Process Modeling with Cooperative Agents

George T. Heineman[1]

Columbia University
Department of Computer Science
500 West 120th Street
New York, NY 10027
heineman@cs.columbia.edu

Abstract

Concurrency Control is the ability to allow concurrent access of multiple *independent* agents while still maintaining the overall consistency of the database. We discuss the notion of Cooperation Control, which gives a DBMS, the ability to allow cooperation of multiple *cooperating* agents, without corrupting the consistency of the database. Specifically, there is the need for allowing cooperating agents to cooperate while preventing independent agents from interfering with each other. In this paper, we use the MARVEL system to construct and investigate cooperative scenarios.

1 Introduction

Concurrency Control in database management systems allows multiple independent agents to concurrently access the database while maintaining its consistency. *Cooperation Control* extends this concept by considering situations with *cooperating* agents. To realize cooperation, we need to have semantic information about how the agents will act. Our research on Process Centered Environments (PCEs) has shown that these systems have a rich body of semantic information available. In such environments, a process is formally specified in a Process Modeling Language (PML). As part of this specification, the cooperation between agents needs to be provided.

There are several reasons why multiple agents might need to cooperate:

1. Uniqueness of agents – There might be certain tasks which can only be carried out by a particular agent; consider a task which can only be performed by a database administrator.
2. Encapsulation of tasks – The process might be designed such that there are clusters of tasks which are separated from other tasks. This hierarchical organization of tasks becomes necessary as the size and number of tasks grows.

[1]Heineman is supported in part by IBM Canada

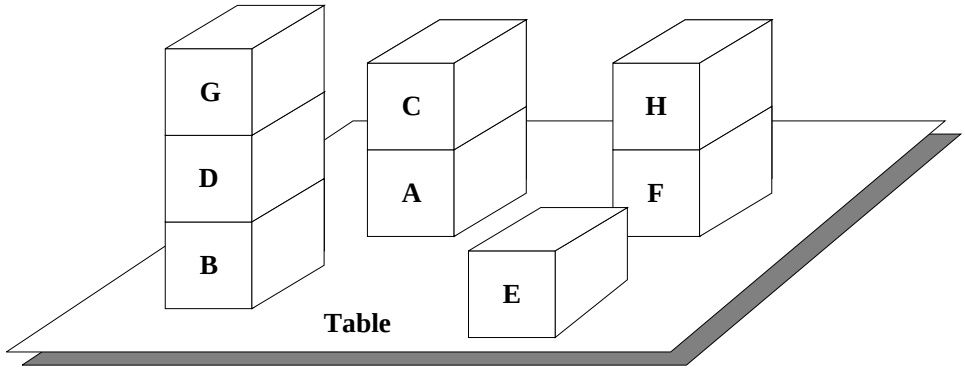

Figure 1: Blocks World

3. Group tasks – There are tasks which need multiple agents to work in concert with each other; consider a conference phone call between three parties.

The MARVEL project is an example of a PCE applied to software development. In this PCE, the process of software development is formally encoded in terms of rules, and the concurrency control of the database is tailored to provide specific behavior. In this paper we explore how to use MARVEL to produce a cooperative environment. We start with a simple example of cooperating agents in a "Blocks World" environment, and then apply our results to a fragment of the ISPW-7 [3] sample problem. We conclude with a discussion of the limitations and benefits of this approach.

2 Example problem

Consider the "Blocks World" example, as shown in figure 1. Blocks can either sit on the table, or on top of another block (the table is large enough to accommodate all blocks). A block **X** is *clear* if no block is sitting on top of **X**. Only clear blocks may be moved, and a block cannot have two blocks sitting directly on it. To move **A** on top of **E**, for example, **C** must first be moved to the table; then both **A** and **E** are clear, and the move can take place.

The PROLOG program in figure 3 is a goal-directed process which solves the problem of putting block **X** on top of **Y** by first making sure that both **X** and **Y** are clear, thus allowing the move to take place. Note that the **Table** may not be moved, but blocks may be moved onto it. This particular process achieves the `put_on(X,Y)` goal by first achieving two sub-goals `clear_space(X)` and `clear_space(Y)`. Figure 2 shows the solution for the request `put_on(d,a)`. Note how `put_on` and `clear_space` are recursively defined to invoke each other.

We now introduce multiple agents to this example problem. Assume, in the blocks

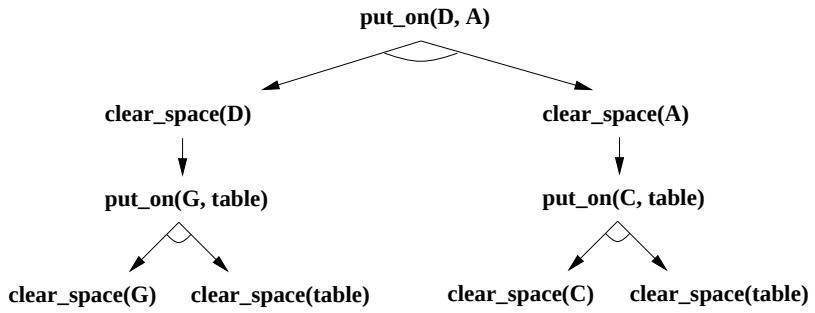

Figure 2: Goal Tree for `put_on(d,a)`

```
on_top_of(c,a).                                      %% Which blocks are on other blocks
on_top_of(d,b).
on_top_of(g,d).
on_top_of(h,f).
on_top_of(BLOCK,table) :- not (on_top_of(BLOCK, X)).    %% When a block is on the table
clear_space(table).                                  %% Always enough room on the table
clear_space(UNDER)  :- not (on_top_of(TOP, UNDER)).
clear_space(UNDER)  :- on_top_of(TOP, UNDER), put_on(TOP, table).
put_on(SRC,DST)     :- clear_space(SRC), clear_space(DST),
                       write('move '), write(SRC), write(' to '), write(DST), nl.
```

Figure 3: PROLOG solution for blocks

world, that there are two agents, *Placer* and *Clearer*. These agents cooperate in the following way:

1. When Placer moves block **X** to sit on object **Y**, Clearer is invoked to clear both **X** and **Y**. Note that **Y** may be a block or the **Table**.
2. When Clearer clears block **X**, Placer is invoked to move block **Y**, sitting on **X**, onto the **Table**.

The responsibilities of each agent are disjoint, and each has private tasks. Placer, for example, has no mechanism for knowing if block **X** is clear; it must blindly invoke Clearer. In similar fashion, Clearer knows how to clear a block only by requesting Placer to move other blocks. This scenario cannot be modeled in a single-process PROLOG environment, so we turn to the MARVEL system to design a multiple agent process.

3 Marvel

A MARVEL environment is defined by a data model, process model, tool envelopes and coordination model for a specific project. The data model is object-oriented,

```
parameters  WR
condition   WR
activity    WR
effects     X

# Lock Modes for builtins
rename  X
move    X  X
copy    S  X
link    X  X
unlink  X  X
delete  SX X
add     X
```

	S	X	ShW	WR
S	yes	no	no	yes
X		no	no	yes
ShW			yes	yes
WR				yes

(S) shared (ShW) shared write
(X) exclusive (WR) weak read

Figure 4: Transaction Table and Lock Compatibility Matrix

and uses classes to define an objectbase. The process is specified by MARVEL's process modeling language, MSL (MARVEL strategy language). Each process step is encapsulated by a *rule*, which has a name and typed parameters.

An MSL rule has four parts, a query, condition, activity and effects. When a rule is requested, a query is made on the database, and the rule's condition is checked. If it is satisfied, the activity is carried out and the assertions are made. A rule's activity is a shell envelope [2] which allows an administrator to integrate conventional tools into the process. There is a rule engine which employs chaining to drive the process. Backward chaining is initiated to satisfy the failed condition of a user's rule request. Forward chaining carries out the implications of a rule's assertions by firing those rules whose condition has become satisfied by the assertion. Backward and forward chaining are both recursive procedures.

Each rule is encapsulated by a transaction by which the rule accesses the objects it needs. Once the rule's query has determined the necessary objects, the rule processor acquires locks for these objects with lock modes based upon how the rule will access the objects. For example, as seen in figure 4, only those those objects being updated in the effects need to be locked in **X** exclusive mode. This table is the *mapping table* which maps rules to transactions.

A lock conflict situation occurs when a rule attempts to acquire a lock on an object which conflicts with an existing lock held by another rule. The conflicts are determined by a lock compatibility matrix supplied by the administrator. Figure 4 contains a sample table of four particular lock modes: *Shared*, *Exclusive*, *Shared Write*, and *Weak Read*. The matrix defines the compatibility of two lock modes; for example, **ShW** and **X** conflict, while **WR** is compatible with each lock mode.

In response to a particular locking conflict, MARVEL turns to the specified coordination model to determine an appropriate response. This model contains a set of CORD (Coordination Rule Language) rules which outlines the prescribed actions to take. If a rule matches a situation, a set of actions are carried out and the conflict

```
OBJECT_CLASS :: superclass ENTITY;
  clear     : boolean = true;
  on_top_of : set_of OBJECT;
end

OBJECT :: superclass OBJECT_CLASS;
  Movable : boolean = true;
end

TABLE :: superclass OBJECT_CLASS;
  Movable : boolean = false;
end
```

Figure 5: MSL data schema

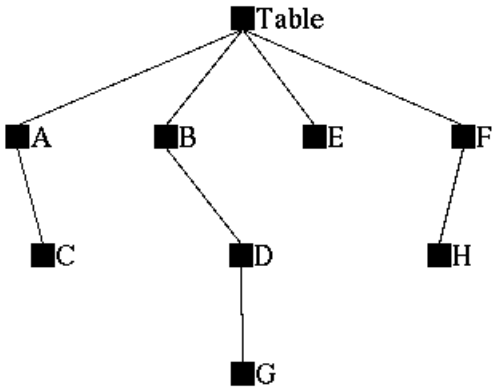

Figure 6: MARVEL blocks representation

is resolved, otherwise the transaction is aborted, and its rule is stopped. We now present a MARVEL environment which solves the multiple agent blocks world.

3.1 Multiple Agent Solution

The data model, shown in figure 5, is comprised of three classes, OBJECT_CLASS, OBJECT, and TABLE. The *clear* attribute of an object tells whether it is clear or not, and the *movable* attribute determines if an object can be moved. The *on_top_of* attribute is a composition attribute which contains the block (if it exists), which is sitting on a given object. Figure 6 shows an objectbase which models the blocks world example from figure 1. The block **B**, for example, has its *clear* attribute equal to `false`, and its *on_top_of* attribute would be equal to the block {**D**}.

The process model has four rules. There are two PUT_ON rules, to handle different cases, and an AUTO_MOVE rule which automatically sets the *clear* attribute of a block X to `false` when a block is placed on X. There is one CLEAR_SPACE rule which makes a particular block clear. The rules are shown in figure 7.

```
# When ?src comes from on top of another object
put_on [?src:OBJECT, ?dst:OBJECT_CLASS]:
  (exists OBJECT ?under suchthat (member [?under.on_top_of ?src])):

  { CLEARER clear_space ?src.Name ?dst.Name }

  (and          (move [?src ?dst on_top_of ?under])
       no_chain (?under.clear = true));
  no_assertion;

# When ?src comes from the TABLE
put_on [?src:OBJECT, ?dst:OBJECT_CLASS]:
  (exists TABLE ?tbl suchthat (member [?tbl.on_top_of ?src])):

  { CLEARER clear_space ?src.Name ?dst.Name }

  (move [?src ?dst on_top_of ?tbl]);
  no_assertion;

hide auto_move[?o:OBJECT]:
  # This rule doesn't apply to the Table, since the Table is always clear
  (exists OBJECT_CLASS ?under suchthat (and (member [?under.on_top_of ?o])
                                            (?under.Movable = true))):
  { }
  (?under.clear = false);

clear_space [?tbl:TABLE]:
 :
 { }
 ;

clear_space [?object:OBJECT]:
 :
  no_chain (?object.clear = true)
 { }
 ;

clear_space [?under:OBJECT]:
  (and (exists OBJECT ?obj suchthat no_chain (member [?under.on_top_of ?obj]))):
  no_chain (?under.clear = false)

  { PLACER put_on ?obj.Name "Table" }

  (?under.clear = true);
  no_assertion;
```

Figure 7: MARVEL multiple agent solution

```
ENVELOPE clear_space;
INPUT
  string : SRC;
  string : DST;
OUTPUT none;
BEGIN
  ## Clear both objects by invoking an agent to execute: clear_space(SRC)  clear_space(DST)
  SCRIPT_FILE=/tmp/clear_space
  echo "#!marvel script"      > $SCRIPT_FILE
  echo "clear_space $SRC"    >> $SCRIPT_FILE
  echo "clear_space $DST"    >> $SCRIPT_FILE

  ## Invoke the agent ##
  OUTPUT_FILE=/tmp/OUTPUT
  marvel -b $SCRIPT_FILE > $OUTPUT_FILE

  ## Check status and clear up ##
  RC=1
  ERROR=`grep "Failed while interpreting ${SCRIPT_FILE}" ${OUTPUT_FILE}`
  if [ "x$ERROR" = "x" ]
  then
    RC=0     # Succeeded
  fi
  rm $OUTPUT_FILE
RETURN "$RC";
END
```

Figure 8: SEL envelope for PUT_ON

In order to separate tasks belonging to different agents, the PUT_ON and CLEAR_SPACE rules have no logical condition associated with them. The PUT_ON[X,Y] rule, for example, must invoke an agent to clear both X and Y for it to perform its operation. To do so, the PUT_ON rule executes the shell envelope shown in figure 8. This envelope creates a new agent which will execute CLEAR_SPACE[X] and CLEAR_SPACE[Y], returning "0" on success, and "1" on failure. This return code will direct MARVEL to assert the appropriate effect as defined in the PUT_ON rule (i.e., on success, the `move` operation is asserted). This process is recursive as the agent executing CLEAR_SPACE[X] might create a new agent to complete its task.

The final information MARVEL needs is the coordination model, which is defined in terms of CORD rules. Specifically, the MSL rules in figure 7 will produce conflicting database access. Consider issuing the PUT_ON[**D**,**A**] rule on the example in figure 6. This, we have seen, will cause an agent to be created to invoke CLEAR_SPACE[**D**] and CLEAR_SPACE[**A**]. The original PUT_ON rule, however, must access the objects **D** and **A** in exclusive access mode, since it must prevent other agents from interfering with its operation. The objectbase would become inconsistent if another agent mistakenly placed another block on **D** after the CLEAR_SPACE[**D**] invocation has completed, but before CLEAR_SPACE[**A**] has started. However, CLEAR_SPACE[**D**] (invoked by the cooperating agent) needs to access **D** in an exclusive mode also, since it removes **G** from on top of **D**. We need some mechanism for allowing the cooperating agents to access information jointly, while preventing conflicting access by independent agents.

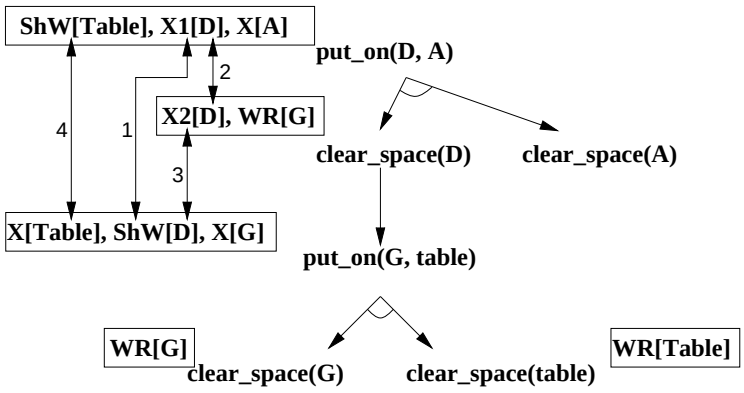

Figure 9: Locking conflicts for PUT_ON[**D**, **A**]

In our multiple agent block world example, there are four particular situations, labeled 1 through 4, which are resolved by the control rules in figure 10. These situations correspond exactly to those locking conflicts in figure 9. In each case, the CORD action simply ignores the conflict, allowing the lock request to succeed, and thus the entire process succeeds.

We now explain the process trace in figure 9, omitting all intention locks (these are normally acquired because of the composition of the objectbase; see [1]). When PUT_ON[**D**,**A**] is requested, the first PUT_ON rule is fired, and the three locks are acquired (**X1**[**D**] is the first exclusive lock requested for block **D**). This rule executes the `clear_space` envelope which invokes an agent to CLEAR_SPACE[**D**]. To execute this rule, two locks need to be acquired; however a conflict occurs as the second **X**[**D**] lock is requested, since the two locks are incompatible. This lock conflict is repaired by the second condition pair in the `OBJECT_conflict` CORD rule. Note that both **X** locks are set on **D**. The CLEAR_SPACE rule executes the `put_on` envelope which invokes another agent to PUT_ON[**G**, **Table**]. As these locks are acquired, three separate conflicts occur, and each is handled by the appropriate CORD condition pair. We omit the right side of the process tree (CLEAR_SPACE[**A**]) as its execution is identical.

4 Software Process Application

Figure 11 is a partial fragment from the ISPW-7 sample problem [3]. We apply the concepts shown in this paper to this fragment, and show how multiple agents can cooperate. There are three agents, the Reviewer, the Designer, and the Programmer. They each have a set of tasks (in white boxes) that they must perform. The solid arrows define the sequence of tasks for an individual agent, and the dashed arrows show how the agents communicate with each other. The long grey vertical boxes represent the transactions encapsulating each agents's actions. The work starts when

```
OBJECT_conflict [ OBJECT ]
bindings:
   ?t1 = holds_lock ()
   ?t2 = requested_lock ()
body:
   if (and (?t1.rule = clear_space)          # An agent is using put_on[obj, table]
           (?t2.rule = put_on))              # to clear_space for user command
   then {                                    # put_on[A, B], where object "obj" is sitting on
       notify(?t2, "Conflict-1")             # object A.
       ignore()
   }
   if (and (?t1.rule = put_on)               # An agent is using clear_space[obj] to
           (?t2.rule = clear_space))         # clear space for user command
   then {                                    # put_on[obj, A] or put_on[A, obj].
       notify(?t2, "Conflict-2")
       ignore()
   }
   if (and (?t1.rule = put_on)               # An agent is using put_on(X, Y) and a
           (?t2.rule = put_on))              # subagent has been invoked to use
   then {                                    # put_on(Y,table) to clear_space for Y.
       notify(?t2, "Conflict-3")
       ignore()
   }
end_body;

TABLE_conflict [ TABLE ]

bindings:
   ?t1 = holds_lock ()
   ?t2 = requested_lock ()
body:
   if (and (?t1.rule = put_on)               # Two agents are trying to place
           (?t2.rule = put_on))              # blocks on the same table
   then
     {
       notify(?t2, "Conflict-4")
       ignore()
     }
end_body;
```

Figure 10: CORD coordination rules

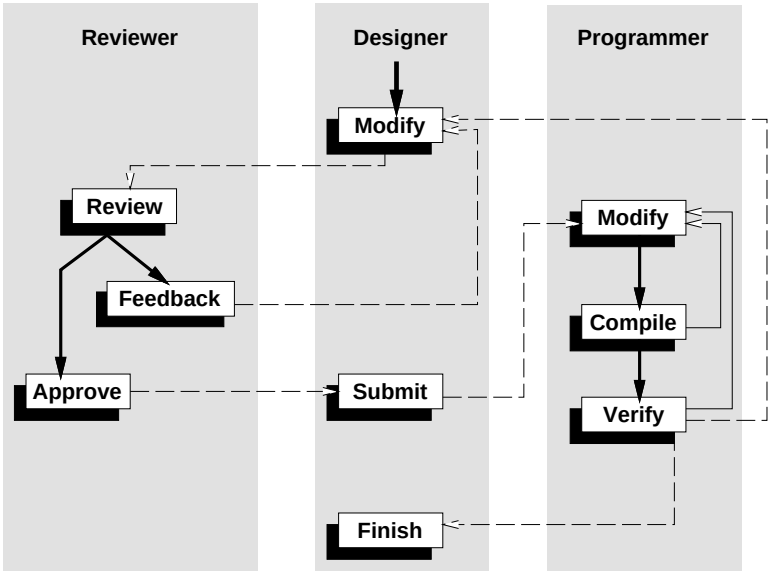

Figure 11: Partial fragment from ISPW-7

the Designer submits a modified design for review. The Reviewer either approves the design or produces feedback, and replies to the Designer who either continues to modify the design, or submits it to the Programmer. Once the Programmer has made the necessary modifications, the code is compiled and verified, and the Designer is notified of either success or failure, in which case the design is finished, or further modified, respectively.

The data model and process model which specify this process are shown in figure 13. This somewhat complex-looking set of MSL rules is abstractly pictured in figure 12, where each rule is represented by a box whose logical condition is above the box, and whose effects are below. A horizontal line of **o**'s represents a rule invoking an agent. In order for these agents to cooperate, two conflicting situations need to be handled: when the Reviewer and the Programmer read the design which the Designer is modifying. We use the same lock compatibility table and mapping table from figure 4. The MODIFY_DESIGN rule invokes a separate agent to review the design, and the locking conflict is resolved by the CORD rules in figure 14.

5 Conclusions

The approach outlined in this paper has its shortcomings. In this prototype example of cooperating agents, a new agent is created each time one is needed. In addition to wasting resources, this will sometimes incorrectly model certain situations. The MARVEL system needs to be modified slightly to allow inter-agent communication between existing agents, and this is one focus of future work. In addition, the CORD rule

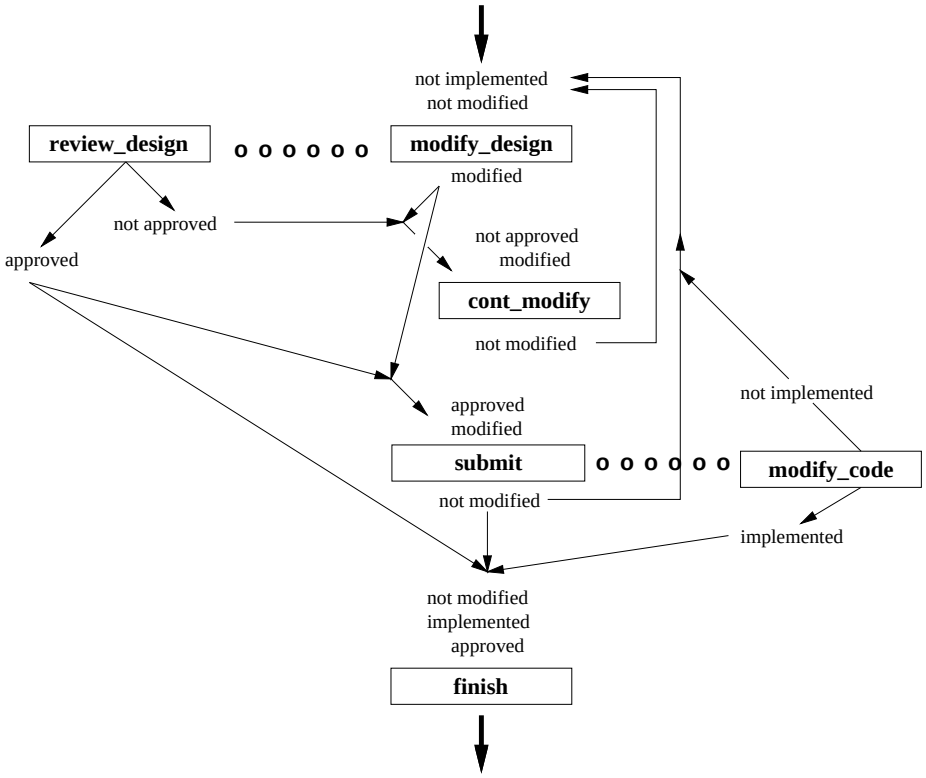

Figure 12: Cooperative solution to ISPW-7 fragment

approach needs more extensions to be able to fully differentiate between interferences of cooperating agents and independent agents. We are in the process of enhancing CORD, and this is an issue we need to address. Finally, the approach of tailoring lock modes for rules, as described in figure 4, is too general to be of much use. Making all locks compatible avoids conflicts, but introduces chaos since there would be no control over the operations. There currently exists in MARVEL a way to specifically determine lock modes for the activity section of a rule, but this needs to be extended to all symbols (and the objects bound to them) within the rule.

Even with its limitations, this paper does address, and propose solutions to, certain issues regarding cooperating agents. The primary result of this work is to show how non-serializable behavior can be controlled by a set of coordination rules to allow cooperating agents to function properly, while still preventing independent agents from interfering with each other. The coordination rule approach can be applicable to any process modeling system, since the CORD rule language is orthogonal to the underlying PML which represents the process. We are currently implementing a transaction manager component, called PERN, which will allow researchers to tailor the concurrency control of a database to suit their applications' needs.

```
OBJECT :: superclass ENTITY;
  design : DESIGN;
  code   : CODE;
end

DESIGN :: superclass ENTITY;
  contents    : text;
  modified    : (Yes, No, Initial) = Initial;
  approved    : (Yes, No, Initial) = Initial;
  implemented : (Yes, No, Initial) = Initial;
end

CODE :: superclass ENTITY;
  contents : text;
end

modify_design[?o:OBJECT]:
  (exists DESIGN ?d suchthat (member [?o.design ?d])):
  (and (or no_backward (?d.modified = No)
           no_chain    (?d.modified = Initial))
       (or no_backward (?d.implemented = No)
           no_chain    (?d.implemented = Initial)))
  { MODIFY_TOOL modify_design ?o.Name }    # invokes separate agent to review design
  (and (?d.modified    = Yes)
       (?d.implemented = No));
  no_assertion;

review_design[?o:OBJECT]:
  (exists DESIGN ?d suchthat (member [?o.design ?d])):
  { MODIFY_TOOL review_design }
  no_chain (?d.approved = No);
  no_chain (?d.approved = Yes);

hide continue_modify_design[?o:OBJECT]:
  (exists DESIGN ?d suchthat (member [?o.design ?d])):
  (and no_backward (?d.modified = Yes)
       no_backward (?d.approved  = No))
  { }
  (and no_chain (?d.approved = Initial)
                (?d.modified = No));

hide submit[?o:OBJECT]:
  (exists DESIGN ?d suchthat (member [?o.design ?d])):
  (and no_backward (?d.modified  = Yes)
       no_backward (?d.approved  = Yes))
  { MODIFY_TOOL submit ?o.Name }    # invokes separate agent
  (?d.modified = No);

modify_code[?o:OBJECT]:
  (exists CODE   ?c suchthat (member [?o.code   ?c]))):
  { MODIFY_TOOL verify_code ?o.Name }
  no_chain (?d.implemented = Yes);
  no_chain (?d.implemented = No);

hide finish[?o:OBJECT]:
  (and (exists DESIGN ?d suchthat (member [?o.design ?d]))
       (exists CODE   ?c suchthat (member [?o.code   ?c]))):
  (and no_backward (?d.modified     = No)
       no_backward (?d.implemented  = Yes)
       no_backward (?d.approved     = Yes))
  { }
  (and no_chain (?d.modified    = Initial)
       no_chain (?d.implemented = Initial)
       no_chain (?d.approved    = Initial));
```

Figure 13: MSL rules for fragment ISPW-7 solution

```
DESIGN_conflict [ DESIGN ]

bindings:
   ?t1 = holds_lock ()
   ?t2 = requested_lock ()
body:
   if (and (?t2.rule = review_design)          # A sub-agent requests to review a design
           (?t1.rule = modify_design))         # which has just been modified.
   then {
       notify(?t2, "DESIGN_conflict-1")
       ignore()
   }
   if (and (?t2.rule = modify_code)            # A sub-agent requests to review a design
           (?t1.rule = submit))  # which has just been modified
   then {
       notify(?t2, "DESIGN_conflict-2")
       ignore()
   }
end_body;
```

Figure 14: CORD rules for ISPW fragment

References

[1] Israel Z. Ben-Shaul, Gail E. Kaiser, and George T. Heineman. An architecture for multi-user software development environments. In Herbert Weber, editor, *5th ACM SIGSOFT Symposium on Software Development Environments*, pages 149–158, Tyson's Corner VA, December 1992. Special issue of *Software Engineering Notes*, 17(5), December 1992.

[2] Mark A. Gisi and Gail E. Kaiser. Extending a tool integration language. In Mark Dowson, editor, *1st International Conference on the Software Process: Manufacturing Complex Systems*, pages 218–227, Redondo Beach CA, October 1991. IEEE Computer Society Press.

[3] Dennis Heimbigner and Marc Kellner. Software process example for ISPW-7, August 1991. /pub/cs/techreports/ISPW7/ispw7.ex.ps.Z available by anonymous ftp from ftp.cs.colorado.edu.