COOL: A Language for Describing Coordination in Multi Agent Systems

Mihai Barbuceanu and **Mark S. Fox**
Enterprise Integration Laboratory,
Department of Industrial Engineering, University of Toronto,
4 Taddle Creek Road, Rosebrugh Building, Toronto, Ontario, M5S 1A4
email:**{mihai, msf}@ie.utoronto.ca**

Abstract

Agent interaction takes place at several levels. Current work in the ARPA Knowledge Sharing Effort has addressed the information content level by the KIF language and the intentional level by the KQML language. In this paper we address the coordination level by means of our Coordination Language (COOL) that relies on speech act based communication, but integrates it in a structured conversation framework that captures the coordination mechanisms agents use when working together. We are currently using this language (i) to represent coordination mechanisms for the supply chain of manufacturing enterprises modeled as intelligent agents and (ii) as an environment for designing and validating coordination protocols for multi-agent systems. This paper describes the basic elements of this language: conversation objects, conversation rules, error recovery rules, continuation rules, conversation nesting. The actual COOL source code and a running trace for the n-queens problem are presented in the Appendix.

Topic areas: Coordination, Intelligent agents in enterprise integration

1 Introduction

Coordinating the behavior of autonomous intelligent agents is a major concern of many application domains. Consider for example the problem of managing the supply chain of a manufacturing enterprise. The supply chain is a world-wide network of suppliers, factories, warehouses, distribution centres and retailers through which raw materials are acquired, transformed into products and delivered to customers. 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, political situations change overnight, 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.

Our research addresses coordination problems in the supply chain by organizing the supply chain as a network of cooperating, intelligent agents, [9, 15, 13] each performing one or more supply chain functions, and each coordinating their actions with other agents. Our focus is on supporting the construction of supply chain intelligent agent systems in a manner that guarantees that agents use the best communication and coordination mechanisms available with minimal programming effort on the developers' side. We achieve this goal (i) by developing communication and coordination tools allowing agents to cooperatively manage change and cooperatively reason to solve problems, (ii) developing ontologies that semantically unify agent communication, (iii) developing intelligent information infrastructures that keep agents consistently aware of relevant information and (iv) packaging the above theories into agent development tools that ensure that agents are able to reuse standardized coordination and reasoning mechanisms, relieving developers from the tedious process of implementing agents from scratch.

One major ingredient that we use in our work is an Agent Communcation Language (ACL) produced by the ARPA Knowledge Sharing Effort [10]. This language, known as Knowledge Query and Manipulation Language (KQML) [2, 3], provides a message format and message handling protocol supporting run-time knowledge sharing and interaction among agents. Interaction is however more than exchanging messages. One aspect of interaction that we strongly require refers to coordination protocols, that is the shared conventions about message exchange that agents use when working together in a coordinated fashion. The goal of this paper is to present a language for describing such coordination protocols which makes use of KQML (or KQML-type languages) at the communication level. The language, named COOL (for COOrdination Language), has been implemented and is currenly used in our distributed supply chain project to model coordination mechanisms among agents. The paper presents the major elements of the language, including conversation objects, conversation rules, error recovery rules, continuation rules, conversation nesting. The appendix provides a COOL specification of the n-queens problem and a trace of how the problem is solved by coordination among agents (queens).

2 Levels of Agent Interaction

Agent interaction takes place at several levels. The first level is concerned with the *information content* communicated among agents. A piece of information communicated at this level may be a proposition (fact) like "(produce 200 widgets)". The ARPA Knowledge Sharing Effort has produced the KIF [5] logic language for describing the information content transmitted and the conceptual vocabularies (or ontologies [6]) communicating agents must share in order to understand each other.

The second level specifies the *intentions* of agents. The same information content can be communicated with different intentions. For example:

- (*ask* (produce 200 widgets)) - the sender asks the receiver if the mentioned fact is true,
- (*tell* (produce 200 widgets)) - the sender communicates a belief of his to the receiver,
- (*achieve* (produce 200 widgets)) - the sender requests the receiver to make the fact one of his beliefs
- (*deny* (produce 200 widgets)) - the sender communicates that a fact is no longer believed.

KQML [2, 3] has been designed as a universal language for expressing such intentions such that all agents would interpret them identically. KQML supports communication through explicit linguistic actions, called *performatives*. As such, KQML relies on the speech act [12] framework developed by philosophers and linguists to account for human communication. Work is currently being done [8] for endowing KQML with formal semantics based on the speech-act theory as formalized and extended within the fields of Computational Linguistics and Artificial Intelligence [1].

The third level is concerned with the *conventions* that agents share when interacting by exchanging messages. The existence of shared conventions makes it possible for agents to coordinate [7, 17] in complex ways, e.g. by carying out negotiations [14, 18] about their goals and actions. As an example, consider the supply chain of our TOVE virtual manufacturing enterprise [4, 11] as a multi-agent system. The Order Acquisition Agent interacts with the customer and acquires an order for 200 lamps with a due date for 28 sept 94. It sends this as a *proposal* to the Logistics Agent. Knowing that Logistics can only answer with *accepting*, *rejecting* or *counter-proposing*, Order Acquisition is able to check that the actual response is one of these and carry out a corrective dialogue with Logistics if this is not the case or if other events occur (such as delays or message shuffling). If Logistics answers with a counter-proposal (e.g. 200 lamps with due date 15 oct 94), Order Acquisition may use knowledge about acceptable trade-offs and negociate with Logistics an amount and a due-date that can be achieved and satisfies the customer. In its turn, upon receiving the order proposal, Logistics will start negotiations with the Scheduling agent to determine the feasibility of scheduling the production of the order and with the Transportation agent to determine feasibility of the delivery date.

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

Finally, a fourth level of interaction is concerned with how *agents* are modeled, (e.g. which are their beliefs, goals, authorities etc. in the organizations they are part of). We address this aspect by building organizational models and representing the agents as components of these organizations. This work will be reported elsewhere.

3 COOL: A Language Layer for Defining Coordination Models And Protocols

In any multi-agent system, the coordination level must be explicitely captured in order to have agents cooperate in non-trivial ways. We model the coordination level by means of a coordination language that is used in particular for describing coordination in the supply chain of the TOVE enterprise and in general as a coordination specification language for any multi-agent system.

3.1 Basic components

We model a coordination activity as a conversation among two or more agents, specified by means of a finite state machine (FSM):

- The states of the FSM represent the *states* a conversation can be in. There is a distinguished *initial* state any conversation starts in, and several *terminating* states that when reached signal the termination of the conversation.
- The messages exchanged are represented as *performatives* (speech acts) of the agent communication language. The content level of performatives is not part of the negotiation protocol, but determines the course of an individual negotiation as it is used in the decision-making of agents.
- A set of *conversation rules* specify how 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.
- A set of *error recovery rules* specify how incompatibilities among the state of a conversation and the incoming messages are handled.
- A set of *continuation rules* specify how agents accept requests for new conversations or select a conversation to continue from among the existing ones.
- *Conversation classes* specify the states, conversation rules and error rules that are specific to a type of conversation. An agent has several conversation classes it can use when communicating with other agents.
- *Actual conversations* instantiate conversation classes and are created whenever agents engage in communication.

These elements are described in detail in the remainder of this section.

3.2 Speech acts

Agents cooperate and coordinate through communication. We assume the existence of a standard set of speech acts that define the communicative actions available within an organization. These speech acts are represented as performatives of the agent communication language. To the standard speech acts provided by our agent communication language - KQML - we have added a number of higher order speech acts like:

- *Propose*. This is used to propose to an agent a subgoal to achieve. For example:
  `(propose :content (produce (widgets 200)(time “19-sep-94”))).`
- *Counter-Propose.* A counter proposal is another subgoal that partially satisfies the initial goal of a propose. The use of this speech act can result in a sequence of counter-proposals from both the original proposer and the respondent. An example is:

```
(counter-propose :content (produce (widgets 200)(time "20-sep-
94"))).
```

- *Accept and Reject*. These are used to signal acceptance, respectively rejection of a proposal or counter-proposal. Rejection starts a new negotiation phase.
- *Cancel.* This cancels a previously accepted proposal or counter-proposal.
- *Commit*. Positive confirmation that an agent puts itself in a state that will satisfy a proposal; also ends a negotiation or renegotiation phase.
- *De-Commit*. Cancellation of a previous Commit.
- *Satisfy.* An agent announces that a requested goal has been achieved. For example:

  ```
  (satisfy :content (counter-propose :content (produce (widgets
  200)(time "20-sep-94")))
  ```

- *Fail*. An agent informs that execution of a committed goal has failed.

3.3 FSM specification

The models of coordination involving speech acts like the above ones are described with finite state machines. An example is shown in figure 1.

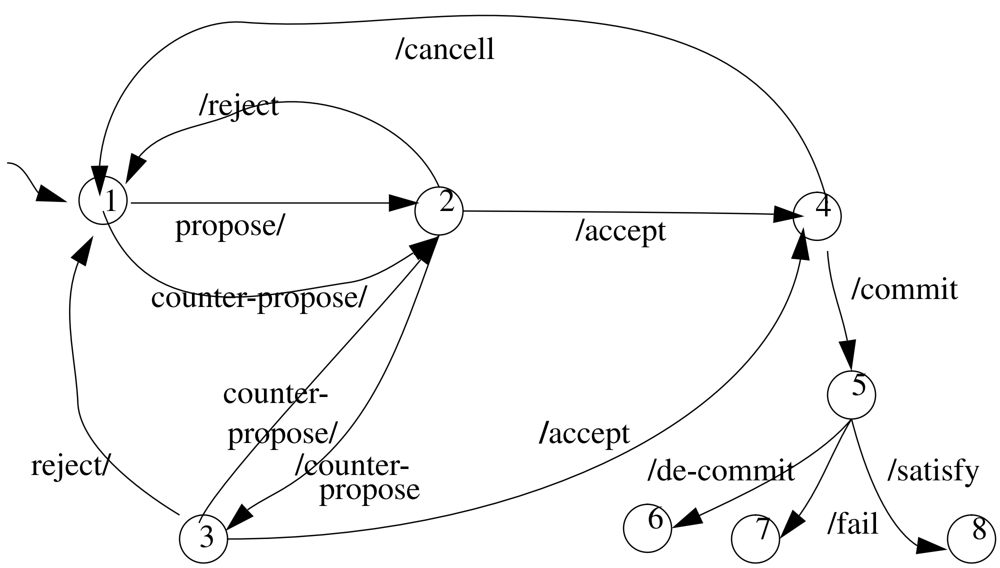

FIGURE 1. State transitions for negotiation.

Figure 1 uses the notation <received speech act.> / <sent speech act> to label edges in the diagram. When the conversation is in a given state and if a speech act is received, the agent performs local processing (not shown), sends out the shown speech act and switches to the state pointed to by the edge. State 1 is the initial state and states 6, 7 and 8 are final states. A conversation starts in state 1 with the agent receiving a speech act. If it is a proposal, the agent switches to state 2. In this state it may reject the proposal, or issue a counter-proposal. If a counter-proposal is issued (state 3), it may be accepted by the interlocutor, or rejected. If it is rejected, another propose counter-propose cycle may start. If a (counter-) proposal is finally accepted, the conversation switches to state 4. Here the agent may change its mind (cancel) or commit to achieving the subgoal. In state 5, the agent may report satisfaction of commitment, failure or unilateral de-commitment.

This example of conversation - and coordination - model describes message exchange from the viewpoint of the agent that is satisfying requests from other agents. From the viewpoint of the agent that makes the request in the first place the conversation model will be different.

Conversation models of this kind are hence described from the viewpoint of one participating agent. The other participating agents have their own models and this poses difficult problems for the verification/validation of conversations (dead-locks, message shuffling, etc.). VonMartial [16] describes techniques for designing consistent asynchronous conversations described by FSMs.

3.4 Conversation rules

Conversation rules specify agents' reasoning for choosing the next edge in the diagram, what internal processing they do when switching states and what messages they send out. A rule example is the following:

```
(def-conversation-rule r1
  :current-state 2
  :received
   (propose :sender ?initiator :content (produce (?what ?amount)(time
  ?date)))
  :such-that
  (and (achievable (produce ?what ?amount))
   (not-achievable (time ?date))
   (possible-alternative (time ?date) (time ?date1)))
  :next-state 3
  :transmit (counter-propose :content (produce (?what ?amount) (time
  ?date1))))
```

Rules have local variables that are unified when rules are applied. Above, `?what`, `?amount`, `?date` and `?date1` are such variables. Besides these local variables, there exists a persistent conversation environment that provides persistent variables that can be used to transmit values between rule firings. For example, the initiator of a conversation is stored in such a variable (`?initiator` above) and can be used in rules.

To test for incoming messages, conversation rules provide constructs for testing the first message in the queue - considering the queue ordered - or to look for messages anywhere in the queue - considering the queue as a set. The latter is useful as often the communication services can not deliver messages in the order they were sent. It is also possible to check in a rule for the ordered or unordered occurence of more than one message. This allows for better handling of messages from several agents in conversations involving more than two participants.

3.5 Error recovery rules

A situation that can easily occur is that the current message received in a conversation can not be handled by any of the rules in the current state. This signals an error that can have many causes - message delays, destroyed message order, lost messages, wrong messages sent out, etc.

Agents cope with this situation at two levels. First, they can use more elaborate conversation structure and rules that take such possibilities into account. Second, they can invoke a set of error recovery rules associated with each conversation. Error recovery rules may perform any action deemed appropriate, such as discarding inputs, initiating clarification conversations with the interlocutor, changing the state of the conversation, or just reporting the error and terminating the conversation. The advantage of error recovery rules is that they allow complex error recovery policies to be explicitly designed and (re)used among many agents and conversations.

3.6 Defining conversations

We distinguish among conversation *classes* and *actual* conversations. A conversation class specifies the states, variables, conversation rules, error recovery rules and control mechanisms that apply these kinds of rules. An actual conversation is an instance of a conversation class. There can be many actual conversations instantiating the same conversation class (for different agents and different states and messages exchanged).

The linguistic construct we use to define conversation-classes bundles together the above elements. An example conversation class definition is:

```
(def-conversation-class Cnv-1
  :initiator ?initiator
  :respondent ?respondent
  :variables (?v1 ?v2)
  :initial-state s0
  :final-states (s5 s7)
  :conversation-rules ((s0 r1 r2) ...)
  :conversation-rule-applier CRA-1
  :error-rules (e1 e2 ...)
  :error-rule-applier ERA-1)
```

In this definition, `:initiator` and `:respondent` are slots holding the names (and possibly initial values) of distinguished persistent variables. `:conversation-rules` and `:error-rules` hold the corresponding sets of rules governing the conversation (note that conversation rules are indexed on the state they apply to). `:conversation-rule-`

applier and :error-rule-applier hold the functions that apply the two kinds of rules.

3.7 Initiating conversations

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

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

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

Our current mechanism for retrieving the conversation class that will manage a request for a new conversation is based on two elements. First, any message that is a request for conversation must have an aditional slot :intent slot (another - and last - extension to KQML) that contains a description of the intent of the requesting agent. The receiving agent tries to find a conversation class that matches the expressed :intent of the sender. This is done by having conversation classes specify an :intent-test predicate that will be used with the actual intent as argument. If the test determines that a conversation class can serve the :intent of a request, then the second element is used. This is a verification that in the initial state of the selected conversation class there exists at least one rule that can be triggered by the received message. If this is the case, a new (actual) conversation controlled by the retrieved conversation class is created and the receiver agent will use it as its conversation with the sender.

3.8 Interrupting conversations

The need to interrupt ongoing conversations arises from two major reasons. First, an agent a that has an ongoing conversation with an agent b may need sometime during that conversation to start a new conversation with an agent c. For example this may be required to acquire information, to achieve a goal or to correct an error. Second, an agent a having a conversation with an agent b may be interrupted during this conversation by a request from an agent c.

To allow for these situations, we let each agent have a set of ongoing conversations. When an agent initiates a new conversation, the new conversation object is added to this set. When a conversation has to be interrupted because another conversation must take place, the old conversation is suspended, and the system marks the suspended conversation as waiting for the new conversation to complete. This creates dependency records among conversations that are used when selecting the next conversation to work on. Because conversation objects can be inspected, the states and variable values of a conversation that another conversation waits for can be used by the waiting conversation when the latter is resumed.

For example, consider again the supply chain of an enterprise organized as a multi-agent system. The Order Acquisition Agent may have a conversation with the Logistics Agent about a new order. The Logistics Agent may temporarily suspend this conversation to start a conversation with the Scheduling Agent to inquire about the feasibility of a due date. Having obtained this information, the Logistics Agent will resume the interrupted conversation with Order Acquisition.

3.9 Continuation rules

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

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

Continuation rules perform two functions. First, they test the input queue of the agent and apply the conversation class recognition mechanism (section 3.7) to initiate new conversations. Second, they test the data base of ongoing conversations and select one existing conversation to execute.

Which of these two actions has priority (serving new requests versus continuing existing conversations) and which request or conversation is actually selected, is represented in the set of continuation rules associated to the agent. Our agent definition mechanism allows the specification, for each agent, of both the set of continuation rules and the continuation rule applier.

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

```
(def-continuation-rule cont-1

:input-queue-test
```

```
(lambda(queue)
 (if queue (exists-conv-class-initially-accepting (first queue))
nil))).
```

3.10 Defining agents

The last element of the framework is a simulation environment that allows defining the agents in the system, their conversation classes, actual conversations and continuation rules. A number of functions are provided that allow agent systems thus defined to be simulated by managing message passing and the activation of individual agents. The n-queens solution in the Appendix uses this facility.

4 Concluding remarks

Agent interaction takes place at several levels. Current work has addressed the information content level by the KIF language, the intentional level by the KQML language (both outcomes of the ARPA Knowledge Sharing Effort). We propose to address the coordination level by the COOL language that relies on speech act based communication, but integrates it in a structured conversation framework that captures the coordination mechanisms agent used when working together. COOL provides constructs for describing:

- structured conversations (as finite state machines),
- conversation rules for describing the state transitions within a conversation,
- error rules for specifying corrective actions to take when unexpected, delayed or otherwise perturbed communication occurs,
- continuation rules for allowing agents to define their own policies for selecting which conversation to continue,
- a mechanism for managing multiple conversations of a single agent, by maintaining dependencies among conversations (such as having one conversation wait for another to reach a given status).

We are currently using this language to represent coordination mechanisms for the supply chain of manufacturing enterprises modeled as intelligent agents. Because of the simulation capabilities of our implementation, we are also using the language as an environment for designing and validating coordination protocols, without having to build actual agent systems for this purpose.

The Appendix presents the actual COOL source code and a running trace for the n-queens problem described as a coordination problem in the sense of our language.

5 Acknowledgements

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.

6 References

1. Cohen, P., Morgan, J., Pollack, M. (editors) Intentions in Communication, MIT Press Cambridge, MA, 1990.

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

3. Finin, T., Fritzson, R., McKay, D. and McEntire, R., KQML - An Information and Knowledge Exchange Protocol, in Kazuhiro Fuchi and Toshio Yokoi, editors, Knowledge Building and Knowledge Sharing, Ohmsha and IOS Press, 1994.

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

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

6. Gruber, T. R., Toward principles for the design of ontologies used for knowledge sharing, Report KSL 93-04, Stanford University, august 1993.

7. Jennings, N., R., Commitments and conventions: The foundation of coordination in multi-agent systems, The Knowledge Engineering Review, vol. 8:3, pp 223-250, 1993.

8. Labrou, Y. and Finin, T., A Semantics Approach for KQML - A General Purpose Communication Language for Software Agents, 1993.

9. Pan, J.Y.C., Tenenbaum, J. M. An Intelligent Agent Framework for Enterprise Integration, IEEE Transactions on Systems, Man and Cybernetics, 21, 6, pp. 1391-1408, 1991.

10. Patil, R., Fikes, R., Patel-Schneider, P., McKay, D., Finin, T., Gruber, T., and Neches, R. The ARPA Knowledge Sharing Effort: Progress report. In B. Nebel, C. Rich, and W. Swartout, editors, Principles of Knowledge Representation and Reasoning: Proceedings of the Third International Conference (KR'92), San Mateo, CA, Nov. 1992. Morgan Kaufmann.

11. Roboam, M., Fox, M. S. Enterprise Management Network Architecture, A Tool for Manufacturing Enterprise Integration, Artificial Intelligence Applications in Manufacturing, AAAI Press/MIT Press, 1992.

12. Searle, J. Speech Acts, Cambridge University Press, Cambridge, UK, 1969.

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

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

15. Tenenbaum, J. M., Gruber, T. R., Weber, J.C. Lessons from SHADE and PACT, Enterprise Modeling and Integration, C. Petrie (ed), McGraw-Hill 1992.

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

17. Winograd, T. and Flores, F. (1986) Understanding Computers and Cognition: A New Foundation for Design, Ablex Publishers, 1986.

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

7 Appendix: COOL solution to the n queens problem

;;; Notes:
;;; - assume 4 queens
;;; - ?x, ?y, etc. denote variables
;;; - (?(function-name arg1 arg2 ...)) denotes an evaluable expression that is replaced
;;; by its value in the body of a transmitted message. Any variables used in the
;;; expression are first replaced by their values.

;; 1. the 4 queens as agents

```
(def-agent 'q1)
(def-agent 'q2)
(def-agent 'q3)
(def-agent 'q4)
```

;; 2. the conversation for the leftmost (first) queen

```
(def-conversation-class 'qc-1
 :name 'first-queen-class
 :content-language 'list
 :speech-act-language 'kqml
 :initial-state 's0
 :final-states '(yes no))
```

;; 3. the conversation for the middle queens (no matter how many)

```
(def-conversation-class 'qc-2
 :name 'middle-queen-class
 :content-language 'list
 :speech-act-language 'kqml
 :initial-state 's0
 :final-states '(yes))
```

;; 4. the conversation for the rightmost (last) queen

```
(def-conversation-class 'qc-3
 :name 'last-queen-class
 :content-language 'list
 :speech-act-language 'kqml
 :initial-state 's0
 :final-states '(yes))
```

;; 5. rules for first-queen-class

```
(def-conversation-rule 'r11
 :name 'r1
 :current-state 's0
```

```
 :next-state 's1
 :transmit
       '(propose :sender q1
        :receiver q2
        :content (?(choose-new-position ?agent nil))
        :conversation c1))

(def-conversation-rule 'r12
 :name 'r2
 :current-state 's1
 :received '(reject :sender q2 :content ?c)
 :such-that '(another-position-exists ?agent ?c)
 :next-state 's1
 :transmit
        '(propose
        :sender q1
        :receiver q2
        :content (?(choose-another-position ?agent ?c))
        :conversation c1))

(def-conversation-rule 'r13
 :name 'r3
 :current-state 's1
 :received '(reject :sender q2 :content ?c)
 :such-that '(not(another-position-exists ?agent ?c))
 :next-state 'no
 :do '(format t "~%;;; No solution possible"))

(def-conversation-rule 'r14
 :name 'r4
 :current-state 's1
 :received '(accept :sender q2 :content ?c)
 :next-state 'yes
 :do '(format t "~%;;; Solution found ~s" ?c))
```

;; 6. rules for middle-queen-class

```
(def-conversation-rule 'r21
 :name 'r1
 :current-state 's0
 :received '(propose :sender ?s
 :content ?c
 :conversation ?conv)
 :such-that
        '(and (at-left ?s ?agent)
        (not(new-position-exists ?agent ?c)))
 :next-state 's0
 :transmit '(reject :sender ?agent
 :receiver ?s
 :content ?c
 :conversation ?conv))

(def-conversation-rule 'r22
```

```
 :name 'r2
 :current-state 's0
 :received '(propose :sender ?s :content ?c
 :conversation ?conv)
 :such-that
        '(and(at-left ?s ?agent)
        (new-position-exists ?agent ?c))
 :next-state 's1
 :transmit
       '(propose :sender ?agent
        :receiver (?(right-of ?agent))
        :content (?(choose-new-position ?agent ?c))
        :conversation ?conv))

(def-conversation-rule 'r23
 :name 'r3
 :current-state 's1
 :received '(accept :sender ?s :content ?c :conversation ?conv)
 :such-that '(at-right ?s ?agent)
 :next-state 'yes
 :transmit
        '(accept :sender ?agent
        :content ?c
        :receiver (?(left-of ?agent))
        :conversation ?conv))

(def-conversation-rule 'r24
 :name 'r4
 :current-state 's1
 :received '(reject :sender ?s :content ?c :conversation ?conv)
 :such-that
        '(and(at-right ?s ?agent)
        (another-position-exists ?agent ?c))
 :next-state 's1
 :transmit
        '(propose :sender ?agent
        :receiver (?(right-of ?agent))
        :content (?(choose-another-position ?agent ?c))
        :conversation ?conv))

(def-conversation-rule 'r25
 :name 'r5
 :current-state 's1
 :received '(reject :sender ?s :content ?c :conversation ?conv)
 :such-that
        '(and(at-right ?s ?agent)
        (not(another-position-exists ?agent ?c)))
 :next-state 's0
 :transmit
        '(reject :sender ?agent
        :receiver (?(left-of ?agent))
        :content (?(remove-last ?c))
        :conversation ?conv))
```

;; 7. rules for last-queen-class

```
(def-conversation-rule 'r31
 :name 'r1
 :current-state 's0
 :received '(propose :sender ?s :content ?c :conversation ?conv)
 :such-that
        '(and (at-left ?s ?agent)
        (new-position-exists ?agent ?c))
 :next-state 'yes
 :transmit
        '(accept :sender ?agent
        :receiver (?(left-of ?agent))
        :content (?(choose-new-position ?agent ?c))
        :conversation ?conv))

(def-conversation-rule 'r32
 :name 'r2
 :current-state 's0
 :received '(propose :sender ?s :content ?c :conversation ?conv)
 :such-that
        '(and(at-left ?s ?agent)
        (not(new-position-exists ?agent ?c)))
 :next-state 's0
 :transmit
        '(reject :sender ?agent
        :receiver (?(left-of ?agent))
        :content ?c
        :conversation ?conv))
```

;;; Execution trace - exchanged messages

```
;;; (PROPOSE :SENDER Q1 :RECEIVER Q2 :CONTENT (0) :CONVERSATION C1)
;;; (PROPOSE :SENDER Q2 :RECEIVER Q3 :CONTENT (0 2) :CONVERSATION C1)
;;; (REJECT :SENDER Q3 :RECEIVER Q2 :CONTENT (0 2) :CONVERSATION C1)
;;; (PROPOSE :SENDER Q2 :RECEIVER Q3 :CONTENT (0 3) :CONVERSATION C1)
;;; (PROPOSE :SENDER Q3 :RECEIVER Q4 :CONTENT (0 3 1) :CONVERSATION
C1)
;;; (REJECT :SENDER Q4 :RECEIVER Q3 :CONTENT (0 3 1) :CONVERSATION C1)
;;; (REJECT :SENDER Q3 :RECEIVER Q2 :CONTENT (0 3) :CONVERSATION C1)
;;; (REJECT :SENDER Q2 :RECEIVER Q1 :CONTENT (0) :CONVERSATION C1)
;;; (PROPOSE :SENDER Q1 :RECEIVER Q2 :CONTENT (1) :CONVERSATION C1)
;;; (PROPOSE :SENDER Q2 :RECEIVER Q3 :CONTENT (1 3) :CONVERSATION C1)
;;; (PROPOSE :SENDER Q3 :RECEIVER Q4 :CONTENT (1 3 0) :CONVERSATION
C1)
;;; (ACCEPT :SENDER Q4 :RECEIVER Q3 :CONTENT (1 3 0 2) :CONVERSATION
C1)
;;; (ACCEPT :SENDER Q3 :CONTENT (1 3 0 2) :RECEIVER Q2 :CONVERSATION
C1)
;;; (ACCEPT :SENDER Q2 :CONTENT (1 3 0 2) :RECEIVER Q1 :CONVERSATION
C1)
;;; Solution found (1 3 0 2)
;;; No agent can be activated
```