University of Toronto
Enterprise Integration Laboratory

COOL: A Language for Representing and Executing Coordinated Behavior in Multi-Agent Systems

User Manual

Mihai Barbuceanu

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

mihai@ie.utoronto.ca
http://www.ie.utoronto.ca/EIL/ABS-page/ABS-intro

April 1996

1 Introduction

In previous reports we have explained the basic notions of the language and provided examples for their use in applications like Supply Chain Integration. Here, we provide a more formal specification of the language. We distinguish between the language and its possible implementations in various computational environments. Our current prototype implementation is in Winterp, a C-based object oriented interpretive language integrating Motif and Lisp. Other implementations are equally possible. The current implementation provides a lot of expressivity and flexibility somewhat at the expense of computational efficiency. We believe that this trade-off is reasonable, because the computational efficiency of the coordination component plays a small part in the overall efficiency of a multi-agent system. The dominant part stems from the legacy applications integrated in the system and from communication issues.

Other possible alternative implementations can be developed as object libraries in languages like C++ or SmallTalk. As shown in our reports about interactive debugging and capturing of coordination knowledge, we also rely on extensive visual interfaces for monitoring and intervening into sytem operation at various levels and moments. Implementations that can make good use of GUI packages and systems will be able to support these important concepts as well.

From the viewpoint of communication, we currently use TCP/IP as provided by our host Winterp system. Alternatives might include CORBA, especially if the rest of the system is written in a CORBA supported OO language.

Whatever the implementation we strongly recommend having a top level interpretive loop (like the read-eval-print in Lisp) that would enable interactive access to the system. Depending on the implementation, a lot of things that are taken for granted in the Winterp version - like dynamic data structures (lists), flexible string manipulation, source level debugging, very easy parsing, automatic memory management - may become more difficult to achieve and increase the cost of the implementation.

2 Basic elements

1. A **keyword** is any symbol starting with a ':'.

Examples: :a, :xxx123

2. **Symbol**: any sequence of letters and digits starting with a letter.

Examples: a, A, Abc, X123

3. **Numbers**: we only use integers in COOL, they have the usual appearance.

Examples: 1, 123, 777.

4. **Strings**: any sequence of characters enclosed between "-s

Example: "123", "byebyebye"

5. **Variable**: a symbol starting with '?' or with '??'

Examples: ?sss, ?x, ?v1, ??u, ??v1.

6. **Lists**:

<list>::= (<element>*)

<element>::= symbol | number | keyword | string | <list>

Examples:

(this IS a LisT with 7 elements)

(the first element of this list is "the")

(1 2 3 :first 1 :second 2 :last “this”)

((nested list) as the first and (the last)))

(((((very) many) nested) words) here)

6. **T** and **nil**.

T is the reserved symbol standing for the *true* logical value. nil is the reserved symbol standing for the *false* logical value. Anything different from nil is also considered *true*.

2.1 Pattern of COOL constructs

COOL constructs defining agents, rules, etc. follow the syntax pattern shown bellow:

(<construct-type> <required arguments> <optional arguments>).

Here we have:

<construct-type> ::= symbol

<required arguments>::= list of arguments

<optional arguments>::= list of keyword value pairs, where value is any <element>

3 Agents

3.1 Syntax

<agent-definition>::=

(def-agent <name>

:conversation-classes <list of conversation classes>
:conversations <list of conversations>
:continuation-rules <list of continuation-rules>
:continuation-rules-incomplete <T or nil>
:continuation-control <name of control fn>[default: agent-control]

)

- <name> is the global name of the defined agent in the COOL environment. Different agents must have different names.
- <list of conversation classes> is the list of names of the conversation classes this agent has. In order for an agent to use a conversation class, it must had been declared in this way. The conversation classes must had been declared before use. Alternatively, to avoid problems of forward referencing, COOL provides a def-associations construct allowing one to declare conversation classes for an agent at the end of the program, after the agent and the conversation classes have been declared.
- <list of conversations> list of actual conversations of an agent. Must had been defined before use, but to avoid referencing problems the def-associations construct can be used.
- <list of continuation-rules> list of continuation rules for this agent. Same observations as for conversation classes wrt. prior declaration and tye use of def-associations.
- <T or nil> flag specifying whether the set of continuation rules of this agent is complete or not. Note that this flag marks the entire set as incomplete, so even if every individual rule is complete the knowledge acquisition interface for continuation rules will still be popped up (under the assumption that new rules may be added or existing ones deleted).

- <name of control fn>[default: agent-control] This is the name of the pluggable function used as the interpreter of continuation rules. It can be defaulted to a standard interpreter like agent-control, provided with the language implementation.

3.2 Semantic issues

The general convention in COOL is that one can not reference constructs before declaring them. To avoid the usual problem related with this, we provide a def-associations form that can perform any association in the language, including the association of conversation-classes and continuation-rules to agents. These associations are usually described at the end of the program, when all components have been declared.

All constructs (agents, conversation classes, conversations, the rules) have unique global names in COOL. These names have to be used in declarations or associations.

As an implementation hint, it is useful to implement agents as objects in an underlying OO language on top of which COOL is built. Reading the above declaration of an agent should trigger the creation of an object storing the attributes of an agent. In our prototype implementation, reading the above declaration instantiates a system provided agent class.

3.3 Examples

The following examples are from the supply-chain application. Note the quotation required by the reader of the underlying Lisp implementation and the use of a non-standard continuation control function (in this case the one calling the knowledge acquisition interface)

```
(def-agent 'customer :continuation-control 'agent-control-ka)

(def-agent 'logistics :continuation-control 'agent-control-ka)

(def-agent 'plant1 :continuation-control 'agent-control-ka) ; assembly

(def-agent 'plant2 :continuation-control 'agent-control-ka) ; paint

(def-agent 'plant3 :continuation-control 'agent-control-ka)

(def-agent 'transp1 :continuation-control 'agent-control-ka)

(def-agent 'transp2 :continuation-control 'agent-control-ka)
```

4 Conversation Managers

4.1 Syntax

```
(def-conversation-manager <name>

  :agents <list of agents>
  :trace-agent <list of traced agents>
  :trace-message <list of traced messages>
  :trace-conv <list of traced connversations>
  :trace-conv-rule <list of traced conversation rules>
  :trace-err-rule <list of traced error recovery rules>
  :trace-cont-rule <list of traced continuation rules>
  :trace-conv-class <list of traced conversation classes>

)
```

This declares a conversation manager having the unique global name <name>. The other arguments describe various tracing options. These options belong more to the particular implementation,

but for orientation should always include message tracing which is of major interest. See the examples section for examples of tracing specifications and the source code of the applications.

4.2 Semantics

The purpose of the conversation manager is to specify which agents are controlled by a manager. It is possible in COOL to define several conversation managers, each with its own set of agents. The need for the conversationmanager occurs as agents in a COOL environment are not separate processes, hence have to be run individually by some controlling program. We are working on removing this limitation and allowing agents that are separate Unix processes. The conversation-manager will still be need if there exist agents that are not separate processes and we believe that is useful in cases when creating a separate process would be to much of an overhead.

4.3 Examples

(def-conversation-manager 'm1)

As an alternative way of requesting tracing, one can use separate functions as illustrated in the following example from the nquuens application:

(trace-agent m1 q1 q2 q3 q4)

(trace-conv m1 '((,q1 c1 first-queen-class)(,q2 c1 middle-queen-class)(,q3 c1 middle-queen-class)(,q4 c1 last-queen-class)))

(trace-conv m1 q2)

(trace-conv m1 q3)

(trace-conv m1 q4)

(trace-message m1 q1 q2 q3 q4)

Above, q1 q2 q3 and q4 are the agents (queens), first-queen-class, middle-queen-class and last-queen-class are conversation classes and c1 is the single conversation going on. See the source code for details.

5 Conversation Classes

5.1 Syntax

(def-conversation-class <name>

:content-language <language name>
:speech-act-language <language name>
:ontology <ontology name>
:rules <list of conversation rules>
:rules-incomplete <T or nil>
:control <conversation rule control fn>[default interactive-choice-control]
:initial-state <state name>
:final states <list of state names>
:variables <list of variables>
:recovery-rules <list of recovery rule names>
:recovery-rules-incomplete <T or nil flag>
:recovery-control <recovery rules control fn>[default recovery-control]
:intent-check <intent check fn>

)

- <name> is the unique name of the conversation class
- <language name> is the name of the content language or the speech-act-language.

The speech-act-language is usually KQML. The content language is at the discretion of the user. If the :content-language is specified, the system will automatically insert it in the KQML messages that are received or sent by the conversation.

- <ontology name> is the ontology used by the conversation. If given, it will be automatically inserted into messages received or sent by any conversation described by this class. This simplifies writing the message patterns for received messages and the output messages.
- <list of conversation rules> list of conversation rule unique names for this conversation class. To avoid problems of forward referencing def-associations can be used.
- <T or nil> value of a flag that specifies the incomplete status of a rule. Value T means the rule is incomplete, nil means the rule is complete.
- <conversation rule control fn> name of a function used as the interpreter for conversation rules. This is a pluggable interpreter and is defaulted by the system. The default interpreters should be accessible as the value of some parameter that users can set at will.
- <state name> for the initial state, theis is its name. This is the state a conversation starts in.
- <list of state names> list of states. For the final states, when the conversation reaches any of these states it will terminate.
- <variables> list of variables of this conversation class. It is not necessary to declare variables in advance, they can be created dynamically, when and if needed. This declaration is useful especially to document the purpose and use of variables in a conversation class. Remember that variable names start with '?' or '??'.
- <recovery rules> list of unique names of the recovery rules used in the conversation.
- <recovery rules control fn> name of pluggable interpreter of recovery rules. Defaulted by various system configurations.
- <intent check> the predicate that will be applied to the value of the :intent slot of an incoming message to determine if this class can handle the incoming message. Can be function name, lambda expression or something else (depending on what the host language allows).

5.2 Semantic issues

Note that the conversation class specifies only the initial and final states of a conversation. All intermediate states are specified in the conversation rules as the :next-state parameter. This gives flexibility in modifying the conversation class (important as many modifications occur on the fly, as the system is running).

For both conversation and recovery rules, subsequent use of def-associations can avoid the problem of forward references.

In a standard implementation, the above declaration would be translated into an object that would hold all the attributes of the conversation class.

Currently, the <variables> are instance variables (they are stored in each actual conversation). A future version may introduce class variables as well (whose value is stored in the conversation class itself and inherited by all actual conversations). Variables are persistently stored for the duration of the conversation.

5.3 Examples

(def-conversation-class 'logistics-execution-net

```
:name 'logistics-execution-net
:content-language 'list
:speech-act-language 'kqml
:initial-state 'start
:final-states '(fail success)
:control 'interactive-choice-control-ka
:variables
  '(?order                          ; the received order
   ?alt-order          ; another order the agent might have proposed to the customer
   ?activities                      ; (activity1 activity2 ...)
   ?ranked-contractors              ;((activity ag1 ag2 ...)...)
   ?large-team                      ;((activity ag1 ag2 ...)...)
   ?small-team                      ;((activity ag)...)
   ?committed-contractors           ;(ag...)
   ?failed-contractors              ;((ag reason)...)
   ?replacement-contractors         ;(ag ...)
   ?successful-contractors          ;(ag...)
   ))
```

This is an example from the supply chain application. It specifies the main conversation of the Logistics agent, used to interact with the customer and with enterprise agents. Note the use of the conversation rule interpreter that invocks the knowledge acquisition interface and the documentation of the variables in the conversation.

```
(def-conversation-class 'form-small-team-class

  :name 'form-small-team-class
  :content-language 'kif
  :speech-act-language 'kqml
  :initial-state 'start
  :final-states '(ok failed)
  :control 'interactive-choice-control-ka
  :variables
    '(?large-team                    ; ((activity ag ag ..) ..) is consumed step by step
     ?current-activity               ; activity
     ?agent-group                    ; (ag ...)
     ?current-agent                  ; ag , from the group,
     ?result                         ; ((activity ag) ..)
     ))
```

The second example illustrates a conversation class used by Logistics to negotiate forming teams with enterprise agents to satisfy orders. Note again the structure of the conversation's local data base formed by a set of variables that persistently store various results used during the negotiation.

6 Conversation Rules

6.1 Syntax

```
(def-conversation-rule <name>

  :name <name>
  :comment <comment>
  :current-state <state>
  :received-test <received-test>
  :received <received>
```

:received-many <received-many>
:received-queue-test <received-queue-test>
:waits-for-test <waits-for-test>
:such-that <such-that>
:next-state <next-state>
:transmit <KQML message>
:wait-for <wait-for>
:do-before <do-before>
:do-after <do-after>
:do <do>
:interactive-execution-fn <interactive-execution-fn>
:incomplete <T or nil>

)

- <name> is the unique global name of the rule.
- <comment> is any string used to document the rule
- <state> is a state name
- <received-test> is a predicate of one argument, the received message. Used to perform procedural checks on the received message.
- <received> is a message pattern against which the actual message will be checked. The way to specify message patterns and the semantics of pattern-matching are described in the next section. The use of pattern-matching enables the user to specify declaratively the expected structure of a message in order to apply a rule.
- <received-many> list of patterns against corresponding received messages will be matched. This enables us to match several messages in a rule.
- <received-queue-test> predicate of one argument, the queue or messages for the conversation. Enables checking the entire queue before applying the the rule.
- <waits-for-test> predicate of one argument, the list of terminated conversations this conversation is waiting for. This can be used only if the current conversation is waiting for other conversations to terminate, enabling the conversation to be resumed as soon as the test condition is satisfied.
- <such-that> predicate of any number of arguments applied on a list of bindings produced by matching the patterns in :received or :received-many. A number of standard variables bound by the system are available.
- <next-state> state name, the next state if the rule is applied.
- <KQML meesge> a KQML message to be transmitted as an effect of rule execution. Anywhere in the KQML message, values of the form (?<expr>) are replaced by the value of <expr>. Inside <expr>, free variables are first replaced by their values. This gives a way of performing arbitrary computations to determine any components of the message to be sent.
- <wait-for> list of conversations this conversation is waiting for to terminate. The conversation is put on wait as a consequence of executing the rule.
- <do-before> executable actions to be carried out before transmitting the <KQML message>
- <do-after> executable actions to be carried out after transmitting the <KQML message>
- <do> executable actions to be carried out at an unspecified moment in relation to when the <KQML message> is transmitted
- <interactive-execution-fn> a user written function that has application-specific GUI-s guiding the execution of the action part of the rule.

6.2 Semantic Issues

This construct describes a rule whose execution is as follows:

If

current state of the conversation is :current-state
and :received-test is satisfied by the last message
and last received message matches :received
and there exists a set of messages in the queue matching :received-many (order matters)
and the message queue satisfies :received-queue-test
and :wait-for-test predicate satisfied by the list of conversations this conversation is waiting for
and :such-that predicate satisfied

Then

go to :next-state
and transmit the :transmit message
and put the conversation on wait for the mentioned :wait-for conversations
and do the :do-before actions before transmitting the message
and do the :do-after actions after transmitting the message
and do the :do actions anytime

or *if* interactive-execution-fn exists, execute it (assume it will carry out all above actions)

The syntax and semantics of pattern-matching operations (involved in the :received and :received-many conditions) will be discussed in the next subsection.

The use of predicates and other executable functions in places like :such-that and do clauses assumes that programmers are allowed to place as free variables any variables of the conversation. These will be replaced with their actual values before evaluating the forms they are in. Moreover, the following variables are always bound by the system and can be used as well:

?convn - the name of the current conversation

?agent - the agent who owns the current conversation

?message - the :received message

?conv - the current conversation (as object).

See the examples for further details.

6.3 Examples

6.3.1 Example 1.

```
(def-conversation-rule 'cc-1
  :name 'cc-1
  :current-state 'start
  :transmit
   '(propose
     :sender ?agent
     :receiver logistics
     :content
        (customer-order
           :date "08feb96"
           :priority 1
           :destination Montreal
```

```
            :customer BoM
            :line-items
              ((:id INV1
                :product desk-lamp
                :due-date "09mar96"
                :quantity 100
                :price 10
                :unit-transportation-cost 0.1)
               (:id INV2
                :product clip-reading-lamp
                :due-date "12mar96"
                :quantity 150
                :price 8
                :unit-transportation-cost 0.1)))
                :conversation ?convn)
                :next-state 'proposed
  :do '(send ?conv :set-ivar 'status 'waiting-for-input)
  :incomplete t

)
```

This is a rule from the Supply Chain application. It shows how a Customer agent can send an order to Logistics as the first action in a conversation initiated by the customer. Note the use of free variables like ?agent and ?convn.

6.3.2 Example 2.

```
(def-conversation-rule 'crn-1

  :name 'crn-1
  :current-state 'start
  :received
    '(propose
     :sender customer
     :content(customer-order :line-items ?li))
  :next-state 'order-received
  :transmit
    '(tell
     :sender ?agent
     :receiver customer
     :content '(working on it)
     :conversation ?convn)
  :do '(put-conv-var ?conv '?order (cadr(member :content ?message)))
  :incomplete nil

)
```

In the second example we have a rule that the Logistics agent could be using to respond to the previous message sent by the customer. Note the use of pattern-matching in the :received slot. By matching the message transmitted by the customer against the pattern in the :received slot we determine that the rule is applicable (and ?li gets bound to the list of line-items). Applying the rule results in Logistics responding with the message

(tell :sender logistics :receiver customer :content (working on it) :conversation c1)

assuming c1 is the name of the conversation. As a side-effect, Logistics creates a variable ?order in this conversation's data base, containing the customer's order. This is extracted from the message sent by the customer as the value associated to the :content keyword.

6.3.3 Example 3.

```
(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))
```

This is a rule from the n-queens program shown in a subsequent section of this document. Note the binding of variables in the :received slot and then the use of evaluable expressions in the :transmit slot. The receiver of the transmitted message is dynamically computed as whichever agent is at the right of the current agent, while the content is also dynamically computed by choosing a new position on the table. (See the full program for details). The ability to use evaluable expressions and variables in the transmitted message allows for power of expression and conciseness. Because of this for example, we could write a n-queens programe where all middle queens are described by the same conversation class that relies on rules like the above to determine the right-of queen and the new position of the current queen dynamically.

7 Pattern matching in rules

Conversation classes are generic plans of agents specifying behavior in certain situations. An important way of ensuring genericity of specification is being able to specify behavior that is associated to *classes* of messages (as opposed to individual ones) received by an agent. For example, the Logistics agent behaves in a certain way for *any* proposal for manufacturing an order received from the Customer. In COOL we allow the received messages that a rule considers to be specified by a pattern that describes an entire class of actual messages for which the rule is applicable.

The following rules govern the use of patterns in COOL:

1. Patterns describe messages that are formed by a starting message type (performative in KQML) followed by any number of keyword value pairs.

2. The order of the keyword value pair is irrelevant, in the sense that messages with any actual order of these pairs will match a pattern that is matchable.

3. The values in the keyword value pairs can be arbitrary lists. If they contain keyword value pairs, order of the pairs is irrelevant.

4. Patterns may contain variables which are unified with corresponding values from the message.

7.1 Syntax

<variable>::= ?<symbol> | ??<symbol>

The two notations (e.g. ?x and ??x) have different matching semantics, to be explained later on.

<pattern>::= any message containing variables

7.2 Semantics

The ?<symbol> variable matches any single element in the corresponding position. The ??<symbol> variable matches a list on the corresponding position. Unification rules apply: (i) an unbound variable unifies with some value and gets bound to it (ii) a bound variables unifies with some value iff its value is equal to that value.

7.3 Examples

Message:

(propose

:language list
:sender a1
:receiver a2
:content (produce 22 widgets (22 apr 96)))

Patterns and bindings:

1. (propose :sender a1)

Result of matching: T
Bindings: none

2. (?type :sender a1)

Result of matcing: T
Bindings: ((?type propose))

3. (propose :content ?c)

Result of matching: T
Bindings: ((?c (produce 22 widgets (22 apr 96)))

4. (propose :content (produce 223 widgets (22 apr 96)))

Result of matching: nil

5. (propose :content (produce ??what ?when))

Result of matching: T
Bindings: ((??what (22 widgets))(?when (22 apr 96)))

6. ??msg

Result of matching: T
Bindings: ((??msg ((propose :language list :sender a1 :receiver a2 :content (produce 22 widgets (22 apr 96))))))

7. (propose :content (?do ?amount ?what (?day ?month ?year)))

Result of matching: T
Bindings: ((?do produce)(?amount 22)(?what widgets)(?day 22)(?month apr)(?year 96))

8. (propose :content (?do ?amount ?what ?day ?month ?year))

Result of matching: nil

9.(propose :content (??what (?dm 96)))

Result of matching: nil

10. (propose :content (??what (??dm 96)))

Result of matching: T
Bindings: ((??what (produce 22 widgets))(??dm (22 apr)))

11. (propose :content (produce ?n ?w (?n apr ?y)))

Result of matching: T
Bindings: ((?n 22)(?w widgets)(?y 96))
Note: ?n successfully checked for equality second time

12. (propose :content (produce ?n widgets(?n apr ?n)))

Result of matching: nil
Note: ?n bound to 22 does not unify with 96.

8 Recovery Rules

8.1 Syntax

(def-recovery-rule <name>

:name <name>
:comment <string>
:input-queue-test <predicate>
:conversation-test <predicate>
:do <expression>
:skip-input-until <predicate>
:go-state <state>
:error <expression>
:incomplete <T or nil>)

Where:

- <name> is the name of the rule
- <comment> is a string stored with the rule
- <predicate> is any expression that is treated like a predicate (a result non-nil is interpreted as true). Free variables are bound to their values in the conversation.
- <expression> is any executable expression (the result is not interpreted). Free variables are bound to their values in the conversation.
- <state> is a state in the conversation class

8.2 Semantics

The intended semantics in this version of the language is as follows:

If

predicate in :input-queue-test, applied on the input-queue of the conversation, is true
and predicate in :conversation-test, applied on the conversation, is true
and conversation is in :current-state

then

do the :do actions (can use free variables bound to their values in the conversation)
and skip input messages from the conversation queue until condition in :skip-messages-until, applied on the message queue, becomes true
and go to state named in :go-state

and signal an error whose message is produced by the evaluation of the expression in :error.

In the current version hence, one can ignore some messages, can change the state of the conversation, can signal an error or can do other (arbitrary) actions - all of these to respond to unexpected situations or events.

Remember that recovery rules are applied when no conversation rules can be applied in the current state of a conversation. An implementation may choose to make application of recovery rules conditional on the values of some global flag (this is also done in our prototype implementation).

9 Continuation Rules

9.1 Syntax

(def-continuation-rule <name>

:name <name>
:comment <comment>
:input-queue-test <predicate>
conversations-test <predicate>
:do <action>
:incomplete <T or nil>)

Where:

- <name> is the name of the conversation
- <commment> is a descriptive string stared with the conversation
- <predicate> is a predicate
- <action> is any evaluable expression.

9.2 Semantics

In the current version of the language, the meaning of coordination rules is as follows:

If

the :input-queue-test predicate applied on the *input queue of the agent* is true and represents a new conversation name (that the agent does not have yet)

xor

the :conversations-test applied on the list of existing conversations of the agent is true and represents an existing conversation name

then

create the new conversation (first case) and execute it

xor

execute the existing conversation (second case)

and

do the actions in :do (both cases).

9.3 Examples

The following are continuation rules routinelly used in our applications:

9.3.1 Example 1:

```
(def-continuation-rule 'cont-0

  :name 'cont-0
  :conversations-test 'cvt-0)

(defun cvt-0(conversations &aux aux)

  (if conversations
    (if (setq aux(exists-conv-runnable-or-waiting conversations))
        aux
        nil)
   nil))

(defun exists-conv-runnable-or-waiting (conversations &aux aux)

 (dolist(c conversations nil)

  (when(or(and (eq (send c :status) 'waiting-for-input)
                    (send c :input-queue))
              (eq (send c :status) 'runnable))
    (return c))))
```

In this example, cont-0 is a continuation rule which checks if there exist conversations that are either (1) runnable or (2) waiting for input and having available input. In any of these cases, the first conversation satisfying this condition is returned and consequently run.

9.3.2 Example 2:

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

  :name 'cont-1
  :input-queue-test 'iqt-1)

(defun iqt-1 (queue)

 (if queue

     (exists-conv-class-initially-accepting (car queue))

     nil))
```

In this example, cont-1 checks if there exist conversation classes that in their initial state accept the first message in the agent queue. If such a conversation class is found, it is instantiated as a new conversation of the agent. The intent/intent-check mechanism is used here.

9.3.3 Example 3:

```
(def-continuation-rule 'cont-2

  :name 'cont-2
  :conversations-test 'cvt-1)

(defun cvt-1(conversations &aux aux)

 (if conversations

  (if (and(setq aux(exists-conv-not-receiving-input conversations))
          (not(eq(send aux :status) 'waits-for-conversations)))

    aux nil) nil))
```

In this example, a conversation that in the initial state does not receive inputs and is not on wait for another conversations is selected. Usually, this conversation transmitts messages in in the initial state and this is a way of kicking-off the interactions.

10 Associations

Associations provide a way to avert the problem of referencing an object that has not been defined yet. Associations can be used anywhere in the program as long as the objects it references have been defined.

10.1 Syntax

(def-association

:conv-rule <conv-rules> :conv-class <class>
:cont-rule <cont-rules> :agent <agent>
:err-rule <err-rules> :conv-class <class>
:conv-class <classes> :agent <agent>
:conv <conversations> :agent <agent>
:agent <agents> :manager <manager>)

Where:

<conv-rules>::= <conv-rule-name> | (<conv-rule-name> ...)

<cont-rules>::= <cont-rule-name> | (<cont-rule-name >...)

<err-rules>::= <err-rule-name> | (<err-rule-name> ...)

<classes>::= <class-name> | (<class-name> ...)

<conversations>::= (<class-name>< conv-name>) | ((<class-name>< conv-name>)...)

<agents>::= <agent-name> | (<agent-name> ...)

10.2 Semantics

The form describes the association of

(1) conversation-rules to conversation classes

(2) continuation-rules to agents

(3) error recovery-rules to conversation classes

(4) conversation classes to agents

(5) conversations to agents

(6) agents to conversation managers.

The keywords are indicative of these associations.

10.3 Example

The following is the full association table of a version of the supply chain application.

```
(def-association
 :agent '(logistics customer plant1 plant2 plant3 transp1 transp2)
     :manager m1
 :conv-class '(customer-conversation)
     :agent customer
```

```
:conv-class '(logistics-execution-net form-large-team-class form-small-team-class
              kick-off-execution-class find-contractor-class)
    :agent logistics
:conv-class '(answer-form-large-team-class answer-form-small-team-class
              answer-kick-off-execution-class monitor-execution-class)
    :agent plant1
:conv-class '(answer-form-large-team-class answer-form-small-team-class
              answer-kick-off-execution-class monitor-execution-class)
:agent plant2
:conv-class '(answer-form-large-team-class answer-form-small-team-class
              answer-kick-off-execution-class monitor-execution-class)
    :agent plant3
:conv-class '(answer-form-large-team-class answer-form-small-team-class
              answer-kick-off-execution-class monitor-execution-class)
    :agent transp1
:conv-class '(answer-form-large-team-class answer-form-small-team-class
              answer-kick-off-execution-class monitor-execution-class)
    :agent transp2
:cont-rule '(cont-0 cont-1 cont-2 cont-3)
    :agent customer
:cont-rule '(cont-0 cont-1 cont-2 cont-3)
    :agent logistics
:cont-rule '(cont-0 cont-1 cont-2 cont-3)
    :agent plant1
:cont-rule '(cont-0 cont-1 cont-2 cont-3)
    :agent plant2
:cont-rule '(cont-0 cont-1 cont-2 cont-3)
    :agent plant3
:cont-rule '(cont-0 cont-1 cont-2 cont-3)
    :agent transp1
:cont-rule '(cont-0 cont-1 cont-2 cont-3)
    :agent transp2
:conv-rule '(crn-1 crn-2 crn-3 crn-4 crn-5 crn-6 crn-7 crn-8 crn-9 crn-10 crn-11 crn-12
             crn-13 crn-14 crn-15 crn-16 crn-17 crn-18 crn-19 crn-20 crn-21 crn-22 crn-23
             crn-24 crn-25 crn-26 crn-27)
    :conv-class logistics-execution-net
:conv-rule '(cc-1 cc-2 cc-3 cc-4 cc-5 cc-6 cc-7 cc-8 cc-9 cc-10 cc-11 cc-12 cc-13)
    :conv-class customer-conversation
:conv-rule '(flt-1 flt-2 flt-3 flt-4 flt-5 flt-6)
    :conv-class form-large-team-class
:conv-rule '(aflt-1 aflt-2 aflt-3 aflt-4 aflt-5)
    :conv-class answer-form-large-team-class
:conv-rule '(fst-1 fst-2 fst-3 fst-4 fst-5 fst-6 fst-7 fst-8 fst-9 fst-10 qfst-11 fst-12)
    :conv-class form-small-team-class
:conv-rule '(afst-1 afst-2 afst-3 afst-4 afst-5 afst-6 afst-7 afst-8 afst-9 afst-10)
    :conv-class answer-form-small-team-class
:conv-rule '(koe-1 koe-2 koe-3 koe-4)
    :conv-class kick-off-execution-class
:conv-rule '(akoe-1 akoe-2 akoe-3)
    :conv-class answer-kick-off-execution-class
:conv-rule '(fcc-1 fcc-2 fcc-3 fcc-4 fcc-5 fcc-6 fcc-7 fcc-8 fcc-9 fcc-10 fcc-11)
    :conv-class find-contractor-class
:conv-rule '(mec-1 mec-2)
    :conv-class monitor-execution-class)
```

11 A Full COOL Program for Solving N-queens

11.1 Description

As an example of COOL programming we present n-queens, a well-known problem that lends itself well to multi-agent modeling. We model each queen as an agent. Each queen moves along one column of the table. For this reason, to describe the position of a queen we only need the number of its line.

Queens have limited knowledge about the positions of the other queens. We use a message passing convention where each queen can only talk to its neighbors on the left and on the right (the first queen talks only with the queen on its right side, while the last queen talks only to the queen on its left side).

Queens use three speech acts in communication. *Propose* is used when a queen places itself in a position and sends that to the queen on its right. *Accept* is a message sent by a queen to the queen on its left to signal that it has found a position that is not in conflict with the queens on its left. *Reject* is a message sent to the queen on the left to express the fact that the sending queen can not find any position compatible with those of the queens on the left.

The content of messages, for a queen q_i (in the i-th column) is the list of positions of all queens on its left. Thus a queen only knows where the queens on its left are placed. If the rightmost queen is able to find a position, then the problem ends successfully. If the leftmost queen can not find any position, then no solutions are possible any more.

In summary, the following types of messages are exchanged:

(propose :sender q_i :receiver q_{i+1} :content (n_1 n_2 ... n_i))

(accept :sender q_i :receiver q_{i-1} :content (n_1 n_2 ... n_i))

(reject :sender q_i :receiver q_{i-1} :content (n_1 n_2 ... n_{i-1}))

11.2 Source code

```
;;;
;;; COOL solution for the n-queens problem
;;; created: mb
;;; date: nov 4, 94
;;; ported to xlisp: feb, 20, 1995, mb
;;;

;;; assume 4 queens

;; 1. the 4 queens as agents
;;-------------------------

(def-agent ‘q1)
(def-agent ‘q2)
(def-agent ‘q3)
(def-agent ‘q4)

;; 2. the conversation manager
;;---------------------------

(def-conversation-manager ‘m1)

;; 3. The three conversation classes
;;---------------------------------
```

```
;;
;; First, the conversation for the leftmost queen
;;

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

;;
;; Second, 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))

;;
;; Third, 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))

;; 4. 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))

;; 5. 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))

;; 6. 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)
 :do ‘(format t “~%;;; Solution found ~s” ?c))

(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))


;; 7. Functions
;;-------------

(defun make-list (n &key (initial-element nil) &aux aux)
 (when (or (not(numberp n))
           (<= n 0))
   (error “make-list, arg must be positive number” n))
 (dotimes (i n aux)
   (setq aux (cons initial-element aux))))

(defun remove-last(l) (reverse(cdr(reverse l))))

(defun at-left(a1 a2 &aux )
 “ is a1 at left of a2?”
   (string< (symbol-name a1) (symbol-name a2)))

(defun at-right(a1 a2 &aux)
 “ is a1 at the right of a2 “
```

```
  (string< (symbol-name a2)(symbol-name a1)))

(defun left-of(a &aux(queens '(q4 q3 q2 q1)) aux)
 " returns the agent at the left of a"
  (when(setq aux(member a queens))
     (cadr aux)))

(defun right-of(a &aux(queens '(q1 q2 q3 q4)) aux )
 " returns the agent at the right of a "
   (when(setq aux(member a queens))
      (cadr aux)))

(defparameter *queens-no* 3) ; 0, 1, 2, 3, - 4 queens

(defun new-position-exists(queen table &aux (column 0) (this-column (length table)))
 " checks if a new position exists for extending the partial solution in table "
 (cond((null table) t)
        (t(dotimes (i (1+ *queens-no*) nil)
            (dolist (pos table
               (return-from new-position-exists t)) ; pos = number, 0 .. 3
             (if(conflicts pos column i this-column)
               (progn
                  (setq column 0)
                  (return nil))
                  (incf column)))))))

(defparameter *tried-positions* (make-list (1+ *queens-no*)))
 ;; has form ((pos1 pos2 ...) (pos1 pos2 ...) ...)

(defun another-position-exists(queen table &aux (column 0)
                                                (this-column (1- (length table)))
                                                (old-column (car (last table)))
                                                prev-table )
 " checks if another position exists for the last queen in table "

 (rplaca (nthcdr this-column *tried-positions*)
 (nconc (nth this-column *tried-positions*) (list old-column)))

 (cond((null table) t)
        (t(setq prev-table (reverse(cdr(reverse table))))
          (dotimes (i (1+ *queens-no*) nil)
             (unless(member i (nth this-column *tried-positions*))
               (dolist (pos prev-table
                           (return-from another-position-exists t)) ;pos=number, 0 .. 3
                 (if(conflicts pos column i this-column)
                   (progn
                     (setq column 0)
                     (return nil))
                     (incf column)))))));

(defun choose-new-position(queen table &aux (column 0) (this-column (length table)))
 " chooses a new position, extending the partial solution in table "

 (cond((null table) (list 0))
        (t(dotimes (i (1+ *queens-no*)
                                        (error "CHOOSE-NEW-POSITION error"))
             (dolist (pos table
                                (return-from choose-new-position
                                        (nconc table (list i)))) ; pos = number, 0 .. 3
```

```
                (if(conflicts pos column i this-column)
                   (progn
                      (setq column 0)
                      (return nil))
                      (incf column)))))))

(defun choose-another-position(queen table &aux (column 0)
                                                (this-column (1- (length table)))
                                                (old-column (car (last table)))
                                                prev-table)
 " chooses another position for the last queen in table "

 (dotimes(i (- *queens-no* this-column))
       (rplaca(nthcdr (+ i (length table)) *tried-positions*) nil))

 (cond((null table) t)
        (t(setq prev-table (reverse(cdr(reverse table))))
          (dotimes (i (1+ *queens-no*)
                        (error "CHOOSE-ANOTHER-POSITION error"))
             (unless(member i (nth this-column *tried-positions*))
                (dolist (pos prev-table
                          (return-from choose-another-position
                                       (progn (rplaca (last table) i)
                                              table))) ; pos = number, 0 .. 3
                   (if(conflicts pos column i this-column)
                      (progn
                         (setq column 0)
                         (return nil))
                         (incf column))))))))

(defun conflicts (l1 c1 l2 c2)
 (or(= l1 l2)
     (= c1 c2)
     (= (abs (- l1 l2))
     (abs (- c1 c2)))))

;; 8. Continuation rules
;;----------------------

(def-continuation-rule 'cont-0
 :name 'cont-0
 :conversations-test 'cvt-0)

(defun cvt-0(conversations &aux aux)
 (if conversations
    (if (setq aux(exists-conv-waiting-and-input conversations)) aux nil)
    nil))

(defun exists-conv-waiting-and-input (conversations &aux aux)
 (dolist(c conversations nil)
    (when(or(and (eq(send c :status) 'waiting-for-input)
                    (send c :input-queue))
              (eq(send c :status) 'runnable))
       (format t "~% cvt-0: agent: ~s status ~s result ~s"
                  (send(send c :agent):name)
                  (send c :status)
                  (send c :name))
       (return c))))
```

```
(def-continuation-rule ‘cont-1
 :name ‘cont-1
 :input-queue-test ‘iqt-1)

(defun iqt-1 (queue)
 (if queue (exists-conv-class-initially-accepting (car queue)) nil))

(def-continuation-rule ‘cont-2
 :name ‘cont-2
 :conversations-test ‘cvt-1)

(defun cvt-1(conversations &aux aux)
 (if conversations
    (if (and(setq aux(exists-conv-not-receiving-input conversations))
            (not(eq(send aux :status) ‘waits-for-conversations))) aux nil)
    nil))

;; 9. the associations table
;;--------------------------

(def-association
 :conv-rule ‘(r11 r12 r13 r14) :conv-class qc-1
 :conv-rule ‘(r21 r22 r23 r24 r25) :conv-class qc-2
 :conv-rule ‘(r31 r32) :conv-class qc-3
 :cont-rule ‘(cont-0 cont-1 cont-2) :agent q1
 :cont-rule ‘(cont-0 cont-1 cont-2) :agent q2
 :cont-rule ‘(cont-0 cont-1 cont-2) :agent q3
 :cont-rule ‘(cont-0 cont-1 cont-2) :agent q4
 :agent ‘(q1 q2 q3 q4) :manager m1
 :conv-class qc-1 :agent q1
 :conv-class qc-2 :agent q2
 :conv-class qc-2 :agent q3
 :conv-class qc-3 :agent q4
 :conv ‘(qc-1 c1) :agent q1
)

;; 10. Traces
;;-----------

;(trace-agent m1 q1 q2 q3 q4)
;(trace-conv m1 ‘((,q1 c1 first-queen-class)
; (,q2 c1 middle-queen-class)
; (,q3 c1 middle-queen-class)
; (,q4 c1 last-queen-class)))
;(trace-conv m1 q2)
;(trace-conv m1 q3)
;(trace-conv m1 q4)
;(trace-message m1 q1 q2 q3 q4)
```

11.3 Execution traces

11.3.1 Message trace

This trace shows only the exchange of messages during problem solving.

```
;;; Message Log of Cool solution to n-queens
> (manage-conversations m1)
```

;;; (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 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)
;;; Solution found (1 3 0 2)
;;; (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

11.3.2 Full trace, including agent and conversation activation

;;; Trace of Cool solution to nqueens (4 in this example)

#P"/kingston/mihai/model/coordination/cool-nqueens.lsp"
> (trace send-message)
(SEND-MESSAGE)
> (manage-conversations m1)

;;; Activating agent Q4
;;; Agent Q4 idle: no conversation to be initiated or resumed
;;; Activating agent Q3
;;; Agent Q3 idle: no conversation to be initiated or resumed
;;; Activating agent Q2
;;; Agent Q2 idle: no conversation to be initiated or resumed
;;; Activating agent Q1
;;; Executing conversation C1
1 Enter SEND-MESSAGE (PROPOSE :SENDER Q1 :RECEIVER Q2 :CONTENT (0) :CONVERSATION C1)
1 Exit SEND-MESSAGE (PROPOSE :SENDER Q1 :RECEIVER Q2 :CONTENT (0) :CONVERSATION C1)

;;; Activating agent Q4
;;; Agent Q4 idle: no conversation to be initiated or resumed
;;; Activating agent Q3
;;; Agent Q3 idle: no conversation to be initiated or resumed
;;; Activating agent Q2
;;; Executing conversation C1
1 Enter SEND-MESSAGE (PROPOSE :SENDER Q2 :RECEIVER Q3 :CONTENT (0 2) :CONVERSATION C1)
1 Exit SEND-MESSAGE (PROPOSE :SENDER Q2 :RECEIVER Q3 :CONTENT (0 2) :CONVERSATION C1)

;;; Activating agent Q1
;;; Agent Q1 idle: no conversation to be initiated or resumed
;;; Activating agent Q4
;;; Agent Q4 idle: no conversation to be initiated or resumed
;;; Activating agent Q3
;;; Executing conversation C1
1 Enter SEND-MESSAGE (REJECT :SENDER Q3 :RECEIVER Q2 :CONTENT (0 2) :CONVERSATION C1)

1 Exit SEND-MESSAGE (REJECT :SENDER Q3 :RECEIVER Q2 :CONTENT (0 2) :CONVERSATION C1)

;;; Activating agent Q2
;;; Executing conversation C1
1 Enter SEND-MESSAGE (PROPOSE :SENDER Q2 :RECEIVER Q3 :CONTENT (0 3) :CONVERSATION C1)
1 Exit SEND-MESSAGE (PROPOSE :SENDER Q2 :RECEIVER Q3 :CONTENT (0 3) :CONVERSATION C1)

;;; Activating agent Q1
;;; Agent Q1 idle: no conversation to be initiated or resumed
;;; Activating agent Q4
;;; Agent Q4 idle: no conversation to be initiated or resumed
;;; Activating agent Q3
;;; Executing conversation C1
1 Enter SEND-MESSAGE (PROPOSE :SENDER Q3 :RECEIVER Q4 :CONTENT (0 3 1) :CONVERSATION C1)
1 Exit SEND-MESSAGE (PROPOSE :SENDER Q3 :RECEIVER Q4 :CONTENT (0 3 1) :CONVERSATION C1)

;;; Activating agent Q2
;;; Agent Q2 idle: no conversation to be initiated or resumed
;;; Activating agent Q1
;;; Agent Q1 idle: no conversation to be initiated or resumed
;;; Activating agent Q4
;;; Executing conversation C1
1 Enter SEND-MESSAGE (REJECT :SENDER Q4 :RECEIVER Q3 :CONTENT (0 3 1) :CONVERSATION C1)
1 Exit SEND-MESSAGE (REJECT :SENDER Q4 :RECEIVER Q3 :CONTENT (0 3 1) :CONVERSATION C1)

;;; Activating agent Q3
;;; Executing conversation C1
1 Enter SEND-MESSAGE (REJECT :SENDER Q3 :RECEIVER Q2 :CONTENT (0 3) :CONVERSATION C1)
1 Exit SEND-MESSAGE (REJECT :SENDER Q3 :RECEIVER Q2 :CONTENT (0 3) :CONVERSATION C1)

;;; Activating agent Q2
;;; Executing conversation C1
1 Enter SEND-MESSAGE (REJECT :SENDER Q2 :RECEIVER Q1 :CONTENT (0) :CONVERSATION C1)
1 Exit SEND-MESSAGE (REJECT :SENDER Q2 :RECEIVER Q1 :CONTENT (0) :CONVERSATION C1)

;;; Activating agent Q1
;;; Executing conversation C1
1 Enter SEND-MESSAGE (PROPOSE :SENDER Q1 :RECEIVER Q2 :CONTENT (1) :CONVERSATION C1)
1 Exit SEND-MESSAGE (PROPOSE :SENDER Q1 :RECEIVER Q2 :CONTENT (1) :CONVERSATION C1)

;;; Activating agent Q4
;;; Agent Q4 idle: no conversation to be initiated or resumed
;;; Activating agent Q3
;;; Agent Q3 idle: no conversation to be initiated or resumed
;;; Activating agent Q2
;;; Executing conversation C1
1 Enter SEND-MESSAGE (PROPOSE :SENDER Q2 :RECEIVER Q3 :CONTENT (1 3) :CONVERSATION C1)

```
1 Exit SEND-MESSAGE (PROPOSE :SENDER Q2 :RECEIVER Q3 :CONTENT (1 3) :CONVER-
SATION C1)

;;; Activating agent Q1
;;; Agent Q1 idle: no conversation to be initiated or resumed
;;; Activating agent Q4
;;; Agent Q4 idle: no conversation to be initiated or resumed
;;; Activating agent Q3
;;; Executing conversation C1
1 Enter SEND-MESSAGE (PROPOSE :SENDER Q3 :RECEIVER Q4 :CONTENT (1 3 0) :CON-
VERSATION C1)
1 Exit SEND-MESSAGE (PROPOSE :SENDER Q3 :RECEIVER Q4 :CONTENT (1 3 0) :CON-
VERSATION C1)

;;; Activating agent Q2
;;; Agent Q2 idle: no conversation to be initiated or resumed
;;; Activating agent Q1
;;; Agent Q1 idle: no conversation to be initiated or resumed
;;; Activating agent Q4
;;; Executing conversation C1
1 Enter SEND-MESSAGE (ACCEPT :SENDER Q4 :RECEIVER Q3 :CONTENT (1 3 0 2) :CON-
VERSATION C1)
1 Exit SEND-MESSAGE (ACCEPT :SENDER Q4 :RECEIVER Q3 :CONTENT (1 3 0 2) :CON-
VERSATION C1)

;;; Solution found (1 3 0 2)
;;; Activating agent Q3
;;; Executing conversation C1
1 Enter SEND-MESSAGE (ACCEPT :SENDER Q3 :CONTENT (1 3 0 2) :RECEIVER Q2 :CON-
VERSATION C1)
1 Exit SEND-MESSAGE (ACCEPT :SENDER Q3 :CONTENT (1 3 0 2) :RECEIVER Q2 :CON-
VERSATION C1)

;;; Activating agent Q2
;;; Executing conversation C1
1 Enter SEND-MESSAGE (ACCEPT :SENDER Q2 :CONTENT (1 3 0 2) :RECEIVER Q1 :CON-
VERSATION C1)
1 Exit SEND-MESSAGE (ACCEPT :SENDER Q2 :CONTENT (1 3 0 2) :RECEIVER Q1 :CON-
VERSATION C1)

;;; Activating agent Q1
;;; Executing conversation C1
;;; Solution found (1 3 0 2)
;;; Activating agent Q4
;;; Agent Q4 idle: no conversation to be initiated or resumed
;;; Activating agent Q3
;;; Agent Q3 idle: no conversation to be initiated or resumed
;;; Activating agent Q2
;;; Agent Q2 idle: no conversation to be initiated or resumed
;;; Activating agent Q1
;;; Agent Q1 idle: no conversation to be initiated or resumed
;;; No agent can be activated
NIL
> (agent-conversations q1)
(
 ***** conversation C1 *****
 class FIRST-QUEEN-CLASS
 topic NIL
 agent Q1
 interlocutor NIL
```

```
 current-state YES
 status TERMINATED
 waits-for NIL
 input-queue NIL
 history ((R2 (REJECT :SENDER Q2 :RECEIVER Q1 :CONTENT (0) :CONVERSATION C1))
(R4 (ACCEPT :SENDER Q2 :CONTENT (1 3 0 2) :RECEIVER Q1 :CONVERSATION C1))))
> (agent-conversations q2)
(
 ***** conversation C1 *****
 class MIDDLE-QUEEN-CLASS
 topic NIL
 agent Q2
 interlocutor (Q1)
 current-state YES
 status TERMINATED
 waits-for NIL
 input-queue NIL
 history ((R2 (PROPOSE :SENDER Q1 :RECEIVER Q2 :CONTENT (0) :CONVERSATION C1))
 (R4 (REJECT :SENDER Q3 :RECEIVER Q2 :CONTENT (0 2) :CONVERSATION C1)) (R5
(REJECT :SENDER Q3 :RECEIVER Q2 :CONTENT (0 3) :CONVERSATION C1)) (R2 (PRO-
POSE :SENDER Q1 :RECEIVER Q2 :CONTENT (1) :CONVERSATION C1)) (R3 (ACCEPT
:SENDER Q3 :CONTENT (1 3 0 2) :RECEIVER Q2 :CONVERSATION C1))))
> (agent-conversations q3)
(
 ***** conversation C1 *****
 class MIDDLE-QUEEN-CLASS
 topic NIL
 agent Q3
 interlocutor (Q2)
 current-state YES
 status TERMINATED
 waits-for NIL
 input-queue NIL
 history ((R1 (PROPOSE :SENDER Q2 :RECEIVER Q3 :CONTENT (0 2) :CONVERSATION C1
)) (R2 (PROPOSE :SENDER Q2 :RECEIVER Q3 :CONTENT (0 3) :CONVERSATION C1)) (R5
(REJECT :SENDER Q4 :RECEIVER Q3 :CONTENT (0 3 1) :CONVERSATION C1)) (R2 (PRO-
POSE :SENDER Q2 :RECEIVER Q3 :CONTENT (1 3) :CONVERSATION C1)) (R3 (ACCEPT
:SENDER Q4 :RECEIVER Q3 :CONTENT (1 3 0 2) :CONVERSATION C1))))
> (agent-conversations q4)
(
 ***** conversation C1 *****
 class LAST-QUEEN-CLASS
 topic NIL
 agent Q4
 interlocutor (Q3)
 current-state YES
 status TERMINATED
 waits-for NIL
 input-queue NIL
 history ((R2 (PROPOSE :SENDER Q3 :RECEIVER Q4 :CONTENT (0 3 1) :CONVERSATION
C1)) (R1 (PROPOSE :SENDER Q3 :RECEIVER Q4 :CONTENT (1 3 0) :CONVERSATION C1))))
```