming," *Artificial Intelligence*, Vol. 60, pp. 51–92 (1993).

Von Martial, Frank, *Coordinating Plans of Autonomous Agents*, Lecture Notes in Artificial Intelligence 610, Springer-Verlag, Berlin, (1992).