Appeared in Intelligent Agents II, LNAI 1037
Springer Verlag, 1996.

The Architecture of an Agent Building Shell

Mihai Barbuceanu and Mark S. Fox

Enterprise Integration Laboratory
University of Toronto,
4 Taddle Creek Road, Rosebrugh Building,
Toronto, Ontario, M5S 1A4

{mihai,msf}@ie.utoronto.ca

Abstract. The agent view provides a level of abstraction at which we envisage computational systems that are able to interoperate globally on "the Net". It abstracts from aspects like the hardware or software platforms of various components or the internal structure, methods or processing of these components, focusing attention on how complex, heterogenous, distributed and evolving systems can be built from interoperable and reusable building blocks. From the practical point of view, multiagent systems engineering requires the ability to reuse abstract descriptions of system components, services, knowledge bases and coordination structures. Based on this recognition, we are developing an Agent Building Shell that provides several reusable layers of languages and services for building agent systems: coordination and communication languages, description logic based knowledge management, cooperative information distribution, organization modeling and conflict management. We are applying the approach to develop multiagent applications in the area of manufacturing enterprise supply chain integration.

1 Introduction

The agent view provides a level of abstraction at which we construe computational systems that interoperate globally on the entire "Net". Such systems will link diverse organizations, people, cultures on a single virtual platform. To build these systems we need to abstract from aspects like the hardware or software platforms of various components or the internal structure, methods or processing of these components, focusing attention on how complex, heterogenous, distributed and evolving systems can be built from interoperable and reusable building blocks. We call the computational entities that can operate at this level *agents*. For our purposes, we consider an agent to be a piece of software that *(1) is significantly autonomous and entrusted in performing its functions and (2) operates on the entire Net by relying on application-independent communication and interaction protocols with other "agents".*

From the practical standpoint, multiagent systems of this sort can not be efficiently build without the ability to reuse abstract descriptions of system components, services, knowledge bases and coordination structures. Based on this recognition, we are developing an Agent Building Shell that provides reusable languages and services for

agent construction, relieving developers from the effort of building agent systems from scratch and guaranteeing that essential interoperation, communication and cooperation services will always be there to support applications.

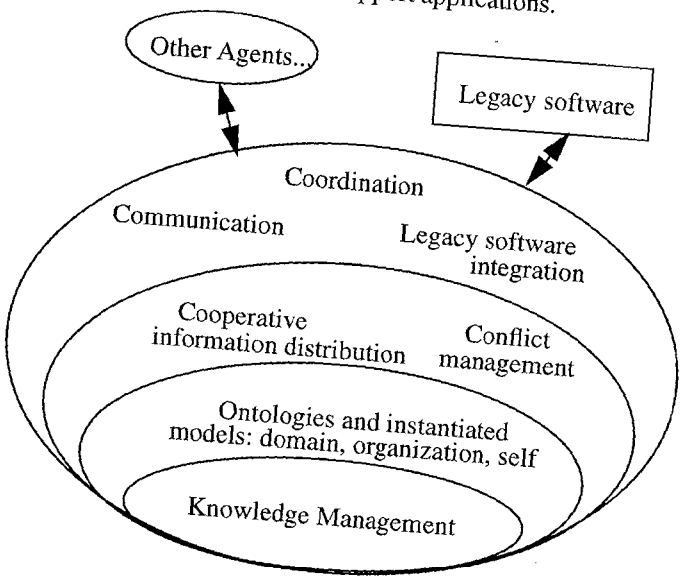


Fig. 1. Architecture of the Agent Building Shell

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

- *Agent interaction*: How do agents communicate? How are the semantic problems related to conflicting or different meanings of the exchanged terms and expressions solved? How do agents coordinate in joint work? How do agents model each other in a cooperative community?
- *Representation*: How do agents represent their local views of the domain? How is the local view updated or maintained as a consequence of interaction? How do agents revise their beliefs due to exchanged information? How do agents share models and how does the shared model change? How are common-sense issues, e.g. time, action, causality, handled?
- *Reasoning*: How do the requirements for communication and coordination impact the internal reasoning of agents? How do agents handle contradictory information, and how is consistency maintained across agents that may have different goals, views, preferences?
- *Integration*: How can pre-existing (legacy) applications be integrated into agents and thus used in agent communities?

We have built support for these issues in a layered agent architecture shown in figure 1. In the remainder of this paper we discuss what these layers provide and how they have been implemented and integrated. The discussion will be carried out in the

context of our main application, the agent-based integration of the supply chain of manufacturing enterprises. Because of that, we start by presenting this application domain and then we continue with the layers of the architecture. In the end, we return to the supply chain and show how the architecture supports its agent-based integration.

2 Integrating the Supply Chain

The supply chain of a modern enterprise is a world-wide network of suppliers, factories, warehouses, distribution centres and retailers through which raw materials are acquired, transformed into products, delivered to customers, serviced and enhanced. In order to operate efficiently, supply chain functions must work in a coordinated manner. But the dynamics of the enterprise and of the world market make this difficult: exchange rates unpredictably go up and down, customers change or cancel orders, materials do not arrive on time, production facilities fail, workers are ill, etc. causing deviations from plan. In many cases, these events can not be dealt with locally, i.e. within the scope of a single supply chain "agent", requiring several agents to coordinate in order to revise plans, schedules or decisions. In the manufacturing domain, the agility with which the supply chain is managed at the (short term) tactical and operational levels in order to enable timely dissemination of information, accurate coordination of decisions and management of actions among people and systems, is what ultimately determines the efficient achievement of enterprise goals and the viability of the enterprise on the world market.

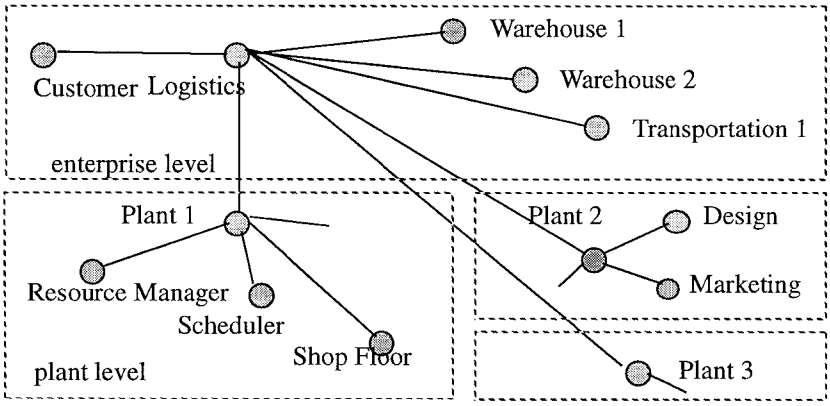


Fig. 2. Multi-level supply chain.

We address these problems by organizing the supply chain as a network of cooperating agents, each performing one or more supply chain functions, and each coordinating their actions with other agents. Figure 2 shows a multi-level supply chain. At the enterprise level, the Logistics Agent interacts with the Customer about an order. To achieve the Customer's order, Logistics has to decompose it into activities (including for example manufacturing, assembly, transportation, etc.). Then, it will negotiate with the available plants, suppliers and transportation companies the execution of these activities. If an execution plan is agreed on, the selected participants will com-

mit themselves to carry out their part. If some agents fail to satisfy their commitment, Logistics will try to find a replacement agent or to negotiate a different contract with the customer.

At the plant level, a selected plant will similarly plan its activities including purchasing materials, using existing inventory, scheduling machines on the shop floor, etc. Unexpected events and breakdowns are dealt with through negotiation with plant level agents or, when no solution can be found, submitted to the enterprise level.

Some of the major challenges for such an application include the criticality of a good decomposition into agents (an issue for the future agent oriented analysis methods), the complexity of coordination knowledge requiring specialized tools for incontext capture and use, the importance of shared conceptualizations for semantically unifying agent interaction, the importance of drawing the right line between what is (or can be) automated and what should remain the responsibility of the human user in a multi-agent system, the need for open architectures in which legacy or purpose built applications can be easily integrated and used.

3 Communication, Coordination and Legacy Software Integration

These three functions, forming the outer layer of the architecture, are supported by our COOrdination Language (COOL). Essentially, COOL is (i) a language for describing the coordination level conventions used by cooperating agents (ii) a framework for carrying out coordinated activities in multiagent systems (iii) a tool for design, experimentation and validation of cooperation protocols and (iv) a tool for incremental, in context acquisition of cooperation knowledge.

3.1 Communication

COOL has a communication component that implements an extended version of the ARPA sponsored KQML language [6]. Essentially, we keep the KQML format for messages, but we leave freedom to developers with respect to the allowed vocabulary of communicative action types[1] (called performatives in KQML). Also, we do not impose any content language. Our implementation provides TCP/IP supported transport and services like persistent storage of received KQML messages, visual tools for message browsing, composition, sorting into folders and general pattern matching.

3.2 Coordination

The purpose of the coordination layer of COOL is to describe, execute, validate and acquire coordination protocols, i.e. shared conventions about agent interaction. COOL provides constructs for the following entities.

1. As long as KQML performatives do not have declarative semantics, this raises no problem, as we work with the operational semantics provided by COOL. When declarative semantics will be available, we will restrict to those performatives that have such semantics.

Agent: a programmable entity that can exchange messages, change state and perform actions. Agents carry out conversations with other agents or perform local actions within their environment. Agents exist in local or remote environments (see bellow).

Agent execution environment. Its purpose is to "run" agents by managing message passing and scheduling agents for execution. Environments exist on different sites (machines) and a directory service makes message transmission work just the same among sites as within sites. The environment also provides a wealth of tools for visual manipulation (browsing, editing, system set-up, animated execution).

Generic conversation descriptions (*conversation classes*). These are rule based descriptions of what an agent does in certain situations (for example when receiving messages with given structure). A conversation class thus specifies the available conversation rules, their interpreter (executor) and the local data-base that maintains the state of the conversation. The latter consists of a set of variables whose values are manipulated by conversation rules. Since rules are indexed on the finite set of values of a particular variable (the *current-state*), conversations admit a finite state machine representation that is often used for visualization purposes. Figure 3 shows the conversation class governing the Customer's conversation with Logistics in our supply chain. Arrows indicate the existence of rules that will move from one state to another. Figure 4 shows the associated transition diagram of this conversation class. Each conversation class describes a conversation from the viewpoint of an individual agent (in figure 3 the Customer). For two or several agents to "talk", the conversation classes governing their interaction must generate sequences of messages that the others' conversation classes can process.

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

Fig. 3. Customer-conversation

Conversation rules describe the actions that can be performed during conversations.- For example, an agent in a given state receives a messages of specified type, does local actions (e.g. updating local data), sends out messages, and switches to another state. Figure 5 illustrates a conversation rule from the customer-conversation.

Error recovery rules specify how incompatibilities among the state of a conversation and the incoming messages are handled. Such incompatibilities can have many causes - message delays, message shuffling, lost messages, wrong messages sent out, etc. Error recovery rules deal with this by performing any action deemed appropriate,

such as discarding inputs, initiating clarification conversations with the interlocutor, changing the state of the conversation or just reporting an error.

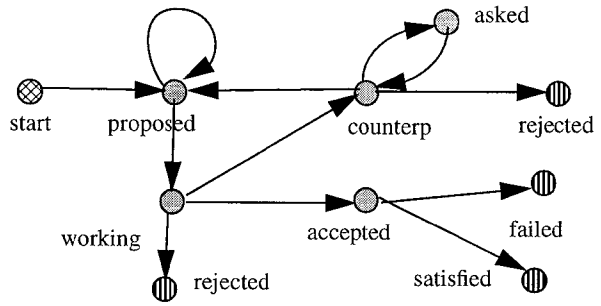


Fig. 4. Finite state representation of customer-conversation

```
(def-conversation-rule 'cc-3
 :current-state 'working
 :received '(counter-propose :sender logistics :content ?c)
 :next-state 'counterp
 :incomplete nil)
```

Fig. 5. Conversation rule cc-3.

Actual conversations instantiate conversation classes and are created whenever agents engage in communication. *Conversation selection* is the process of selecting a conversation class that will handle a conversation with an agent that has sent a message. We do this by matching an expressed :intent of the message with the :intent-test in the existing conversations. *Suspending a conversation* occurs as a consequence of the fact that an agent may shift its attention from one conversation to another. A suspended conversation can be resumed when certain events occur, such as the more prioritar conversation being terminated. This brings *multiple conversation management capabilities* allowing agents to carry out several conversations in the same time, by *selecting the next conversation* to carry out and the conditions under which conversations can be *suspended* or *resumed.*

Continuation rules specify how agents accept requests for new conversations or select a conversation to continue from among the existing ones. Continuation rules perform two functions. First, they test the input queue of the agent and apply the conversation class recognition mechanism to initiate new conversations. Second, they test the data base of ongoing conversations and select one existing conversation to execute.

3.3 Legacy Software Integration

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

3.4 Incontext Acquisition of Cooperation Knowledge

Coordination protocols for supply chain integration are not only very complex but often impossible to specify completely. Rather than requiring developers to get the protocols right from the beginning, we allow conversation rules that are incomplete (do not contain complete conditions, actions, messages, etc.). Such rules are handled by turning control to the user to decide exactly which rule is executed and how. During use, users can decide to augment incomplete rules with more knowledge, ultimately making some of them complete and automatically executable. This approach is provided for both continuation and conversation rules. The support includes visual interfaces in which users can try out various rules, inspect and modify the stati of conversations, data bases, message queues, etc., perform rule specified or user required actions, etc. This provides a unified debugging, development and acquisition environment for cooperation knowledge that has proven invaluable when programming complex multiagent interactions. For illustration, figure 6 shows an incomplete rule from the customer-conversation that allows a user interacting with the Customer agent to answer (indeterminate) questions from the Logistics agent.

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

Fig. 6. Incomplete conversation rule.

4 Description Logics for Knowledge Management

Having described the interaction mechanisms provided by the shell, we now switch to a bottom-up presentation of the other layers. We start with the knowledge management services based on description logics. Description logic languages (or terminological languages) integrate aspects of object-oriented and logic representations [4,5]. They express knowledge in a modular fashion, using inheritance and hierarchical organizations and rely on well-defined declarative semantics to describe the meaning of constructs. For this work, we use our own description logic language, called MODEL [2], that provides the usual concept-forming operators - conjunction, value restrictions, number restrictions - roles and subroles, disjointness declarations, primitive and defined concept specifications. The language's T-Box provides the usual services of constructing the complete form of concepts and automated classification based on subsumption checking. The language's A-Box is essentially a constraint propagation engine that makes instances conform to the various constraints asserted about them. It uses a propositional representation of instances and roles, a boolean constraint propagation TMS [1] and a number of other services including advanced queries and rules.

(concept activity (:and (:all involved-resource resource)(:the state (:oneof enabled active dormant)))
(role involved-resource (:domain activity)(:range resource))
(role produce(:and involved-resource))
(role consume(:and involved-resource))
(concept black-hole-activity (:atleast 1 consume)(:atmost 0 produce))
(concept miracle-activity (:atleast 1 produce)(:atmost 0 consume))
(concept normal-activity (:atleast 1 produce)(:atleast 1 consume))

a - terminology

(i) Complete concepts with implied descriptions. E.g. (:atleast 2 involved-rersource) for normal-activity.

(ii) Organize descriptions in a subsumption lattice, where e.g. normal-activity, black-hole-activity, miracle-activity are subsumed by activity (not represent explicitly).

(iii) Check existence of activities subsumed by black-hole-activity or miracle-activity - these don't make sense in the domain.

b - T-Box actions

Assert: produce(A1 Bolt)$^{[1,7]}$, consume(A1 Steel)$^{[3,11]}$
Propagate: activity(A1)$^{[1,7]}$, activity(A2)$^{[3,11]}$
Recognize: normal-activity(A1)$^{[3,7]}$
Assert clause:
normal-activity(A1)$^{[3,7]}$<- produce(A1 Bolt)$^{[1,7]}$, consume(A1 Steel)$^{[3,11]}$

c - A-Box actions

Fig. 7. Example and services provided by the description logic

Figure 7 shows an example terminology and illustrates how this is processed by the assertional and terminological services of the language. Especially note the use of temporal reasoning at the assertional level (superscripts indicate time intervals during which propositions are believed). Accounts of our approach to temporal reasoning in description logics are available elsewhere [3].

5 Organizational Modeling

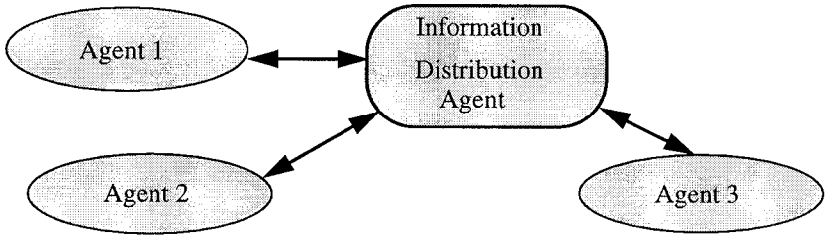
Agents can not operate autonomously unless they have an understanding of the environment they are in. This understanding consists of models of the other agents in the environment, the roles they play, the goals they are pursuing, the actions they are empowered to execute, the information they are interested in, the services they can provide, the established communication and authority channels, etc.

To endow our actual agents with this capability we have developed an organization ontology that provides the necessary distinctions for describing organizations and agents. Actual organizational models built with this terminology are used by agents to: (i) retrieve the most appropriate coordination/conversation model, (ii) carry out conflict management and negotiation, (iii) retrieve recipients of volunteered information, (iv) represent themselves as part of the organization.

We view an organization as a collection of *organization agents* that work to satisfy shared goals. Agents play various *organizational roles*, by which they assume responsibility for *organizational goals*, and use *communication and authority links* to distribute activity among themselves. The organization model defines what is the range of authority of an agent and allows conflicting agents to negotiate in order to resolve the conflicts [8].

6 Cooperative Information Distribution

Cooperative information distribution allows an agent to distribute information to other agents based on the *content* of the information and the expressed *interests* of agents. The essential capability needed to perform this function is being able to *prove* that a piece of information satisfies an expressed interest. This proof is performed by the classification and recognition services of the description logic language.



a - Information Distribution Agent servicing functional agents

Topic of interest of Agent-2:
(concept heavy-component
(:and component (:gt weight 5000))),

that is "any component whose weight is greater than 5000"
(subscribe :content (stream-about :content(query heavy-component[:alltime march-april 94])))

Topic of interest of Agent-3:
(concept weight-change
(:and change (:the changed component) (:gt difference 100))),

that is "any change in weight such that the difference between the new and the old value is at least 100"
(subscribe :content (stream-about :content (query weight-change[:anytime spring 94])))

b - Topics of interest and subscriptions

Agent-1 to IDA:
(achieve :content (part p-111))
(achieve :content (part-of p-111 c-12))
(achieve :content (part-of p-111 c-13))
(achieve :content (part c-12))
(achieve :content (part c-13))

(achieve :content (weight c-12 2700)[:starting feb 94])
(achieve :content (weight c-13 3400)[:starting jan 94]).
IDA inferences: (component p-111), (weight p-111 6100)[:starting feb 94], (heavy-component p-111)[:starting feb 94]
IDA to Agent-2: (tell :content (heavy-component p-111)[march-april 94]).
Agent-1 to IDA: (deny :content (weight c-12 2700)[:starting april 94]), (achieve :content(weight c-12 1000)[:starting april 94]).
IDA to Agent-2: (deny :content (tell :content (heavy-component p-111)[march-april 94]))
IDA to Agent-3: (tell :content (weight-changed c-12 1000)), (tell :content (weight-changed p-111 1000))

Fig. 8. Content-based information distribution scenario.

Figure 8 illustrates an information distribution scenario in which an agent (IDA) performs information distribution functions by routing information among several agents according to their interests. The scenario uses KQML communication primitives. Note the belief revision function performed by the IDA when Agent-1 retracts some information and asserts something different. The IDA sends a denial message to Agent-2 who received information that is no longer valid. We view content-based information distribution as providing the function of a "nervous system" of the agent, by continuously distributing and collecting information that will feed the higher cognitive levels.

Translation. Cooperating agents often conceptualize the world according to partially distinct ontologies, reflecting the different perspectives agents have. This reduces the bandwidth of communication and requires that agents do more work individually to derive the information they need. Information distribution functions can help by performing translation functions that e.g. can make knowledge implicit in one ontology explicit in another, or can organize knowledge in different manners (e.g. introducing/removing intermediate concepts).

Shared model change. Agents make decisions assuming that whatever information has been communicated to them continues to be true unless explicitly invalidated. In other words, agents expect to be notified about any changes of previously received information and will assume that no change occurred if not notified. The solution we support is based on having agents send to the IDA justifying *clauses* describing how a belief of interest to the agent (consequent) depends on other beliefs (antecedents) that can be modified inside the IDA. When all antecedents become true, the IDA sends the consequent belief to all the interested agents. When any of the antecedents becomes false, the IDA sends denial messages wrt the consequent. In this way, any agent is safe to assume that any received belief not explicitly denied is still valid. In [3] we present detailed illustrations of the translation and shared model change management techniques we employ.

7 Conflict Management with the Credibility/Deniability Model

Groups of agents in a multi-agent reasoning system can often hold incompatible beliefs in the following sense:

- Agent-1 believes p and communicates it to Agent-3
- Agent-2 believes q and communicates it to Agent-3
- Agent-3 believes p because it was communicated by Agent-1, q because it was communicated by Agent-2 and has local knowledge stating that *p&q -> false.*

Often, the agent that encounters such a contradiction has to eliminate it by retracting some current belief that supports either p or q. This section addresses the issue of how to determine which of the possible supporting beliefs to retract to reinstall consistency.

7.1 Credibility and Deniability

Consider an organization where the Marketing Agent has determined that for a new automotive product a v6 engine would sell better. Hence marketing will sent the Management Agent a message telling that the engine should be a v6: (v6engine e1)[starting 13 march 94]. From different requirements, the Design Agent has determined that only a l4 engine can be used: (l4engine e1[starting 14 march 94]. Using domain knowledge that the v6engine and l4engine concepts are disjoint, the Management Agent will derive a contradiction.

The conflict management service tries to remove this contradiction by considering two properties of information, *credibility* and *deniability.*

Credibility is defined based on a postulated quasi order of agents in given roles. Any belief originating from an agent a in role r1 is more credible than a belief originating from agent b in role r2 iff (b r2) $<_c$ (a r1), where "$<_c$" is the quasi order. Credibility is an irreflexive and transitive order relation. We can assign[1] to each belief b a numerical value n(b) such that for any two comparable beliefs b1 and b2, b1 $<_c$ b2 $\leftrightarrow$ n(b1) < n(b2).

Deniability. Assuming that Marketing was first to determine that the engine must be a v6, it might have sent this information to the Purchasing Agent. The Purchasing Agent ordered v6 engines from another company. Later, Design discovered that the engine must be a l4. If the Design view is accepted, Purchasing may have troubles in cancelling the order (paying penalties, etc.). This shows that information that has been used for decision-making may be costly to retract later because of the cost of undoing previous decisions. We define the *undeniability* of consumed information as a measure of the cost to retract it (high undeniability means high costs). We often use *deniability* as the inverse of undeniability. Undeniability (or deniability) is determined by the consumers of information. We assume that agents are honest when assessing undeniability and do not use this to minimize their own workload. Figure 9 illustrates the types of beliefs distinguished among in conflict management.

Suppose we have determined a set $\{p_i\}$ of premises which supports a p&q=>false contradiction (in the sense that each p_i is part of an implication chain supporting either

1. By assigning 0 to the least credible beliefs and then incrementing the value for those directly preferred to these, a.s.o.

p or q). To each p_i we can attach: (i) a *credibility measure* - the numeric credibility previously introduced and (ii) an *undeniability measure* - derived from the sum of deniability costs of all propositions that would have to be retracted if p_i is retracted.

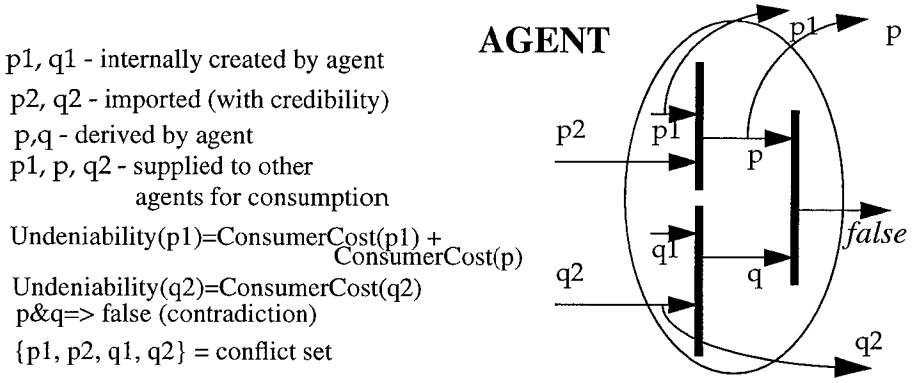


Fig. 9. Types of beliefs in conflict management.

A high credibility means that the proposition is more difficult to retract since a higher competence has to be contradicted. A high undeniability means that the proposition is more difficult to retract because the costs of retraction incurred upon consumer agents will be great. We can represent these two values in a diagram called a *c-u space*, as illustrated in figure 10.

Propositions from the *c-u* space that have both low credibility and low undeniability - as defined e.g. by some threshold values c_t and u_t - are easy to retract because they are not credible enough and do not incur significant costs. Propositions that have high credibility and high undeniability are hard to retract exactly for the opposite reasons. A measure of both credibility *and* undeniability is the distance *r* to the origin. If a proposition with high credibility and/or undeniability is considered for retraction, retraction must be negotiated with the producer and/or consumers.

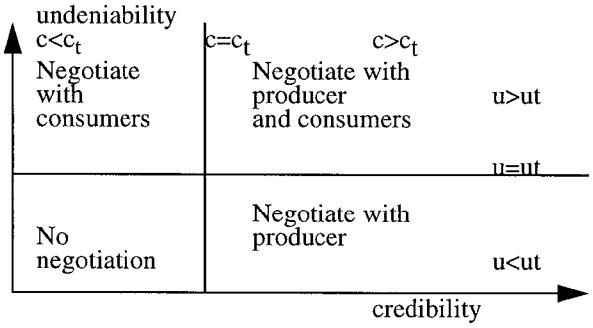


Fig. 10. Negotiation regions in the c-u space.

Figure 10 shows the regions defined by these thresholds in the c-u space. [14] and [15] are examples of work exploring negotiation as a means to mediate among conflicting agents.

7.2 The Model of Contradiction Removal

Given: a contradiction of the form: $p\&q=>false$

1. Determine the support set of p, that is the set of premises p is derived from, and the support set of q, that is the set of premises q is derived from. Together, these two sets form the conflict set.

2. Group the propositions from the conflict set into 4 sets corresponding to the 4 negotiation regions. In each region, order the component propositions in increasing order of the value of $r = (a^2+u^2)^{1/2}$

3. Considering the 4 regions in the order: (1) no-negotiation, (2) negotiation-with-producer, (3) negotiation-with-consumers, (4) negotiation-with-both, take each proposition in order and try to retract it:
   - If the premise falls in the no-negotiation region, retract it
   - If the premise falls into a negotiation region, negotiate with the required agents.

4. Retract the first proposition that passes the above tests and check if after retraction the contradiction can be rederived. If so, repeat the procedure. If no proposition can be retracted, report failure.

Initially, a conflict is discovered by an agent that derives a contradiction. The conflict management model then retracts one (or more) premises that led to the contradiction. Since the retracted premises can be themselves beliefs held by some other agents, their retraction may trigger other retractions in the agents holding them. These propagated retractions are performed using the same model - hence we have a propagated chain of retractions. Further work is needed to understand what to do if these chains are circular or too long.

This model of belief revision has three main advantages. First, it is accurate because the selection of the retracted belief is based on the views of all involved parties at the moment the contradiction is detected. Second, this means that the selection implicitly relies on domain knowledge and on the current state of the global problem-solving effort. Third, by estimating costs and identifying negotiation regions, the model takes advantage from situations in which negotiation may not be required or in which a smaller amount of negotiation may suffice.

8 Back to the Supply Chain

Going back to the supply chain, we implement the supply chain agents as COOL agents and devise coordination protocols appropriate for their tasks. Figure 11 shows the protocol that the Logistics agent executes to coordinate the entire supply chain. The process starts with the Customer agent sending a request for an order (see figures 3 and 4). Once Logistics receives the order, it tries to decompose it into activities like manufacturing, assembly, transportation, etc. This is done by applying a constraint based logistics scheduler. If decomposition is not possible, the process ends. Otherwise, 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.

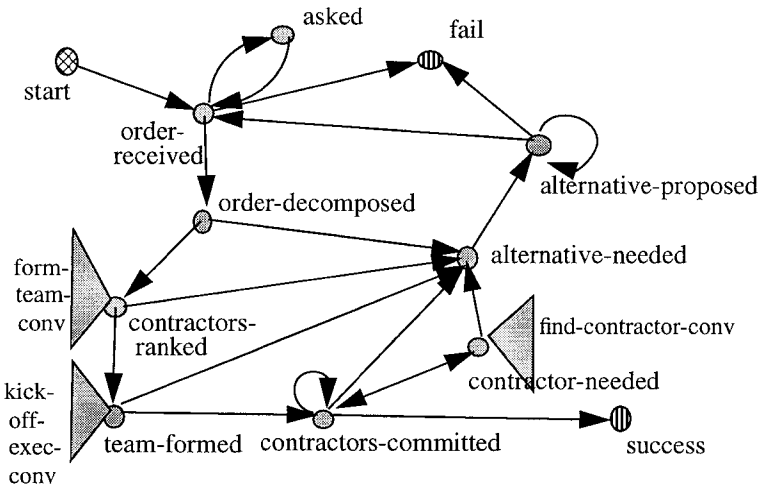


Fig. 11. Logistics execution protocol

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 state *contractors-ranked*. In this state, Logistics starts conversations with each contractor ranked for each activity, in the ranking order. The conversation shown in figure 11 is suspended and Logistics starts a team forming conversation with the potential contractors. If team forming is successful (one contractor found for each activity), then we move to state *team-formed*. Here, a new conversation with the team members kicks 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 ranked list (state *contractor-needed*). If that is not possible, an alternative contract is negotiated (*alternative-needed*).

In order to generate alternative contracts, Logistics relaxes some constraints from the initial order (for example moves a due-date further, changes a price etc.). For those agents using the constraint-based scheduling engine, this requires constraint relaxation and iterative use of the constraint-based scheduler, until the alternative is accepted by the customer. Any other tools, like databases, calendar management, project management, etc. are equally available to agents and humans in this process.

9 Related Work

ARCHON [9] is a general purpose architecture used to develop agent systems in real world domains like electricity distribution and supply. It supports large grain, loosely coupled, and semi-autonomous agents. In ARCHON cooperation knowledge had to be manually coded into a general representation language. We are trying to improve on that by coming with higher level specialized tools like COOL. We continue this

trend by providing reusable, application independent tools for other cooperative services related to information distribution and conflict management. *MACE* [11] is a testbed for distributed architectures that, like our system, includes symbolic representations of other agents as well as aggregation concepts like organizations. It does not however formalize coordination, information distribution and conflict management the way we do. *Agent oriented programming* [13] first formulated the notion of agents as objects with mental state, intentional communication based on speech acts and rule governed behavior. We are focusing on extending and making this view more practical by providing reusable tools. *Agent infrastructures for engineering* are aimed at bringing previously isolated engineering tools on-line. One set of solutions correspond to the *SHADE* [10] architecture: KIF [12] as the interlingua, KQML [6] as the speech act language, and the use of facilitator agents (like the SHADE Matchmaker) that match and route advertisements and subscriptions among the set of cooperating agents. Our shell supports facilitator-type services as generic services that any agent can provide if needed. We have gone further in building a coordination layer on top of KQML, making it much easier to capture and use complex coordination protocols.

10 Conclusions

At the core of the agent shell approach lies the recognition of the fact that practical multiagent systems engineering requires the ability to reuse trusted system components, services and languages. To achieve this goal we have developed a generic agent shell architecture that provides several layers of "agent level" languages and services. These include a novel coordination language and architecture, general representation substrates based on time augmented description logics, content based information distribution, organization modeling ontologies and novel consistency maintenance mechanisms.

Second, in line with recent work by Jennings [9] and others, we are exploring cooperation knowledge as a distinct level of knowledge, by building tools for modeling, acquiring and applying it to various tasks. Using these tools we have build cooperating agents in the framework of the Integrated Supply Chain, a major application area in manufacturing enterprise integration.

11 Acknowledgments

This research is supported, in part, by the Manufacturing Research Corporation of Ontario, Natural Science and Engineering Research Council, Digital Equipment Corp., Micro Electronics and Computer Research Corp., Spar Aerospace, Carnegie Group and Quintus Corp.

References

1. D. McAllester. Truth Maintenance. In *Proceedings AAAI-90,* pp. 1109-1116, 1990.

2. M. Barbuceanu. Models: Toward Integrated Knowledge Modeling Environments, *Knowledge Acquisition 5*, pp. 245-304, 1993.

3. M. Barbuceanu and M.S.Fox. The Architecture of a Generic Agent for Collaborative Enterprises. *EIL Working Paper*, nov. 1994.

4. A. Borgida, R.J. Brachman, D.L. McGuiness, L. Resnick. CLASSIC: A Structural Data Model for Objects. In *Proceedings 1989 ACM SIGMOD International Conference on Management of Data*, pp. 59-67, 1988.

5. R.J. Brachman, J.G. Schmolze. An Overview of the KL-ONE Knowledge Representation System. *Cognitive Science 9(2)*, pp. 171-216, 1985.

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

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

8. M. S. Fox, M. Barbuceanu, M. Gruninger. An Organisation Ontology for Enterprise Modeling: Preliminary Concepts for Linking Structure and Behavior. In *Proceedings of the Fourth Workshop on Enabling Technologies, Infrastructure for Collaborative Enterprises*, IEEE Computer Society Press, 1995.

9. N. R. Jennings. Controlling Cooperative Problem Solving in Industrial Multi-Agent Systems Using Joint Intentions. KEAG Tech. Report 93/25, Dept. of Electronic Engineering, Univ. of London, 1993.

10. D. Kuokka, J. McGuire, J. Weber, J. Tenenbaum, T. Gruber, G. Olsen. SHADE: Knowledge Based Technology for the Re-engineering Problem, Technical Report, Lockheed Artificial Intelligence Center, 1993.

11. L. Gasser, C. Braganza, and N. Herman. MACE: A Flexible Testbed for Distributed AI Research. In M.N. Huhns (ed), *Distributed Artificial Intelligence,* Pitman - Morgan Kaufman, pp 119-152, 1987.

12. M. R. Genesereth, R.E. Fikes. Knowledge Interchange Format, Version 3.0, Reference Manual, Computer Science Department, Stanford University, Technical Report Logic-92-1, 1992.

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

14. K. Sycara. Multi-agent compromise via negotiation. In Les Gasser and Michael N. Huhns, editors, *Distributed Artificial Intelligence, Volume II*, pp. 119-137, Pitman Publishing, London, 1989.

15. G. Zlotkin, J. S. Rosenschein. Negotiation and task sharing among autonomous agents in cooperative domains. In *Proceedings of IJCAI-89*, pp. 912-917, Detroit, MI, 1989.