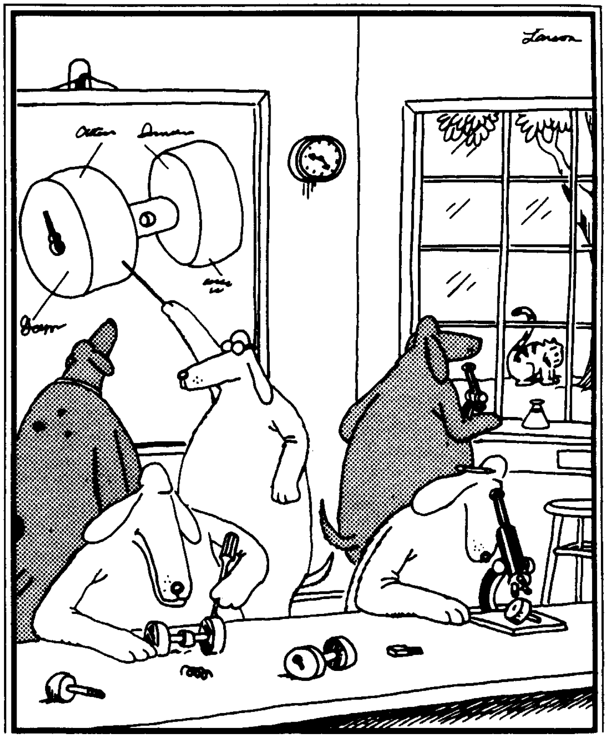
PART II

Reasoning About Mechanical Devices

The overall goal of the FONM representation theory is to support the design and implementation of computational models of mechanical reasoning, particularly improvisation and invention. As a creative problem-solving task, invention can be viewed as an integration of four reasoning tasks: (1) problem recognition, (2) problem interpretation, (3) problem evaluation, and (4) experimentation. This part of the dissertation presents two computational models constructed while developing FONM. Each model makes use of specific aspects of the FONM representation of devices, and demonstrates recognition of knowledge and inference-making capability in its reasoning domain.

○
○
○
○



Knowing how it could change the lives of canines everywhere, the dog scientists struggled diligently to understand the Doorknob Principle.

○
○
○
○

Chapter 6

Mechanical Experimentation

This chapter presents a program which performs mechanical simulation. EDEXP (EDison EXPerimenter) is a computer program that takes the symbolic representation of a simple mechanical door, mutates the door by moving a hinge, simulates the capacity of the door to move, by moving the hinge to different locations, and identifies the relationship between hinge behavior and door rotation. To accomplish this task, EDEXP implements the FONM theory of MOTION and RESTRAIN behavioral primitives.

In this chapter, the reasoning processes associated with hypothesizing, executing, and interpreting the results of mechanical experiments are discussed with respect to their dependency on how knowledge of mechanical behavior is represented and manipulated. The EDEXP model is presented and applied to a door manipulation example.

6.1 Computational Architecture

The two models presented in this and the next chapter utilize the computational architecture from the EDISON project, but have been implemented using different versions of FONM and different computational approaches. As such, those components of the architecture which are specific to each program are presented with respect to the kind of I/O and behavior that the program produces.

6.1.1 EDEXP

EDEXP implements four components of the EDISON computational architecture (Fig. 6.1): (1) a planning demon interpreter, (2) rules for performing planning analysis, mutation, and recognition (3) a planning memory, and (4) a long term memory of abstract patterns associated with devices, device plans, and device behaviors.

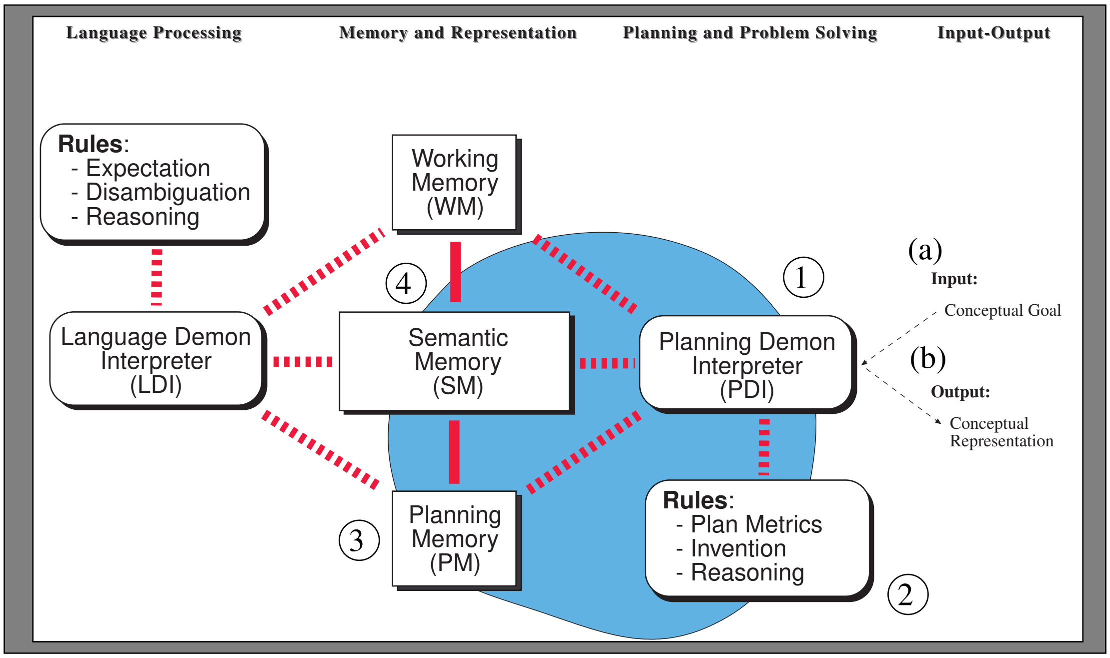


Figure 6.1 The components of EDEXP. Thick solid lines represent inter-memory access. Thick dotted lines represent control access to memories and rules from demon interpreters. Thin dotted lines represent input/output data flow.

- **Semantic Memory (SM)**. Semantic memory represents schematic knowledge of objects, events, and their causal interactions. In EDEXP, SM is comprised of frame-based rules which describe object behavior according to the FONM behavioral process definitions. During planning and behavioral simulation, input conceptualizations are compared to their counterparts in semantic memory, which are used to instantiate incomplete frames.

- **Planning Demon Interpreter (PDI)**. The planning interpreter uses demons to process goals and plans by recursively matching their subgoals to planning memory and comparing the subgoals to the overall task. The interpreter functions in a two modes: (1) experimentation, and (2) hypothesis testing. During experimentation mode, the plans associated with the experiment and device are instantiated with the intent of generalizing a hypothesis which relates the goal and device. During hypothesis testing mode, the planner applies the hypothesis to the chosen plan and device templates to attempt to prove the hypothesis.

- **Planning Memory (PM)**. During planning, EDEXP uses instantiated conceptual frames initially placed onto a memory called *planning memory*. Planning memory serves as a memory retrieval source for reasoning since the items accessed from memory represent the episodic input. Planning memory is maintained until a plan is analyzed.

- **Planning Metrics and Mutation Rules**. During planning, rules are needed to make choices and follow up on them. Three types of rules (implemented as demons) are used, for: (1) general planning, (2) hypothesis generation and testing, and (3) behavior recognition. The processes of planning and hypothesis generation and testing are implemented as demons. Demons associated with behavior recognition are associated with conceptual frames for devices in SM using procedural attachment. The PDI loads potential concepts into planning memory, and the demons associated with each concept are placed onto an associated agenda. The PDI cycles through the demons on the agenda and polls them with respect to the concepts resident in planning memory.

6.1.2 Communicating with EDEXP

EDEXP functions in a single mode. A symbolic object description and a symbolic goal (at ’a’ in Fig. 6.1) are input to the system, and a conceptual representation of the analysis results is returned. The planning demon interpreter (PDI, at 1 in Fig. 6.1) matches the conceptual input (device and task) in planning memory (PM, at 2) to plans and devices in semantic memory (SM, at 3). The mutation metrics are rules (at 4) which modify the planning behavior. Symbolic output is returned at ’b’.

6.2 Object Representation in Task Driven Experimentation

Mechanical problem solving involves the retrieval and adaptation of plans and objects to suit a particular scenario. Adaptation may require using plans for connecting or separating objects, or modifying them in various ways. Experimentation is a fundamental method used to generate and test hypotheses, and thereby obtain new information that can be applied during problem solving. Experimentation requires the ability to (a) modify and simulate objects and plans, (b) recognize new behavior, and (c) form hypotheses regarding that behavior. Each of these requirements is a function of knowledge associated with an object, its behavior, and plans which invoke its function. EDEXP models reasoning tasks which use FONM representations for device statics and low-level device dynamics.

When an object is manipulated, it undergoes state changes which affect its behavior. The representation of objects and object behavior in EDEXP must be sufficient to support evaluations of the device with respect to a goal and plan. For example, consider the simple door illustrated in Fig. 6.2. To perform simple experiments involving the manipulation of this door’s characteristics, the EDEXP representation must support the recognition of functional (i.e., can rotate) and disfunctional doors. The door at (a) is capable of rotation, meaning that it is *mobile*. The door at (b) is both mobile and perturbed. The door depicted in Fig. 6.3a represents the mutation of a single characteristic of the door depicted in Fig. 6.2a, where the upper hinge is relocated. In this case, the door is clearly incapable of rotation (i.e.,

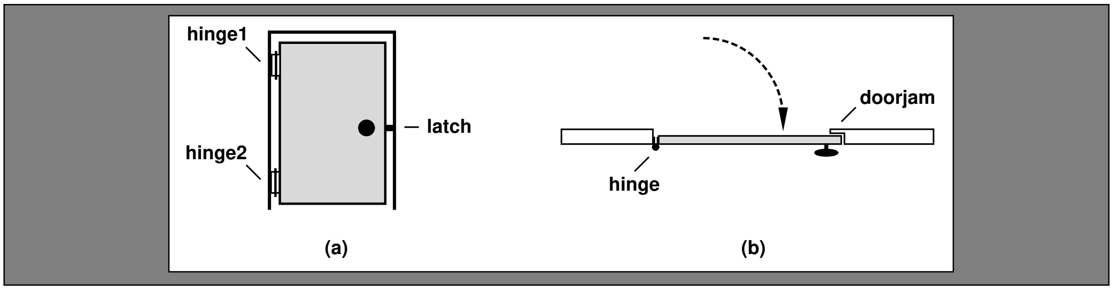


Figure 6.2 A simple functional (a) door and (b) application.

immobile). The door in Fig. 6.3b illustrates a simple modification of the perturbation depicted in Fig. 6.2b, resulting in a door which cannot rotate. The EDEXP door representation must describe the components of the door and their structural relationships. It must also describe the states of the door which change when a hinge is moved, or a force direction is changed, so that mobility and motion enablement and disablement can be recognized.

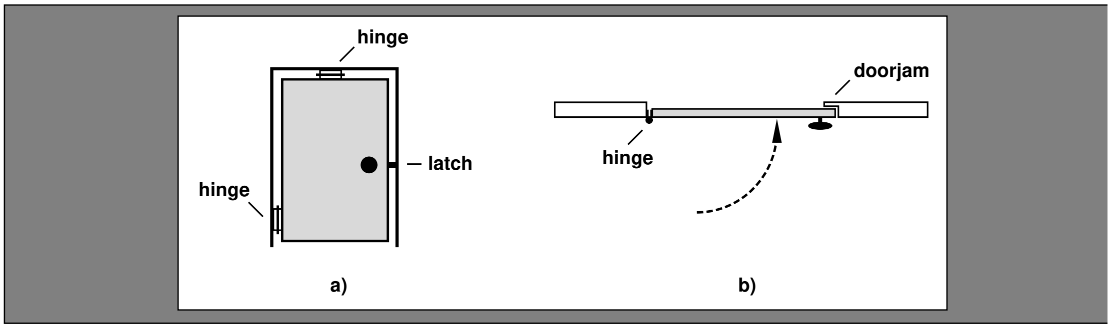


Figure 6.3 Disfunctional door (a) design and (b) application.

6.2.1 Object Statics in EDEXP

In EDEXP, objects are represented as a frame structure with six characteristics: (1) function, (2) components, (3) placement, (4) restraint, (5) mobility, and (6) connectivity. For the simple door depicted in Figs. 6.2, and 6.3, these characteristics are all related through object placement. The components, their connectivity, their mobility, and their placement are associated with portions of the FONM static representation. The processes in which the object is participating and its function are associated with portions of the FONM dynamic representation. Consider the door representation shown in Fig. 6.4, which is associated with the functional door depicted in Fig. 6.2. In this figure, the door has a single filled role (SFUNCTION), which represents a *structural* function (as opposed to an intended function, such as to control access to a space). The role is filled (at 1) with the list (M-ROTATE.1 ABOUT-L (SIDE1 FAR_LEFT)), which represents a rotational motion, about the door's longitudinal axis, with respect to its far left side. The &PART role (at 2) identifies the components of the door (DOORWAY.1 DOORSLAB.1 HINGE.2 HINGE.1), each of which is represented with separate structures. The next four sections will describe the remaining components of the door static and dynamic representation applied to DOORSLAB.1.

```
(DOOR &DOOR.1
   SFUNCTION  ((&M-ROTATE.1 ABOUT-L (SIDE1 FAR_LEFT)))      ①
   &PART <==> (&DOORWAY.1 &DOORSLAB.1 &HINGE.2 &HINGE.1))   ②
```

Figure 6.4 Representation of the door device in EDEXP. The <==> symbol denotes a link between knowledge structures. The ampersand (&) is a reference mechanism for knowledge structures.

6.2.2 Object Placement and Location

Because of the causal dependency between object connectivity and motion, the only information needed to determine door behavior is hinge placement. In FONM, an object's placement is represented by locating its center of gravity with respect to another object. In EDEXP, this is accomplished by referencing region locations of a known object, such as the doorway, or the left side of the door, and using line intersection to locate the object. Consider the door re-illustrated in Fig. 6.5, which describes the representation for hinge1 placement. A region implicitly describes a cartesian axis, so reference to a region and value locates an object axis. Fig. 6.5 depicts four regions for the doorslab: SIDE1, SIDE2, END1, and END2. In EDEXP, SIDE1 and SIDE2 refer to the doorslab width, so reference to SIDE1 in a location places an object somewhere on the ALONG-WIDTH axis of the doorslab. Similarly, any reference to END1 places an object on the ALONG-LENGTH axis of the doorslab, because END1 and END2 refer to the ALONG-LENGTH doorslab axis. In EDEXP, all sizes are referenced using a discrete scale with seven values: FAR-LEFT, LEFT, MID-LEFT, MIDDLE, MID-RIGHT, RIGHT, and FAR-RIGHT. Once a region is specified, a value from this list defines the location on the axis where the object is located. In Fig. 6.5, the value chosen to locate hinge1 on the ALONG-WIDTH dimension (w.r.t. SIDE1) is FAR-LEFT, so hinge1 is located somewhere on the left edge of the doorslab (shown at 1). By choosing a second doorslab region, which identifies another doorslab axis, a second line is identified and an intersection of these lines defines the location of hinge1. In Fig. 6.5, the second region is END1, which references the length of the doorslab, and the second value is LEFT, which specifies that hinge1 is located near the upper left edge of the doorslab (at 2). The representations for both hinge locations are shown in the lower left portion of the figure. Each location is comprised of two (region value) lists.

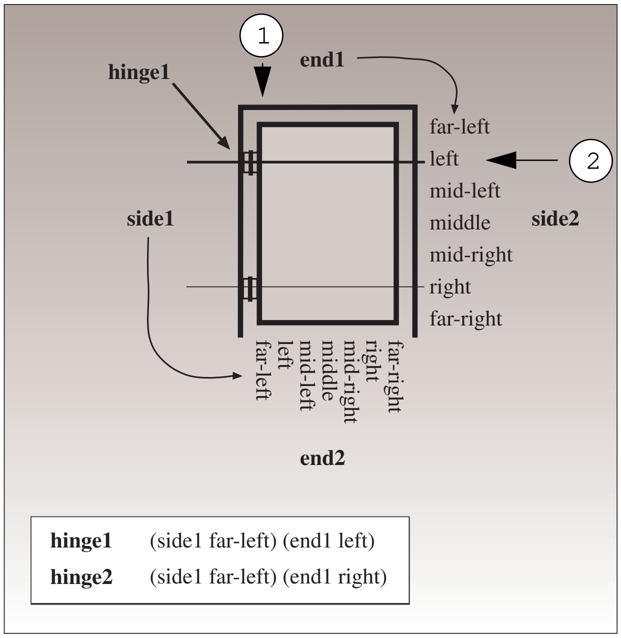
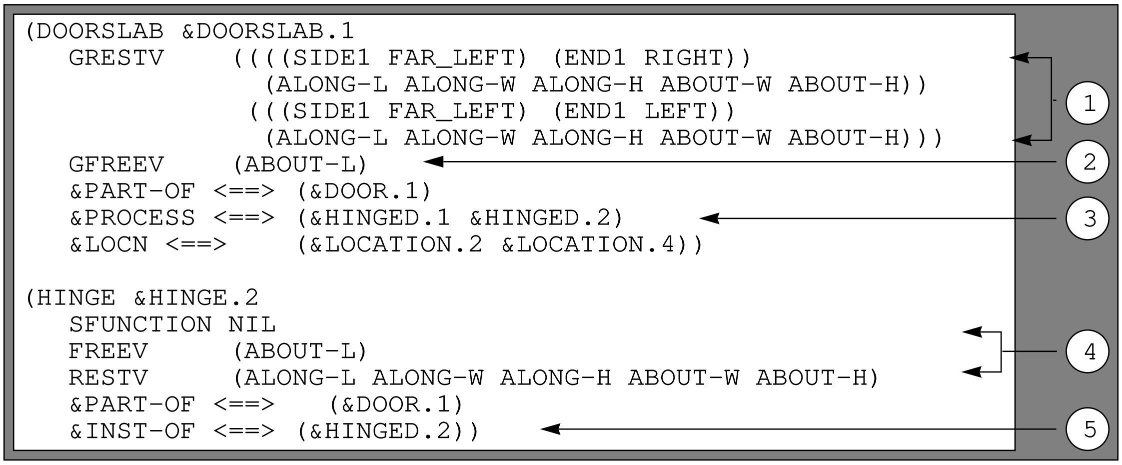


Figure 6.5 Object regions and location values in EDEXP.

6.2.3 Object Restraint and Mobility

In the door representation, only one FONM behavioral state, that for restraint, is needed to represent the door's rotational capacity. As presented on Page 25, object restraint in FONM describes locations and dimensions in which the object cannot move as a direct result of connection. Object *freedom* describes dimensions in which the object can move. Both have meaning locally, where two objects come into contact, and globally, with respect to how the object as a whole moves in space. In EDEXP, individual object restraints resulting from specific connections are represented with a list called RESTV, and their global combination is represented with a list called GRESTV. RESTV is used to identify which of the object's dimensions will support mobility. This information is also used to determine which components of a device can move and how, since connections between objects produce restraint states on the objects, and those restraint states can be analyzed individually and together to identify motion capabilities of each object. Consider the representation of the doorslab and hinge shown in Fig. 6.6.



```
(DOORSLAB &DOORSLAB.1
   GRESTV     ((((SIDE1 FAR_LEFT) (END1 RIGHT))
                (ALONG-L ALONG-W ALONG-H ABOUT-W ABOUT-H))
               (((SIDE1 FAR_LEFT) (END1 LEFT))
                (ALONG-L ALONG-W ALONG-H ABOUT-W ABOUT-H)))
   GFREEV     (ABOUT-L)
   &PART-OF <==> (&DOOR.1)
   &PROCESS <==> (&HINGED.1 &HINGED.2)
   &LOCN <==>     (&LOCATION.2 &LOCATION.4))

(HINGE &HINGE.2
   SFUNCTION NIL
   FREEV      (ABOUT-L)
   RESTV      (ALONG-L ALONG-W ALONG-H ABOUT-W ABOUT-H)
   &PART-OF <==>   (&DOOR.1)
   &INST-OF <==> (&HINGED.2))
```

Figure 6.6 Restraint and mobility representation in EDEXP. In RHAPSODY, the double-headed arrow (<=>) identifies a semantic link between knowledge structures.

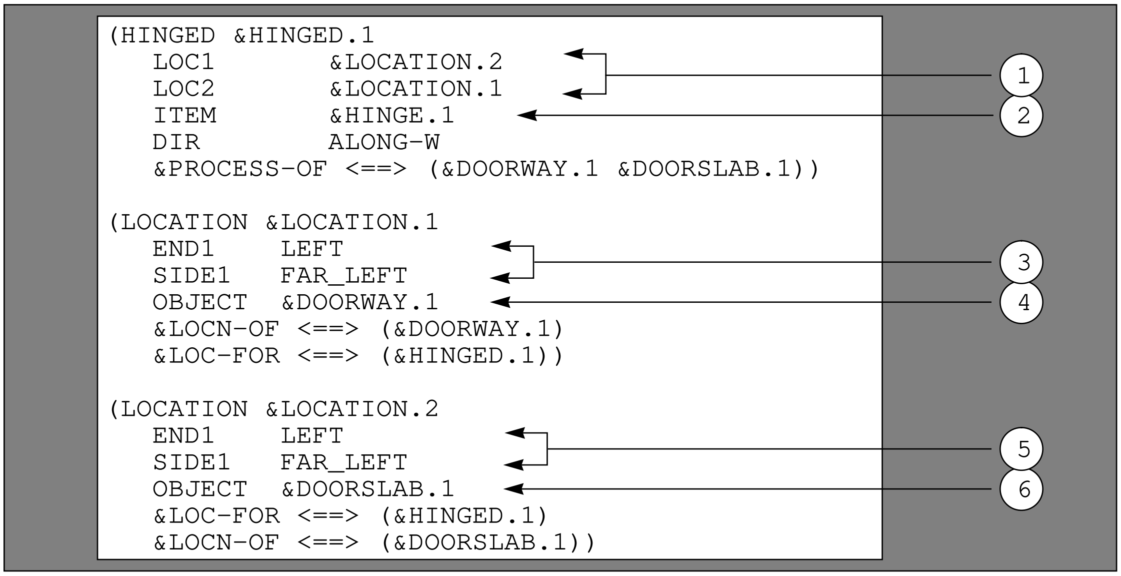
Fig. 6.6 illustrates representations for the door's doorslab and one of its hinges. The doorslab is the portion of the door that moves. Its GRESTV (at 1) is based on the rigid connections between the doorslab and its hinges, and describes two restraint states on DOORSLAB.1. The first is identified by the first sublist as a location, ((END1 FAR-RIGHT) (SIDE1 RIGHT)), and a list of dimensions which are restrained at this location (ALONG-L ALONG-W ALONG-H ABOUT-W ABOUT-H). This restraint means that DOORSLAB.1 cannot translate in any dimension at this location, and that it cannot rotate about is width or depth dimensions. The second restraint in GRESTV describes a second connection on the left side of DOORSLAB.1 which has the same dimensional restraints. The restraint on HINGE.1 is also shown in the figure as RESTV (4). The RESTV slot is used when there is only one connection. Although this is not exactly true, the doorway connection is ignored in this representation due to the rigid connection.

An object also has a slot called FREEV or GFREEV which represents mobility. An object with a single connection is represented with RESTV, and its mobility at that connection (FREEV) is the complement of RESTV. HINGE.1 is represented this way (at 4). When an object, such as DOORSLAB.1 has multiple connections whose resulting restraint states are represented with GRESTV, the global mobility is represented as GFREEV (at 2), which is the intersection of local mobilities.

6.2.4 Object Connectivity

Objects in EDEXP are related through their relative placement and their connectivity. Fig. 6.6 shows two connections, HINGED.1 and HINGED.2 (at 3 and 5), in which

DOORSLAB.1 and HINGE.1 participate. The representation for the HINGED.1 connection is shown in Fig. 6.7. HINGED.1 is a defined by the locations on participating objects,



```
(HINGED &HINGED.1
   LOC1       &LOCATION.2
   LOC2       &LOCATION.1
   ITEM       &HINGE.1
   DIR        ALONG-W
   &PROCESS-OF <==> (&DOORWAY.1 &DOORSLAB.1))

(LOCATION &LOCATION.1
   END1    LEFT
   SIDE1   FAR_LEFT
   OBJECT  &DOORWAY.1
   &LOCN-OF <==> (&DOORWAY.1)
   &LOC-FOR <==> (&HINGED.1))

(LOCATION &LOCATION.2
   END1    LEFT
   SIDE1   FAR_LEFT
   OBJECT  &DOORSLAB.1
   &LOC-FOR <==> (&HINGED.1)
   &LOCN-OF <==> (&DOORSLAB.1))
```

Figure 6.7 Object connectivity in EDEXP.

by the object which is identified with the process, and by the dimension/direction directly affected. HINGED.1 describes a FONM RESTRAIN process between HINGE.1 (at 2), DOORWAY.1 (at 4), and DOORSLAB.1 (at 6). The process is located at LOCATION.1 for DOORWAY.1 and LOCATION.2 for DOORSLAB.1. Each location is described as before, with two region/value pairs (at 3 and 5). As a RESTRAIN process instance, HINGED.1 results in the restraint (((SIDE1 FAR_LEFT) (END1 LEFT)) (ALONG-W)) in the doorslab GRESTV. The other dimensions in the restraint state are identified during program execution, when the combined effects of both hinge connections are put together into GRESTV. It should be noted that the same restraint state exists for the doorway, since the hinge is also connected to the doorway; however, the doorway is grounded (i.e., fixed) which subsumes the restraints produced by any local connectivity, so effects on the doorway are ignored.

6.3 Planning and Experimentation in EDEXP

The EDEXP planner uses rules to construct and connect knowledge structures. Demons are used to recognize motion capability in devices stored on planning memory. The planning function of the program is illustrated in Fig. 6.8.

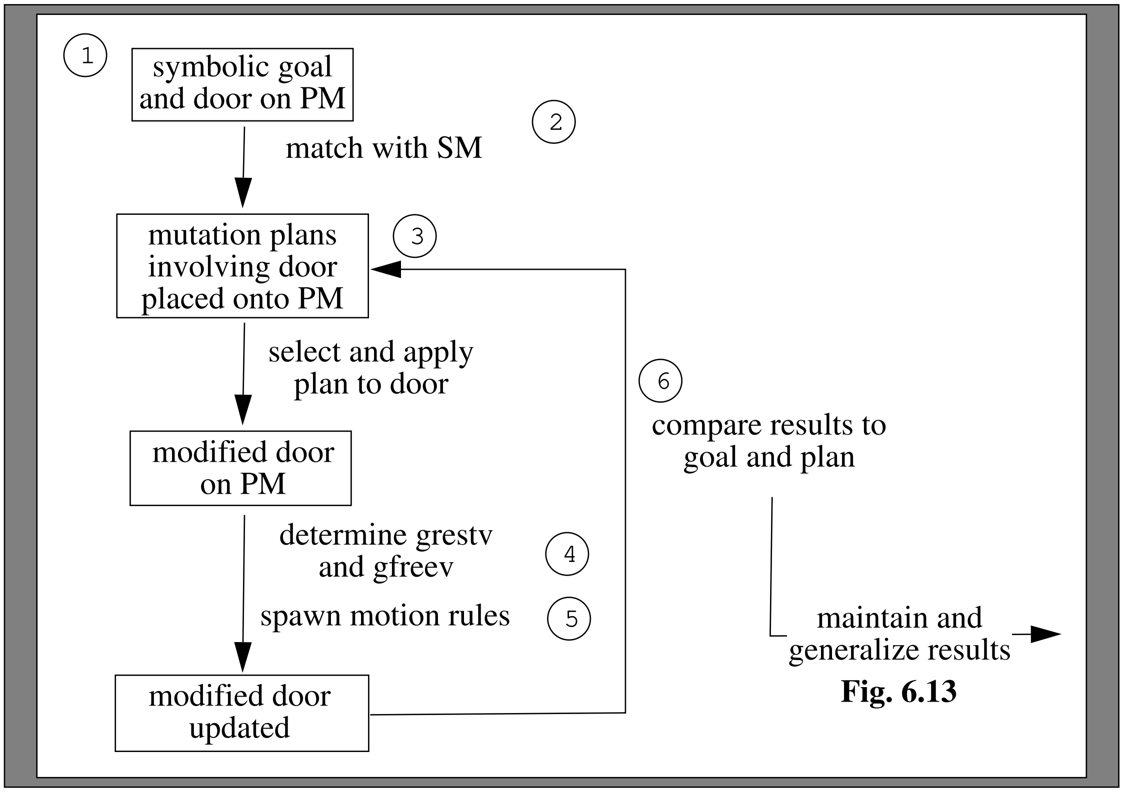


Figure 6.8 Flow chart for planning mode in EDEXP.

When the program is started, it is given a simple representation for a functional door (i.e., one the rotates) and a task to experiment with door function using mutation (at 1 in figure). These items are matched to mutation plans in semantic memory (2), producing a list of possible mutation plans on the door (3). One of the plans is selected for evaluation, and the resulting changes are effected on the door. Rules are then fired to determine the restraints and mobility of the new door as a result of changed connections (4), and demons associated with the door are generated to determine if the new door is capable of motion and, if so, to determine what type of motion is possible (5). The resulting information is used to evaluate the plan with respect to the original task (6), to select a continuing strategy, and is also used in the hypothesis generation phase of the program (shown in Fig. 6.13). The major components of planning phase: (1) plan choice, (2) plan application, and (3) plan evaluation will be described below.

6.3.1 Choice of Plans

In EDEXP, the goal is to experiment with door function (which is represented as rotational motion) using mutation. Consider the goal given to EDEXP illustrated in Fig. 6.9. There

```
(EXPERIMENT #{MOTION} . #{HOUSE-DOOR})
```

Figure 6.9 Goal representation in EDEXP. The #{symbol} notation refers to a representation object.

are three components of this goal: EXPERIMENT, MOTION, and HOUSE-DOOR. The first component (EXPERIMENT) identifies the type of task to be performed. An experiment results in new information about the device, so appropriate plans involve mutation: (a) how the device is used by modifying a known plan, or (b) the device itself by modifying some characteristic of the device's static representation. EDEXP matches the EXPERIMENT task to a list of available plans and those shown in Table 6.1: are returned.

Feature Modification Plans for a Simple Door	
TYPE HINGE	Change function of hinge(s)
MATL HINGE	Change material of hinge(s)
MOVE HINGE	Change location of hinge(s)
SIZE HINGE	Change size of hinge(s)
NUMBER HINGE	Change number of hinges
SIZE DOORSLAB	Change size of doorslab
MATL DOORSLAB	Change material of doorslab

Table 6.1: Feature modification plans for door mutation in EDEXP.

Each plan modifies a single object characteristic at a time. For example, the MOVE HINGE plan (highlighted) selects a hinge, changes the location where it connects the doorslab and doorway, and modifies the appropriate connection in the door representation.

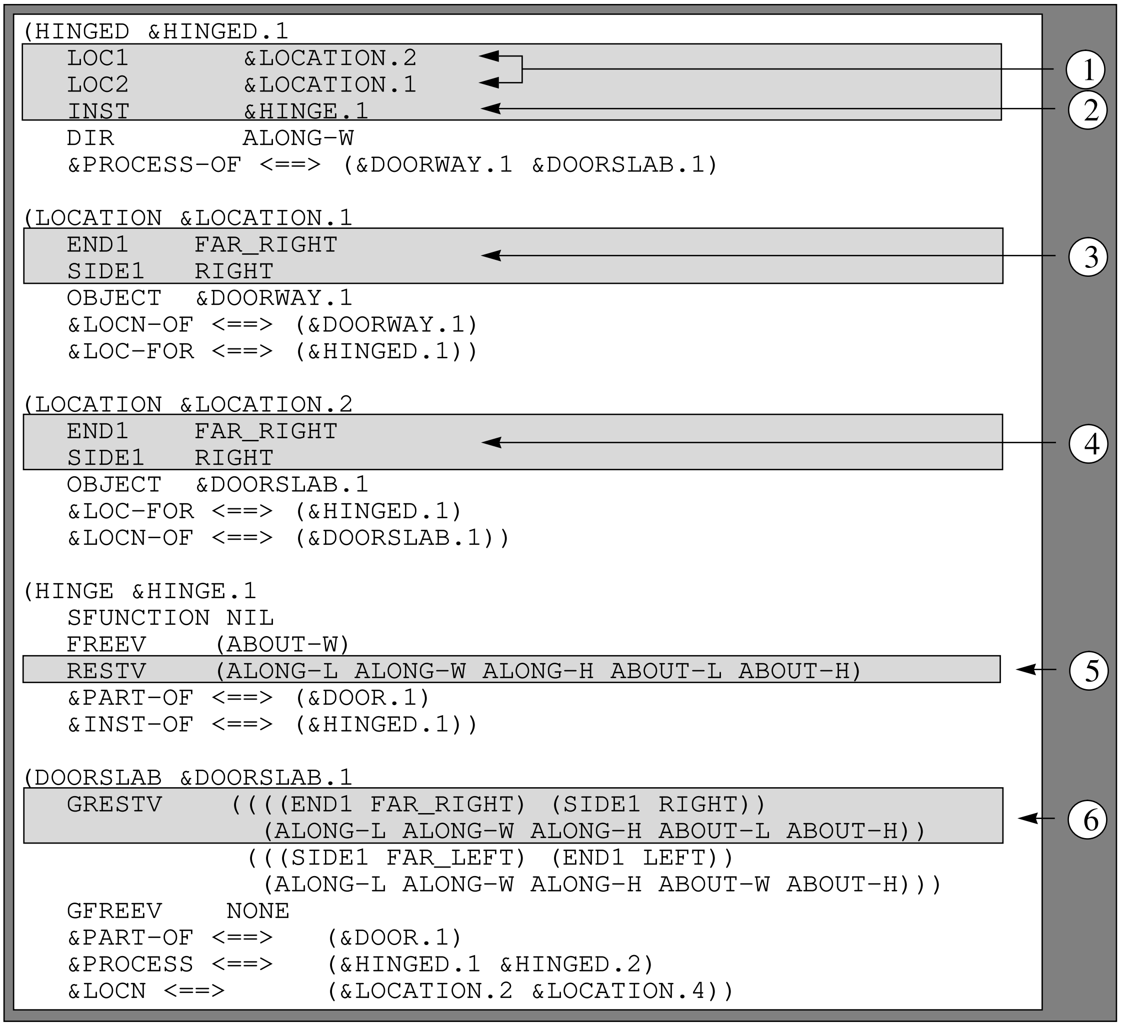
The remaining task components (MOTION and HOUSE-DOOR) define the context in which the goal is evaluated. MOTION identifies the device characteristic (i.e., FONM position state) of interest, and HOUSE-DOOR identifies the device of interest. The available plans are ordered with respect to how closely they match the goal. MOVE HINGE is the only plan in Table 6.1: which is directly related to door motion, since a change in connectivity (according to FONM device dynamics) can affect motion capability.

6.3.2 Plan Application

A feature modification plan is applied by modifying the door's static device representation. In the MOVE HINGE plan, plan application is broken into two steps: (1) a new location is selected for the hinge, and (2) the prior connection is updated to the new connection. In the current simulation, hinge locations are only allowed to be on the boundaries between the doorslab and the doorway. A new location is selected randomly from the list of unused locations available.

When a new/unique location is selected, the connection associated with the hinge being moved is updated to the new location on both the doorslab and the doorway, as shown in

Fig. 6.10. In this example, HINGE.1 is relocated to LOCATION.1 and LOCATION.2 (at 1



```
(HINGED &HINGED.1
   LOC1        &LOCATION.2
   LOC2        &LOCATION.1
   INST        &HINGE.1
   DIR         ALONG-W
   &PROCESS-OF <==> (&DOORWAY.1 &DOORSLAB.1)

(LOCATION &LOCATION.1
   END1    FAR_RIGHT
   SIDE1   RIGHT
   OBJECT  &DOORWAY.1
   &LOCN-OF <==> (&DOORWAY.1)
   &LOC-FOR <==> (&HINGED.1))

(LOCATION &LOCATION.2
   END1    FAR_RIGHT
   SIDE1   RIGHT
   OBJECT  &DOORSLAB.1
   &LOC-FOR <==> (&HINGED.1)
   &LOCN-OF <==> (&DOORSLAB.1))

(HINGE &HINGE.1
   SFUNCTION NIL
   FREEV     (ABOUT-W)
   RESTV     (ALONG-L ALONG-W ALONG-H ABOUT-L ABOUT-H)
   &PART-OF <==> (&DOOR.1)
   &INST-OF <==> (&HINGED.1))

(DOORSLAB &DOORSLAB.1
   GRESTV     ((((END1 FAR_RIGHT) (SIDE1 RIGHT))
                (ALONG-L ALONG-W ALONG-H ABOUT-L ABOUT-H))
              (((SIDE1 FAR_LEFT) (END1 LEFT))
                (ALONG-L ALONG-W ALONG-H ABOUT-W ABOUT-H)))
   GFREEV    NONE
   &PART-OF <==>   (&DOOR.1)
   &PROCESS <==>   (&HINGED.1 &HINGED.2)
   &LOCN <==>      (&LOCATION.2 &LOCATION.4))
```

Figure 6.10 Change in representation for HINGED.1 when HINGE.1 location is changed, and the associated changes in object restraint for DOORSLAB.1.

and 2 in the figure). Both locations refer to the same place on DOORSLAB.1 and DOORWAY.1 (3 and 4), namely ((END1 FAR_RIGHT) (SIDE1 RIGHT)). The restraint state on DOORSLAB.1 is the intersection of restraints at each connection, and the union of restraints across connections. DOORWAY.1 is completely restrained, so the restraint state at LOCATION.1 and LOCATION.2 is the same as that for HINGE.1 (5). The global restraint vector (GRESTV) is shown at 6. This vector is used to determine the mobility of DOORSLAB.1.

6.3.3 Plan Evaluation

A plan is evaluated by first determining the effect of its application on device dynamics, and then by determining how changes in device dynamics affect the original goal. The MOVE HINGE plan produces a change in connectivity between DOORSLAB.1, DOORWAY.1, and HINGE.1. With respect to the FONM representation of device statics and dynamics, changes in connectivity result in local and global changes to device restraint states. When the MOVE HINGE plan is applied, the new restraint states must be identified and integrated with the existing restraints from HINGE.2 before any determination of motion capability (i.e., mobility) can be made. The rules for determining object mobility given its restraints in EDEXP are shown in Table 6.2. According to this table, an object which is *grounded* (FRR1) has all degrees of freedom restrained. Grounded objects are *rigid*, or completely fixed to whatever frame of reference is involved, so they have no mobility (i.e., NONE). In the EDEXP demonstration, fixed means that DOORSLAB.1 is effectively welded to DOORWAY.1 and cannot move. The second mobility recognition rule (FRR2) combines all of the local restraints on the object. When taken as a union, they determine the object's overall, or global, restraint. If all global dimensions are restrained, then the object is rigid by virtue of being connected to objects which collectively eliminate its ability to move. The door shown in Fig. 6.3a is rigid by rule FRR2. FRR3 is similar, in that if the object is connected to grounded objects at any two points in different dimensions, then it is grounded. The same is true if the object is connected to a single grounded object at more than one location (FRR4), if there is no sharing of restraint locations. Any other object is mobile and covered by FRR5, which states that the object will have at least one dimension free to move and identifies the dimension(s). As an example of how object mobility is determined from the connections of components, consider the nonfunctional door depicted in Fig. 6.11. The door clearly cannot move, because the locations where the hinges are con-

Object Mobility Recognition Rules		
Name	**Condition (Object state)**	**Freedom**
`FRR1`	OBJECT1 is grounded	NONE = RIGID = FIXED
FRR2	the union of restraints on OBJECT1 is same as the number of dimensions defined	RIGID, by inheritance
FRR3	OBJECT1 is connected at more than one location to grounded objects	RIGID
FRR4	OBJECT1 is connected to a single object in more than one location and the intersection of dimrs is empty	RIGID
FRR5	otherwise	at least one axis is free

Table 6.2 Object restraint recognition rules in EDEXP.

nected produce restraints which work against each other.

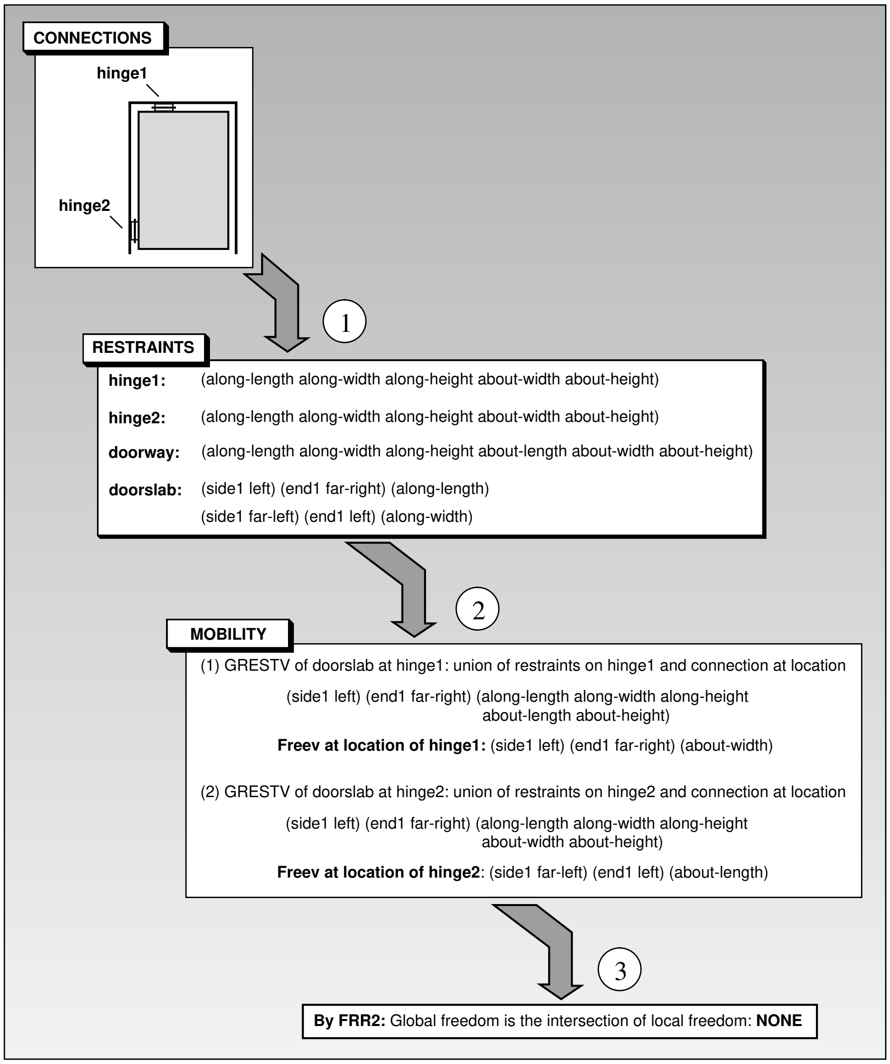


Figure 6.11 Determination of object mobility in EDEXP. Restraints are determined from connections in the door representation.

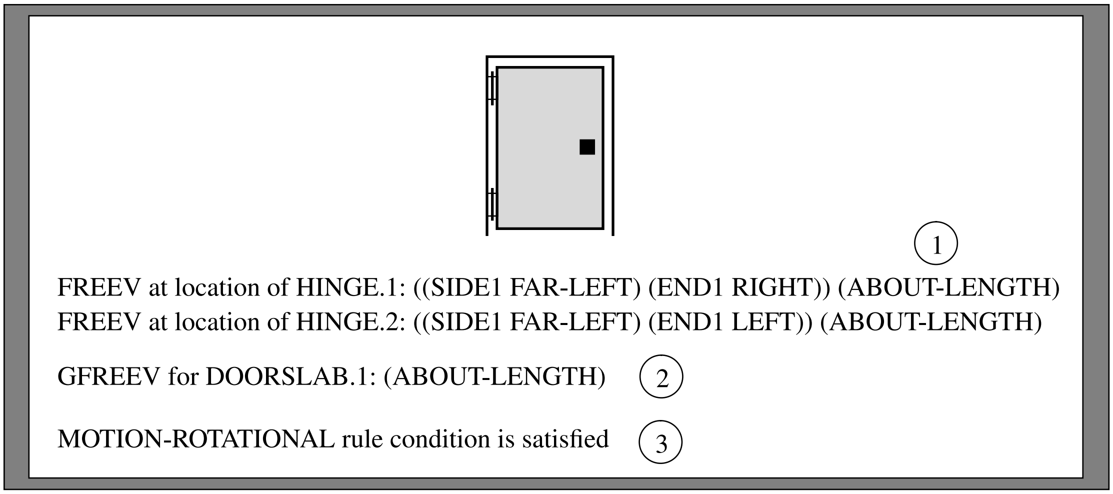
The restraints box (at 1) identifies the local restraints associated with the hinges, the fixed doorway, and the doorslab, as a result of their mutual connectivity. The mobility box (at 2) shows the combination of doorslab local restraints (i.e., GRESTV) at each hinge location. The mobility vector (FREEV) of the doorslab at each hinge location is the dimension complement of the associated restraint in GRESTV. The global mobility of the doorslab (at 3) is the intersection of the local doorslab mobilities. In this example, there is no intersection so the doorslab has a mobility of NONE and is effectively rigid.

Once the new object's mobility is determined, four motion rules are used to determine if/how the new door can move. The four rules used in this model are shown in Table 6.3.

Motion Recognition Rules		
Name	**Condition (object state)**	**Action**
MOTION-LINEAR	OBJECT1 is mobile in at least one translational dimension and no rotational dimensions.	OBJECT1 is capable of LINEAR motion
MOTION-CURVI-LINEAR	OBJECT1 is mobile in at least two translational dimensions and at least one rotational dimension.	OBJECT1 is capable of CURVI-LINEAR motion
MOTION-ROTATIONAL	OBJECT1 is mobile in at least one rotational direction but no translational dimensions.	OBJECT1 is capable of ROTATIONAL motion
MOTION-SLIDING	OBJECT1 is mobile in at least one translational dimension, and OBJECT1 is in direct normal contact to that dimension with OBJECT2	OBJECT1 is capable of SLIDING motion

Table 6.3 Motion recognition rules in EDEXP.

When the MOVE HINGE plan is applied and the new restraint and mobility vectors are identified, the motion rules thus evaluate doorslab motion by identifying the value of the mobility vector. Consider the functional door shown in Fig. 6.12.



①

FREEV at location of HINGE.1: ((SIDE1 FAR-LEFT) (END1 RIGHT)) (ABOUT-LENGTH)
FREEV at location of HINGE.2: ((SIDE1 FAR-LEFT) (END1 LEFT)) (ABOUT-LENGTH)

GFREEV for DOORSLAB.1: (ABOUT-LENGTH) ②

MOTION-ROTATIONAL rule condition is satisfied ③

Figure 6.12 Functional door, local and global mobility, and motion rule which identifies rotational capability.

The doorslab pictured Fig. 6.12 has two local mobility vectors, one for each connection location. At each location the doorslab is free to rotate about its vertical axis (i.e., ABOUT-

LENGTH, at 1). The intersection of local mobility is the doorslab mobility (2), which is the single rotational (ABOUT-LENGTH) dimension. MOTION-ROTATIONAL is the only motion type shown in Table 6.3 which can be recognized given this mobility.

Any mutation to an object which changes its restraints results in an update of the object's mobility. This only happens if the object isn't already rigid. The local behavior of rigid objects is ignored in EDEXP. For example, DOORWAY.1 is grounded, so it is rigid. After the first simulation iteration, EDEXP no longer updates mobility on the doorway, because the changed restraints will not make the doorway any more or less rigid.

Once a determination of motion capability is made for the door, the plan can be evaluated with respect to the original task, since any change in motion capability is important in the (EXPERIMENT #{MOTION} . #{HOUSE-DOOR}) goal. In the EXPERIMENT task, the same plan is applied repeatedly until a rule which relates the hinge being moved in the MOVE HINGE plan, and the door, to door motion, can be generalized.

6.3.4 Hypothesis Generation and Testing

The hypothesis generation phase in EDEXP is shown in Fig. 6.13. When the MOVE HINGE plan is applied, it is evaluated by comparing the results to the original task. If the modified door can move, then the plan application is noted as a success. Otherwise it is not-

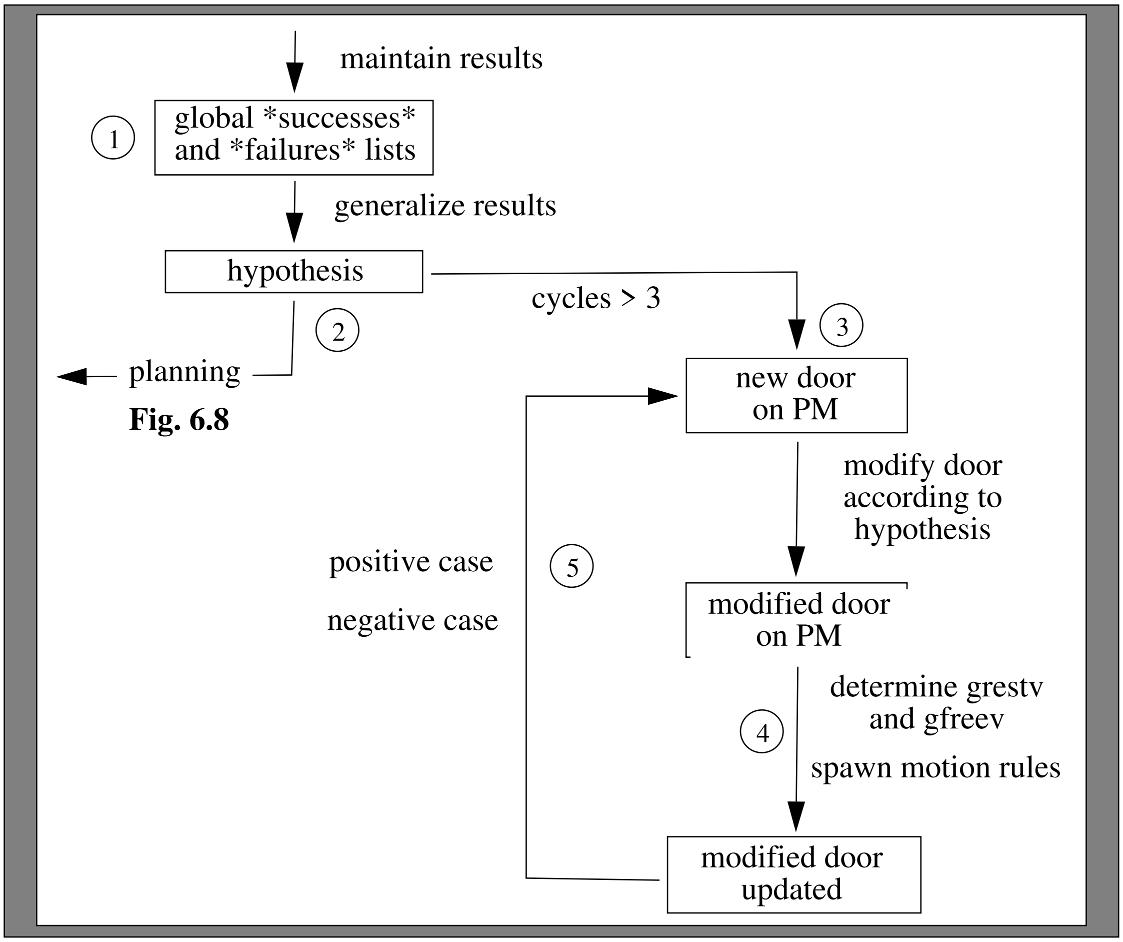


Figure 6.13 Flow chart for hypothesis generation in EDEXP.

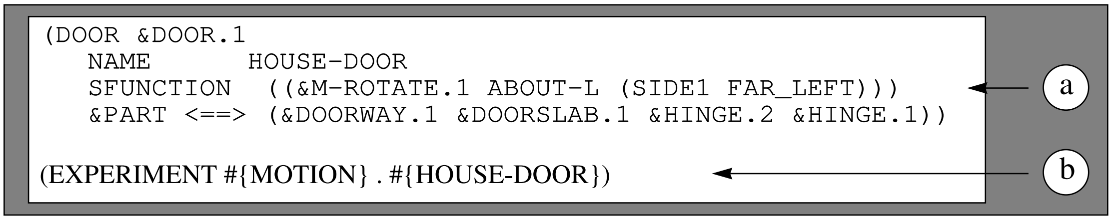
ed as a failure. Planning results are maintained for generalization (at 1). The result is a copy of the associated door representation. When two planning iterations have been made, a hypothesis is generalized by comparing the locations of hinges and whether or not they resulted in door rotational capability (2). The hypothesis takes the form of a generalized hinge location, and is generated by looking at the results of successive plan applications. If the results of consecutive planning applications are the same (success or failure), then similarities between hinge locations of the associated doors are identified. If the results of consecutive planning applications are different, then differences between hinge locations of the associated doors are identified. Hinge locations which are on both lists (similarities and differences) are of little value, so the two lists are intersected after every iteration and the remains become the new hypothesis.

If the hypothesis remains consistent between successive plan applications, a counter is incremented, otherwise the counter is initialized. When the count (arbitrarily) reaches 4 (at 3), then planning applications are stopped, and a testing phase is entered. The testing phase amounts to taking the hypothesis and using it with the MOVE HINGE plan to locate a hinge on a new door, and then simulating the change to determine if the door can rotate (4). Two tests are performed: (1) a hinge placement which results in a rotational door, and (2) a hinge placement which results in a rigid door (at 5).

The hypothesis generation and testing phase is controlled by demons which perform the updating process while plan applications are made. When the count is reached, the demons switch EDEXP into hypothesis testing phase.

6.4 EDEXP Implementation

The basis for EDEXP is the RHAPSODY demon-based AI development tool written at UCLA [Turner and Reeves, 1987]. EDEXP and RHAPSODY were originally written in T [Reese 1984], a scheme-based Lisp dialect developed at Yale, and were ported to Common Lisp in 1988. RHAPSODY provides representational semantics, the McDypar parser, a generator, demon and agenda packages, and graphics facilities. In addition to the graphic display capabilities provided by RHAPSODY, the EDEXP model has its own representation to graphic parser and graphical planning interface. The graphic parser reads representation descriptions of a door and displays an icon representation of the door components. EDEXP makes use of 15 demons to analyze the house-door example described in this chapter. An annotated trace of the example follows. Fig. 6.14a depicts the door representation used in the simulation. The conceptual goal given to the system is represented in Fig. 6.14b.



```
(DOOR &DOOR.1
   NAME      HOUSE-DOOR
   SFUNCTION  ((&M-ROTATE.1 ABOUT-L (SIDE1 FAR_LEFT)))
   &PART <==> (&DOORWAY.1 &DOORSLAB.1 &HINGE.2 &HINGE.1))

(EXPERIMENT #{MOTION} . #{HOUSE-DOOR})
```

Figure 6.14 Conceptual input to the EDEXP **Doors3** experimentation simulation.

```
> (doors-demo)
```

```
   Spawning demon: EXP-PLAN-DEMON.1
   ==========================================================
   TEST: IF   the goal is to EXPERIMENT and develop an
              understanding about some pattern on some template
              (device or part)
   ACT:  THEN try looking for a plan which will be useful
              toward this end
   ==========================================================

running agenda #{AGENDA:TMP.2}:

   Running demon: EXP-PLAN-DEMON.1

        trying to find a plan for the current goal:

                EXPERIMENT
                #{MOTION}
                #{HOUSE-DOOR}

        matching the goal patterns to template functionality:

                #{MOTION}
                #{HOUSE-DOOR}

        evaluating experimental plans
```

The planner is designed to choose among many possible plans, however, it currently prioritizes plans so that the ***MOVE*** *plan is chosen.*

```
        choosing an element to manipulate
```

Like the choice of plans, this demonstration selects ***HINGE.1*** *as the element to manipulate, and object* **location** *as the mutation characteristic.*

```
        go find structurally allowable locations on
        #{DOORWAY.1} and #{DOORSLAB.1}

        moving #{HINGE.1}

        has ((SIDE1 FAR_LEFT) (END1 MID_RIGHT)) been tried?
```

The model has a simple representation of location, based on ***SIDE1/SIDE2*** *and* ***END1*** *and* ***END2.*** *The model has a correspondingly simple notion of value* ***(FAR_LEFT, MID_LEFT, LEFT, MIDDLE, RIGHT, MID_RIGHT, FAR_RIGHT****). When the model selects a location for a hinge placement, a list of locations chosen is maintained so that the model does not continue to re-select the same location over and over again. The selection is otherwise random.*

```
        change physical locations of #{HINGE.1} on #{DOORWAY} to:

                ((SIDE1 FAR_LEFT) (END1 MID_RIGHT))
```

```
        change physical locations of #{HINGE.1} on #{DOORSLAB} to:

                ((SIDE1 FAR_LEFT) (END1 MID_RIGHT))
```

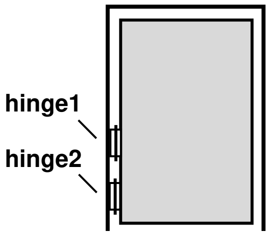


Figure 6.15 A functional door where both hinges have the same local freedom.

*Once **HINGE.1** has been relocated, the door has effectively been mutated into a new door, and the next step in the process is to evaluate the local mobility. This is accomplished with 4 demons. Mobility is determined both locally, at each hinge location, and globally. The global mobility is defined as the union of local mobilities, and a mobility is defined as a motion degree of freedom.*

```
        what is the mobility on #{DOORWAY.1}?

                local freedom: NONE

        what is the mobility on #{DOORSLAB.1}?
```

```
                global freedom: (ABOUT-L)  ◄────────── ④
                local freedom:                                        ③
      ① ─────────────►  ((SIDE1 FAR_LEFT) (END1 MID_RIGHT)) (ABOUT-L)
      ② ─────────────►  ((SIDE1 FAR_LEFT) (END1 LEFT)) (ABOUT-L)
```

*The new door has **HINGE.1** located at the position noted at (1) in the shaded portion above, and **HINGE.2** located at the position noted at (2). The dimension noted on the right (3) represents the local freedom for each hinge, and the dimension at (4) represents the intersection of local mobility on the doorslab as an object. At this point, the motion demons associated with this object are placed onto the PDI agenda to determine its motion capability.*

```
   Spawning demon: MOTION-LINEAR.1
   ==========================================================
   TEST: IF   an object is globally free in at least
              one linear axis
   ACT:  THEN that object is capable of LINEAR motion
   ==========================================================

   Spawning demon: MOTION-CURVI-LINEAR.1
   ==========================================================
   TEST: IF   an object is globally free in all directions,
   ACT:  THEN that object is capable of CURVI-LINEAR motion
   ==========================================================
```

```
   Spawning demon: MOTION-SLIDING.1
   ==========================================================
   TEST: IF   an object is globally free in at least one
              direction, and
         IF   the object is in direct contact normal to
              that direction to another object,
   ACT:  THEN that object is capable of SLIDING motion
   ==========================================================

   Spawning demon: MOTION-ROTATIONAL.1
   ==========================================================
   TEST: IF   an object is globally free in at least one
              rotational axis
   ACT:  THEN that object is capable of ROTATIONAL motion
   ==========================================================
```

*The four demons are placed onto the agenda **AGENDA:TMP.3**. The demons are polled once per cycle against the conceptual structures on PM. Since there is only one object being considered, demon interpretation amounts to seeing if the device enables any motion type and then instantiating a motion process of the appropriate type.*

```
running agenda #{AGENDA:TMP.3}:

   Running demon: MOTION-ROTATIONAL.1

        #{DOORSLAB.1} is capable of ROTATIONAL motion
```

*For this door representation and mobility, **MOTION-ROTATIONAL.1** is satisfied. The **+ACT** (positive test action) is part of the demon which tells what to do if its test condition is met, and, in the **MOTION-ROTATIONAL.1** demon, sets the sfunction slot of **DOOR.1** to **ROTATIONAL**. Part of the **+ACT** of **MOTION-ROTATIONAL.1** is to set a binary flag which is tested by all demons. During any cycle, if the flag is set, the demon will execute another component, its -**ACT** (negative test action). The -**ACT** of all four demons is to be taken off the agenda.*

```
   Killing demon: MOTION-ROTATIONAL.1 with kill value: +ACT

   Running demon: MOTION-SLIDING.1

   Killing demon: MOTION-SLIDING.1 with kill value: -ACT

   Running demon: MOTION-CURVI-LINEAR.1

   Killing demon: MOTION-CURVI-LINEAR.1 with kill value: -ACT

   Running demon: MOTION-LINEAR.1

   Killing demon: MOTION-LINEAR.1 with kill value: -ACT
```

Since a new device has been created, and its capacity to rotate has been identified, the ex-

periment has faithfully been executed (noted at 1 below) and the plan must now be evaluated. In experimentation, a plan is considered good as long as something happens. Any good plan is worth reapplying with new values to ascertain whether more information can be found. ***REPLAN-DEMON.1*** *is used to reapply the current plan.*

```
   Spawning demon: REPLAN-DEMON.1
   ==========================================================
   TEST: IF   the experiment is completed due to a
              generalization goal, and
         IF   the plan is still considered a GOOD plan,
   ACT:  THEN re-plan using the current plan.
   ==========================================================

   Killing demon: EXP-PLAN-DEMON.1 with kill value: +ACT  ◄──────── ①

   Running demon: REPLAN-DEMON.1

        make a copy of #{DOOR.1} for similarity matching

        #{DOOR.1} succeeded on goal ((#{MOTION} . #{HOUSE-DOOR}))

        MOVE is still a good plan and is being retried
```

Here the plan evaluation metric says that if the new door is capable of the same kind of motion that the original door had, then note the change made and retry the plan with new values. A side-effect of ***EX-PLAN-DEMON.1*** *is the generation of a hypothesis for how the experiments and plans affect the overall goal.* ***SIMIL-DEMON.1*** *looks for simple consistencies and inconsistencies in the results of plans evaluated, and to see if a prediction of future behavior can be made. The demon locally keeps track of the number of plan applications which have resulted in a consistent hypothesis. Hypothesis verification is attempted when the value exceeds 3.*

```
   Spawning demon: SIMIL-DEMON.1
   ==========================================================
   TEST: IF  similarities between successful planning
             attempts and unsuccessful planning attempts
             are consistent,
   ACT: THEN try to test if the rule can predict the
             outcome of a trial
   ==========================================================

running agenda #{AGENDA:TMP.4}:

   Running demon: SIMIL-DEMON.1

        re-evaluating plan: MOVE

        moving #{HINGE.1}

        has ((SIDE1 FAR_RIGHT) (END1 RIGHT)) been tried?

        change physical locations of #{HINGE.1} on #{DOORWAY} to:
```

```
                ((SIDE1 FAR_RIGHT) (END1 RIGHT))

        change physical locations of #{HINGE.1} on #{DOORSLAB} to:

                ((SIDE1 FAR_RIGHT) (END1 RIGHT))
```

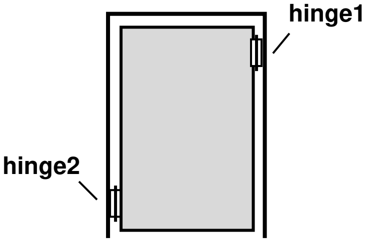


Figure 6.16 A nonfunctional door where both hinges have the same local freedom.

```
        what is the mobility on #{DOORWAY.1}?

                local freedom: NONE

        what is the mobility on #{DOORSLAB.1}?

                global freedom: NONE  <------------------------------ ①
                local freedom:
                        ((SIDE1 FAR_RIGHT) (END1 RIGHT)) (ABOUT-L)
                        ((SIDE1 FAR_LEFT) (END1 LEFT)) (ABOUT-L)
```

*The current door's mobility is evaluated with mobility rule **FRR2**. The local freedom of motion for the doorslab, at both connection locations, is **ABOUT-L**. However, when the location/mobility combinations are intersected, the result is a fixed door (1).*

```
   Spawning demon: MOTION-LINEAR.2
   Spawning demon: MOTION-CURVI-LINEAR.2
   Spawning demon: MOTION-SLIDING.2
   Spawning demon: MOTION-ROTATIONAL.2
```

*The text of these demons has already been shown, and described, as they appear during the simulation. In this and future occurrences they will be collapsed to save space. Since the global freedom of motion is restrained for the new object, none of the motion demons **+ACT**s will fire.*

```
running agenda #{AGENDA:TMP.5}:

   Running demon: MOTION-ROTATIONAL.2
   Killing demon: MOTION-ROTATIONAL.2 with kill value: -ACT
   Running demon: MOTION-SLIDING.2
   Killing demon: MOTION-SLIDING.2 with kill value: -ACT
   Running demon: MOTION-CURVI-LINEAR.2
   Killing demon: MOTION-CURVI-LINEAR.2 with kill value: -ACT
```

```
   Running demon: MOTION-LINEAR.2
   Killing demon: MOTION-LINEAR.2 with kill value: -ACT

   Spawning demon: REPLAN-DEMON.2
   ==========================================================
   TEST: IF   the experiment is completed due to a
              generalization goal, and
         IF   the plan is still considered a GOOD plan,
   ACT:  THEN re-plan using the current plan.
   ==========================================================

running agenda #{AGENDA:TMP.2}:

   Running demon: REPLAN-DEMON.2

        make a copy of #{DOOR.1} for similarity matching

        #{DOOR.1} failed on goal ((#{MOTION} . #{HOUSE-DOOR}))

        MOVE is still a good plan and is being retried
```

*In this case, the **MOVE** plan is being retried because the plan produced a design which no longer functions at all, let alone as it did originally. New versions of **REPLAN-DEMON** and **SIMIL-DEMON** are created for every planning cycle. The **SIMIL-DEMON** keeps track of the overall number of experimental iterations and the number of plans which have been evaluated successfully. When a device plan succeeds with respect to the task, EDEXP looks for similarities between the current and previous device (i.e., hinge locations). When a device plan fails with respect to the task, EDEXP looks for differences from the current and previous device. **SIMIL-DEMON** keeps track of the similarities and differences and slowly evolves a hypothesis of the experimental results.*

```
   Spawning demon: SIMIL-DEMON.2
   ==========================================================
   TEST: IF  similarities between successful planning
             attempts and unsuccessful planning attempts
             are consistent,
   ACT: THEN try to test if the rule can predict the
             outcome of a trial
   ==========================================================

running agenda #{AGENDA:TMP.6}:

   Running demon: SIMIL-DEMON.2

        re-evaluating plan: MOVE

        moving #{HINGE.1}

        has ((END1 FAR_RIGHT) (SIDE1 MID_LEFT)) been tried?

        change physical locations of #{HINGE.1} on #{DOORWAY} to:
```

```
                ((END1 FAR_RIGHT) (SIDE1 MID_LEFT))

        change physical locations of #{HINGE.1} on #{DOORSLAB} to:

                ((END1 FAR_RIGHT) (SIDE1 MID_LEFT))
```

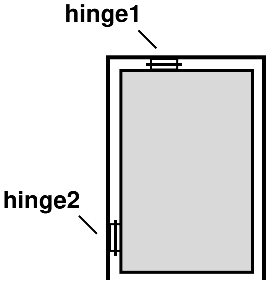


Figure 6.17 Nonfunctional door where hinges have dissimilar local freedom.

```
        what is the mobility on #{DOORWAY.1}?

                local freedom: NONE

        what is the mobility on #{DOORSLAB.1}?

                global freedom: NONE  ◄──────────────────────── ①
```

The current door's mobility is evaluated with mobility rule ***FRR3****. The doorslab is connected at different locations, and in different dimensions, to a grounded object (DOORWAY.1), so there is no global mobility for DOORSLAB.1 (1).*

```
   Spawning demon: MOTION-LINEAR.3
   Spawning demon: MOTION-CURVI-LINEAR.3
   Spawning demon: MOTION-SLIDING.3
   Spawning demon: MOTION-ROTATIONAL.3

running agenda #{AGENDA:TMP.7}:

   Killing demon: MOTION-ROTATIONAL.3 with kill value: -ACT
   Killing demon: MOTION-SLIDING.3 with kill value: -ACT
   Killing demon: MOTION-CURVI-LINEAR.3 with kill value: -ACT
   Killing demon: MOTION-LINEAR.3 with kill value: -ACT

   Spawning demon: REPLAN-DEMON.3
   ==========================================================
   TEST: IF   the experiment is completed due to a
              generalization goal, and
         IF   the plan is still considered a GOOD plan,
   ACT:  THEN re-plan using the current plan.
   ==========================================================

running agenda #{AGENDA:TMP.2}:
```

```
   Running demon: REPLAN-DEMON.3

        make a copy of #{DOOR.1} for similarity matching

        #{DOOR.1} failed on goal ((#{MOTION} . #{HOUSE-DOOR}))

        MOVE is still a good plan and is being retried
```

Once again, the failure of the plan due to moving HINGE.1 to a location where the door cannot rotate is valuable, since the new location was unique and increases the number and type of locations which can be used for hypothesis generalization.

```
   Spawning demon: SIMIL-DEMON.3
   ==========================================================
   TEST: IF  similarities between successful planning
             attempts and unsuccessful planning attempts
             are consistent,
   ACT: THEN try to test if the rule can predict the
             outcome of a trial
   ==========================================================

running agenda #{AGENDA:TMP.8}:

   Running demon: SIMIL-DEMON.3

        re-evaluating plan: MOVE

        moving #{HINGE.1}

        has ((SIDE1 FAR_LEFT) (END1 RIGHT)) been tried?

        change physical locations of #{HINGE.1} on #{DOORWAY} to:

                ((SIDE1 FAR_LEFT) (END1 RIGHT))

        change physical locations of #{HINGE.1} on #{DOORSLAB} to:

                ((SIDE1 FAR_LEFT) (END1 RIGHT))
```

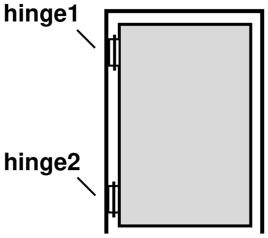


Figure 6.18 Normal functional door.

EDEXP tries a location which results in the original door because its attempt to construct a hypothesis does not maintain the original device hinge locations as part of the history, so *this set of locations is considered valid in this demonstration.*

```
        what is the mobility on #{DOORWAY.1}?

                local freedom: NONE

        what is the mobility on #{DOORSLAB.1}?
```

```
                global freedom: (ABOUT-L)  ◄──────────────────────── ①
                local freedom:
                        ((SIDE1 FAR_LEFT) (END1 RIGHT)) (ABOUT-L)
                        ((SIDE1 FAR_LEFT) (END1 LEFT)) (ABOUT-L)
```

*The current door's mobility is evaluated with mobility rule **FRR5**. The doorslab is connected at different locations, but in the same dimensions, to a grounded object (DOORWAY.1), so there is both local and global rotational mobility for DOORSLAB.1 (1). As a result, the **+ACT** of **MOTION-ROTATIONAL.4** fires.*

```
   Spawning demon: MOTION-LINEAR.4
   Spawning demon: MOTION-CURVI-LINEAR.4
   Spawning demon: MOTION-SLIDING.4
   Spawning demon: MOTION-ROTATIONAL.4

running agenda #{AGENDA:TMP.9}:

   Running demon: MOTION-ROTATIONAL.4

        #{DOORSLAB.1} is capable of ROTATIONAL motion

   Killing demon: MOTION-ROTATIONAL.4 with kill value: +ACT
   Killing demon: MOTION-SLIDING.4 with kill value: -ACT
   Killing demon: MOTION-CURVI-LINEAR.4 with kill value: -ACT
   Killing demon: MOTION-LINEAR.4 with kill value: -ACT

   Spawning demon: REPLAN-DEMON.4
   ==========================================================
   TEST: IF   the experiment is completed due to a
              generalization goal, and
         IF   the plan is still considered a GOOD plan,
   ACT:  THEN re-plan using the current plan.
   ==========================================================

running agenda #{AGENDA:TMP.2}:

   Running demon: REPLAN-DEMON.4

        make a copy of #{DOOR.1} for similarity matching

        #{DOOR.1} succeeded on goal ((#{MOTION} . #{HOUSE-DOOR}))

        MOVE is still a good plan and is being retried

   Spawning demon: SIMIL-DEMON.4
```

```
   ==========================================================
   TEST: IF  similarities between successful planning
             attempts and unsuccessful planning attempts
             are consistent,
   ACT: THEN try to test if the rule can predict the
             outcome of a trial
   ==========================================================

running agenda #{AGENDA:TMP.10}:

   Running demon: SIMIL-DEMON.4

   Spawning demon: TESTGEN-DEMON.1
   ==========================================================
   TEST: IF   strength is high enough to execute,
   ACT:  THEN execute a plan for trying to test
              the generalized concept by simulation
   ==========================================================
        spawning a demon to try to test generalized concept
```

*The demon **SIMIL-DEMON.4** succeeds (highlighted below) when the number of iterations with a consistent hypothesis is at least 4 and the number of planning applications resulting in successes is at least 2. In this case, the successful planning applications alternate with the unsuccessful planning applications, but the hypothesis remains consistent. On the fifth iteration, the demon **TESTGEN-DEMON.1** is spawned. **TESTGEN-DEMON.1** tests the hypothesis generalized by **SIMIL-DEMON** by attempting to generate device instances which are both successful and unsuccessful using the hinge location hypothesis to select HINGE.1 locations.*

```
   Killing demon: SIMIL-DEMON.4 with kill value: +ACT

        re-evaluating plan: MOVE

        moving #{HINGE.1}

        has ((END1 FAR_RIGHT) (SIDE1 RIGHT)) been tried?

        change physical locations of #{HINGE.1} on #{DOORWAY} to:

                ((END1 FAR_RIGHT) (SIDE1 RIGHT))

        change physical locations of #{HINGE.1} on #{DOORSLAB} to:

                ((END1 FAR_RIGHT) (SIDE1 RIGHT))
```

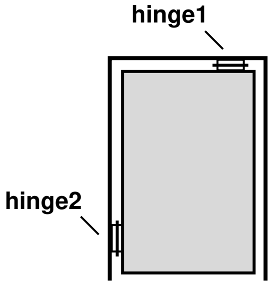


Figure 6.19 Disfunctional door where hinges have dissimilar local freedoms.

```
        what is the mobility on #{DOORWAY.1}?

                local freedom: NONE
```

```
        what is the mobility on #{DOORSLAB.1}?

                global freedom: NONE    ◄──────────────────────── ①
```

*The current door's mobility is evaluated with mobility rule **FRR3**. The doorslab is connected at different locations, but in the same dimensions, to a grounded object (DOORWAY.1), so there is both local and global rotational restraint for DOORSLAB.1 (1). None of the motion demons **+ACT**s fire.*

```
   Spawning demon: MOTION-LINEAR.5
   Spawning demon: MOTION-CURVI-LINEAR.5
   Spawning demon: MOTION-SLIDING.5
   Spawning demon: MOTION-ROTATIONAL.5

running agenda #{AGENDA:TMP.11}:

   Killing demon: MOTION-ROTATIONAL.5 with kill value: -ACT
   Killing demon: MOTION-SLIDING.5 with kill value: -ACT
   Killing demon: MOTION-CURVI-LINEAR.5 with kill value: -ACT
   Killing demon: MOTION-LINEAR.5 with kill value: -ACT

   Spawning demon: REPLAN-DEMON.5
   ==========================================================
   TEST: IF   the experiment is completed due to a
              generalization goal, and
         IF   the plan is still considered a GOOD plan,
   ACT:  THEN re-plan using the current plan.
   ==========================================================
```

*When **TESTGEN:DEMON** is spawned initially, with the idea that, when enough evidence exists, in the form of consistency in the hypothesis generated over multiple plan applications, an evaluation of the generalized device similarities and differences will be made. Since there should be competition between the desire to obtain more data and the desire to* *test a hypothesis, the* ***TESTGEN*** *demon's priority is raised when there are successive planning attempts which produce the same hypothesis. This priority is a component of all demons, but is only manipulated in this demon. With each consistent hypothesis plan application cycle the demon's priority is incremented. When the number of consistent hypotheses reaches three,* ***TESTGEN****'s priority places it at the top of the evaluation queue. If successive planning attempts produce inconsistent hypotheses, then* ***TESTGEN****'s priority is returned to its nominal value. When* ***TESTGEN-DEMON.1*** *is fired, it performs two tests. The first is to generate a door with a hinge arrangement which will succeed according to the hypothesis. The second is to generate a door with a hinge arrangement which will fail according to the hypothesis.*

```
running agenda #{AGENDA:TMP.2}:
```

```
   Running demon: TESTGEN-DEMON.1
        try testing (SIDE1 FAR_LEFT) LOCATION for #{HINGE}
```

The hypothesis generated is ***(SIDE1 FAR_LEFT)****, which is a template for the location variables which have been generalized from the specific planning instances. The meaning of this hypothesis is that any hinge location is acceptable as long as it matches this template. Since no reference to any location along the side is specified, and value is acceptable.*

```
        moving #{HINGE.11}
```

```
        has ((SIDE1 FAR_RIGHT) (END1 MID_LEFT)) been tried?

        has ((SIDE1 FAR_RIGHT) (END1 MID_LEFT)) been tried?
```

```
        has ((SIDE1 FAR_RIGHT) (END1 LEFT)) been tried?

        has ((SIDE1 FAR_RIGHT) (END1 MIDDLE)) been tried?

        has ((SIDE1 FAR_RIGHT) (END1 RIGHT)) been tried?

        has ((END1 FAR_LEFT) (SIDE1 LEFT)) been tried?

        has ((END1 FAR_LEFT) (SIDE1 MIDDLE)) been tried?

        has ((END1 FAR_LEFT) (SIDE1 LEFT)) been tried?

        has ((SIDE1 FAR_RIGHT) (END1 LEFT)) been tried?

        has ((END1 FAR_RIGHT) (SIDE1 MID_RIGHT)) been tried?

        has ((END1 FAR_LEFT) (SIDE1 LEFT)) been tried?

        has ((END1 FAR_LEFT) (SIDE1 MIDDLE)) been tried?

        has ((END1 FAR_RIGHT) (SIDE1 LEFT)) been tried?

        has ((SIDE1 FAR_LEFT) (END1 MID_RIGHT)) been tried?
```

When hypothesis testing is begun, a copy of the door representation is made and placed onto planning memory. The new hinge1 instance is ***HINGE.11*** *In all previous* ***MOVE HINGE*** *plan applications, a list of locations for HINGE.1 that have been tried was maintained, but the list of locations tried was small compared to the list of available locations (7 on each side, 7 on each end). So random selection generally produced a new location on the first try. In the hypothesis testing phase, only those locations satisfying the constraint can be chosen, so the random selection and comparison takes more attempts. The reason for the identical attempt (highlighted above) is that only those locations which have been tried are maintained, so it is possible to ask the same question twice.*

```
        change physical locations of #{HINGE.11} on #{DOORWAY} to:

                ((SIDE1 FAR_LEFT) (END1 MID_RIGHT))

        change physical locations of #{HINGE.11} on #{DOORSLAB} to:

                ((SIDE1 FAR_LEFT) (END1 MID_RIGHT))
```

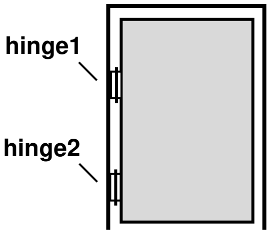


Figure 6.20 Generated functional door from hypothesis.

In the first try, a location for ***HINGE.11*** *is chosen,* **`((SIDE1 FAR_LEFT) (END1 MID_RIGHT))`**, *using the hypothesis, which should yield a door which will rotate.*

```
        what is the mobility on #{DOORWAY.6}?

                local freedom: NONE

        what is the mobility on #{DOORSLAB.6}?

                global freedom: (ABOUT-L)
                local freedom:
                        ((SIDE1 FAR_LEFT) (END1 MID_RIGHT)) (ABOUT-L)
                      ((SIDE1 FAR_LEFT) (END1 LEFT)) (ABOUT-L)
```

Which results in a global mobility for ***DOORSLAB.6*** *in the dimension* ***(ABOUT-L).***

```
   Spawning demon: MOTION-LINEAR.6
   Spawning demon: MOTION-CURVI-LINEAR.6
   Spawning demon: MOTION-SLIDING.6
   Spawning demon: MOTION-ROTATIONAL.6

running agenda #{AGENDA:TMP.12}:
```

```
Running demon: MOTION-ROTATIONAL.6

     #{DOORSLAB.6} is capable of ROTATIONAL motion

Killing demon: MOTION-ROTATIONAL.6 with kill value: +ACT
Killing demon: MOTION-SLIDING.6 with kill value: -ACT
Killing demon: MOTION-CURVI-LINEAR.6 with kill value: -ACT
Killing demon: MOTION-LINEAR.6 with kill value: -ACT
```

```
     POSITIVE test succeeded with (SID FAR) LOCATION for #{HINGE}
     attempt NEGATED test with (SID FAR) LOCATION for #{HINGE}
```

The first portion of the ***TESTGEN-DEMON.1*** *test shows that the generalized hypothesis* ***(SIDE1 FAR_LEFT)*** *is valid. The second portion of the test does the same thing on a negative instance of the hypothesis applied to the new door.*

```
     moving #{HINGE.11}

     has ((END1 FAR_LEFT) (SIDE1 MID_LEFT)) been tried?
```

It takes much fewer attempts to find a negative instance of the hypothesis, which is any location which violates the ***(SIDE1 FAR_LEFT)*** *constraint.*

```
     change physical locations of #{HINGE.11} on #{DOORWAY} to:

             ((END1 FAR_LEFT) (SIDE1 MID_LEFT))

     change physical locations of #{HINGE.11} on #{DOORSLAB} to:

             ((END1 FAR_LEFT) (SIDE1 MID_LEFT))
```

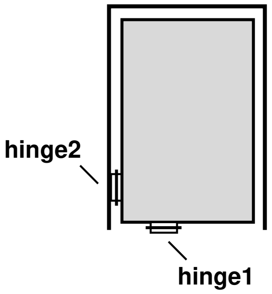


Figure 6.21 Generated disfunctional door from hypothesis.

```
     what is the mobility on #{DOORWAY.6}?

             local freedom: NONE
```

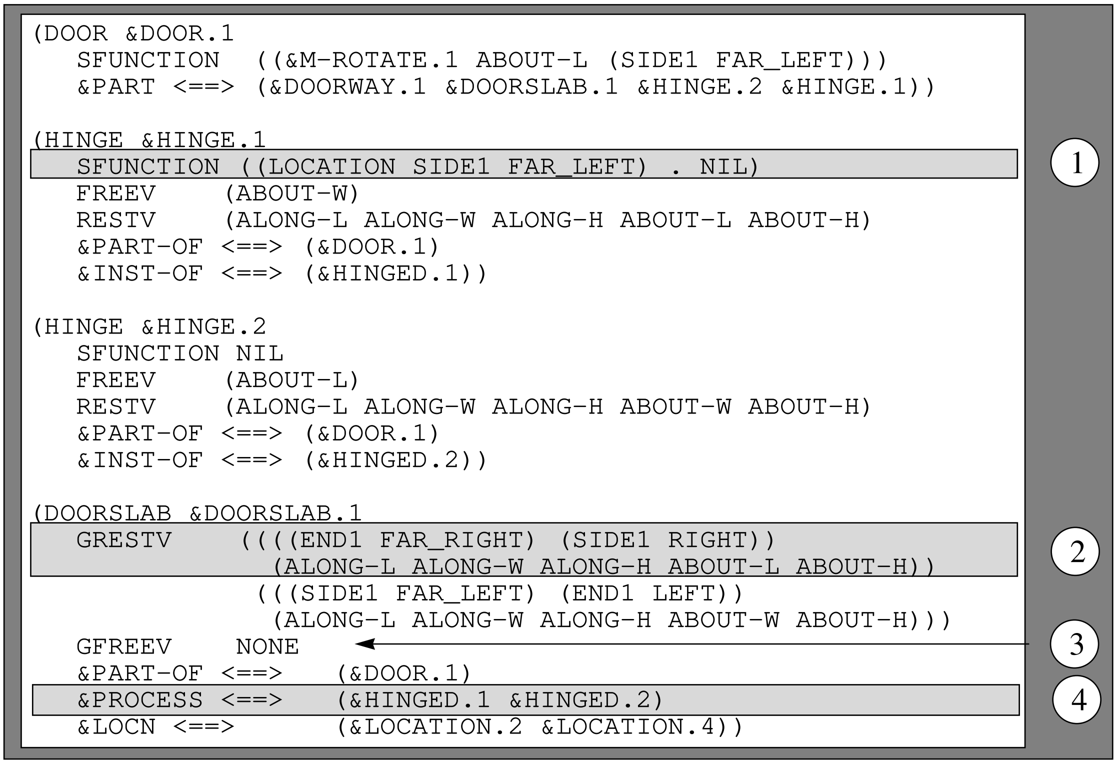
```
     what is the mobility on #{DOORSLAB.6}?

             global freedom: NONE
```

*The negative instance results in no global mobility for **DOORSLAB.6**.*

```
   Spawning demon: MOTION-LINEAR.7
   Spawning demon: MOTION-CURVI-LINEAR.7
   Spawning demon: MOTION-SLIDING.7
   Spawning demon: MOTION-ROTATIONAL.7

running agenda #{AGENDA:TMP.13}:

   Killing demon: MOTION-ROTATIONAL.7 with kill value: -ACT
   Killing demon: MOTION-SLIDING.7 with kill value: -ACT
   Killing demon: MOTION-CURVI-LINEAR.7 with kill value: -ACT
   Killing demon: MOTION-LINEAR.7 with kill value: -ACT

        succeeded test  ◄─────────────────────────────────────────  ①

        updating SFUNCTION of #{HINGE.1} to (LOCATION SID FAR)      ②

   Killing demon: TESTGEN-DEMON.1 with kill value: +ACT
   Killing demon: REPLAN-DEMON.4 with kill value: +ACT
   Killing demon: REPLAN-DEMON.3 with kill value: +ACT
   Killing demon: REPLAN-DEMON.2 with kill value: +ACT
   Killing demon: REPLAN-DEMON.1 with kill value: +ACT
#T
>
```

*The result of **TESTGEN-DEMON.1** is a success (**+ACT** fired), because both the positive and negative hypothesis applications produce doors with the intended structural function (1). As a result, the hypothesis (2) is saved as an attribute of **HINGE.1**.*

The final representation for DOOR.1 is shown in Fig. 6.22 and Fig. 6.23.

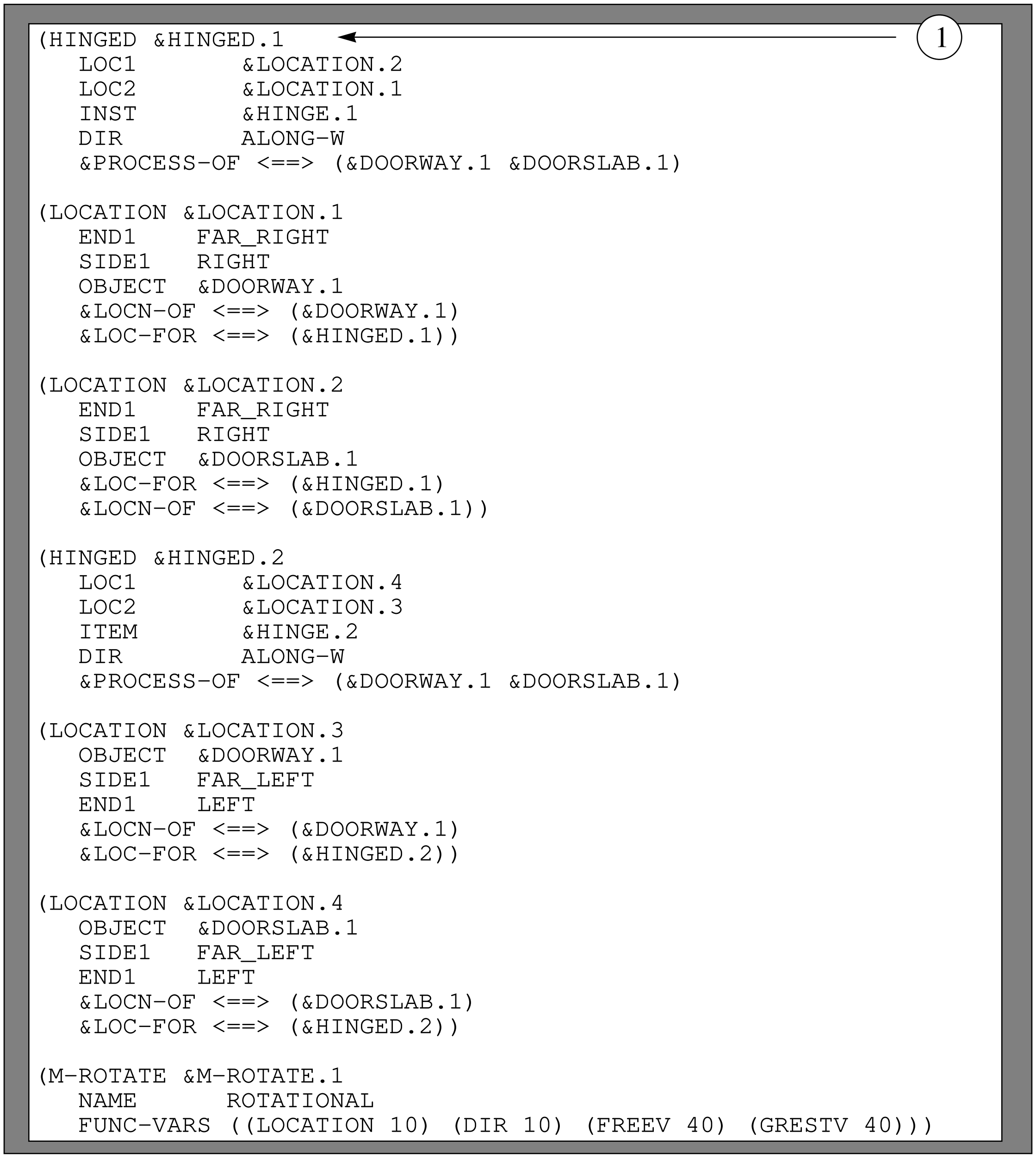


```
(DOOR &DOOR.1
   SFUNCTION  ((&M-ROTATE.1 ABOUT-L (SIDE1 FAR_LEFT)))
   &PART <==> (&DOORWAY.1 &DOORSLAB.1 &HINGE.2 &HINGE.1))

(HINGE &HINGE.1
   SFUNCTION ((LOCATION SIDE1 FAR_LEFT) . NIL)
   FREEV    (ABOUT-W)
   RESTV    (ALONG-L ALONG-W ALONG-H ABOUT-L ABOUT-H)
   &PART-OF <==> (&DOOR.1)
   &INST-OF <==> (&HINGED.1))

(HINGE &HINGE.2
   SFUNCTION NIL
   FREEV    (ABOUT-L)
   RESTV    (ALONG-L ALONG-W ALONG-H ABOUT-W ABOUT-H)
   &PART-OF <==> (&DOOR.1)
   &INST-OF <==> (&HINGED.2))

(DOORSLAB &DOORSLAB.1
   GRESTV    ((((END1 FAR_RIGHT) (SIDE1 RIGHT))
               (ALONG-L ALONG-W ALONG-H ABOUT-L ABOUT-H))
              (((SIDE1 FAR_LEFT) (END1 LEFT))
               (ALONG-L ALONG-W ALONG-H ABOUT-W ABOUT-H)))
   GFREEV    NONE
   &PART-OF <==>   (&DOOR.1)
   &PROCESS <==>   (&HINGED.1 &HINGED.2)
   &LOCN <==>      (&LOCATION.2 &LOCATION.4))
```

Figure 6.22 Doors demo final representation.

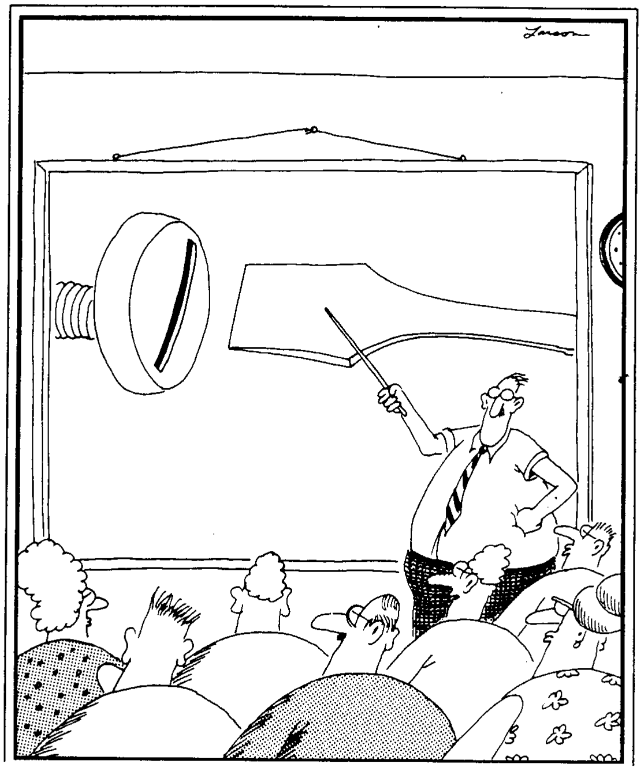
The function of the door illustrated, and its connectivity, is that of the door illustrated in Fig. 6.19, which was the last door representation generated in the planning phase. HINGE.1 has a structural function which has been updated from the hypothesis to include the location constraint (1). The DOORSLAB.1 global restraint vector (2) shows the location of HINGE.1 for that door ((END1 FAR_RIGHT) (SIDE1 RIGHT)) mutation, and DOORSLAB.1 has no mobility (3). The connections are represented as process frames (HINGED.1 and HINGED.2, at 4 in Fig. 6.22 and at 1 in Fig. 6.23). In EDEXP, these structures do not represent the dynamics associated with the associated FONM RESTRAIN process, but they are directly related to the local restraint states seen in the restraint vector GRESTV. The process has two locations, an object, and a dimension. The locations are defined with respect to regions on the object. The restraints are calculated procedurally in

EDEXP.



```
(HINGED &HINGED.1
   LOC1        &LOCATION.2
   LOC2        &LOCATION.1
   INST        &HINGE.1
   DIR         ALONG-W
   &PROCESS-OF <==> (&DOORWAY.1 &DOORSLAB.1)

(LOCATION &LOCATION.1
   END1    FAR_RIGHT
   SIDE1   RIGHT
   OBJECT  &DOORWAY.1
   &LOCN-OF <==> (&DOORWAY.1)
   &LOC-FOR <==> (&HINGED.1))

(LOCATION &LOCATION.2
   END1    FAR_RIGHT
   SIDE1   RIGHT
   OBJECT  &DOORSLAB.1
   &LOC-FOR <==> (&HINGED.1)
   &LOCN-OF <==> (&DOORSLAB.1))

(HINGED &HINGED.2
   LOC1        &LOCATION.4
   LOC2        &LOCATION.3
   ITEM        &HINGE.2
   DIR         ALONG-W
   &PROCESS-OF <==> (&DOORWAY.1 &DOORSLAB.1)

(LOCATION &LOCATION.3
   OBJECT  &DOORWAY.1
   SIDE1   FAR_LEFT
   END1    LEFT
   &LOCN-OF <==> (&DOORWAY.1)
   &LOC-FOR <==> (&HINGED.2))

(LOCATION &LOCATION.4
   OBJECT  &DOORSLAB.1
   SIDE1   FAR_LEFT
   END1    LEFT
   &LOCN-OF <==> (&DOORSLAB.1)
   &LOC-FOR <==> (&HINGED.2))

(M-ROTATE &M-ROTATE.1
   NAME      ROTATIONAL
   FUNC-VARS ((LOCATION 10) (DIR 10) (FREEV 40) (GRESTV 40)))
```

Figure 6.23 Doors demo final representation (continued).



School for the Mechanically Declined

○
○
○
○

Chapter 7

Comprehending Mechanical Device Descriptions

EDCA (EDison Conceptual Analyzer) is a computer program that reads mechanical device descriptions such as **Toy Gun**, in English, and constructs FONM representations of the device behavior being described.

> **Toy Gun**
>
> "An object is pushed into a barrel, against a spring, compressing the spring until it catches on a trigger. When the trigger is pulled, the spring is released, and the object is propelled from the barrel."

Three topics are addressed in this chapter, followed by an annotated trace of the **Toy Gun** example:

(1) Natural language comprehension and device descriptions
(2) The EDCA architecture
(3) The EDCA approach to constructing conceptual representations

7.1 Natural Language Comprehension of Device Descriptions

Natural language device comprehension is a process for dynamically constructing device representations from natural language input and incorporating these representations in a current understanding of the description meaning, called an episode or scenario. For example, in **Toy Gun**, EDCA must infer the parts associated with the trigger mechanism despite the fact that they are not stated in the story. To do so, EDCA must access known schemas concerning triggers, guns, toys, barrels, and springs. EDCA must disambiguate the meanings of "into" and "catches," which have different meanings in different contexts. EDCA must resolve anaphoric references such as which object is "it" in the first sentence, since "it" could be the barrel, the object, or the spring. As the analysis continues, EDCA has to dynamically recognize that "until" means that the previous process will be disabled by the next process, and relate the processes accordingly. Finally, EDCA must construct representations for the 'loading-a-toy-gun' and 'firing-a-toy-gun' functions from the description of its underlying behavior, and incorporate those representations, if necessary, in memory for future use.

7.2 EDCA - The EDISON Conceptual Analyzer

EDCA implements five components of the EDISON computational architecture: (1) a demon interpreter, (2) a lexicon and long term semantic memory, (3) a working memory, (4) rules for performing lexical analysis, disambiguation, and reasoning, and (5) an episodic memory, as illustrated in the shaded portion of Fig. 7.1. Words are input to the language demon interpreter (LDI, at 1). A word is matched against entries in the lexicon (SLM, at 2), and candidate conceptual structures are placed onto working memory (WM, at 3). The demons associated with those structures (4) are placed onto an agenda and polled. When a phrase or sentence is fully input, the partially-instantiated structures in working memory are used for conceptual disambiguation. The overall result is integrated in PM (5) and returned as conceptual output.

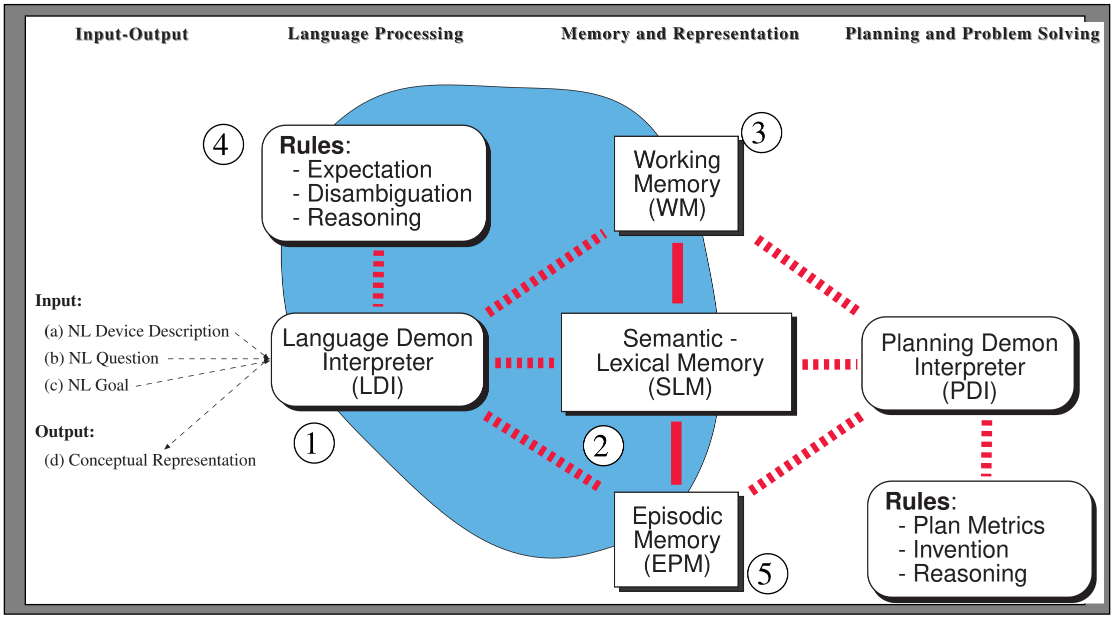


Figure 7.1 The conceptual analyzer components of the EDISON computational architecture. Dotted lines refer to processing control. Solid lines refer to data communication.

- **Semantic and Lexical Memory (SLM)**. Semantic memory represents schematic knowledge of objects, events, and their causal interactions. Lexical memory represents the words associated with conceptual structures in semantic memory. During a parse, input words are compared to their counterparts in lexical memory, and the conceptual components associated with different meaning interpretations are maintained for disambiguation. During conceptual disambiguation, concepts in working memory are associated with semantic and lexical memory, which is used to instantiate incomplete patterns.

- **Language Demon Interpreter (LDI)**. The demon interpreter processes textual input word by word. Each word is potentially associated with many schemas in semantic and lexical memory, collectively called the lexicon. These potential structures compete to determine the word's meaning. During parsing, words are presented and disambiguated as well as possible from the local context using disambiguation rules specific to textual analysis. The results are placed in working memory. When a sentence is entirely parsed, the conceptual structures remaining in working memory are compared to schema in semantic and situation memory to disambiguate the overall meaning using rules for reasoning about mechanical knowledge. When completed, the result is copied to episodic memory.

- **Working Memory (WM)**. During a parse, the conceptual analyzer uses textual input to access conceptual structures from a lexicon in semantic and situation memory. The intermediate conceptual meanings are placed on a list called *working memory*. Working memory serves as a source for disambiguation and reasoning since the lexical items accessed from memory represent the current text. These items also serve as memory retrieval cues. Working memory is maintained until a phrase is disambiguated.

- **Language Processing and Reasoning Rules**. During conceptual analysis, rules are needed to comprehend text. Two types of language processing rules are used, for: (1) word-sense disambiguation, and (2) clausal reference. Rules are associated with lexical entries and conceptual patterns in SLM. The conceptual analyzer loads potential concepts into working memory, and the rules associated with each concept are placed onto an agenda. The demon interpreter polls the rules with respect to all the concepts existing in working memory. When a partially constructed sentence is parsed, reasoning rules are used to identify relationships with conceptual patterns in long term memory, and use those patterns to fill in information missing in the text. Three types of reasoning rules are used: (1) recognition rules, (2) prediction rules, and (2) explanation rules.

- **Episodic Memory (EPM)**. Semantic disambiguation and reasoning begin when the phrasal input is complete. The conceptual analyzer copies the conceptualizations remaining in working memory to a memory list called *episodic process memory*. The concepts in episodic process memory represent EDCA’s mental model of the text. The concepts in episodic process memory are used as retrieval cues to patterns in semantic and situation memory, to fill in existing gaps in causal relationships, and to perform reasoning tasks. When processing is complete, the patterns in episodic process memory are available to incorporate into semantic and situation memory.

7.3 Conceptual Analysis in EDCA

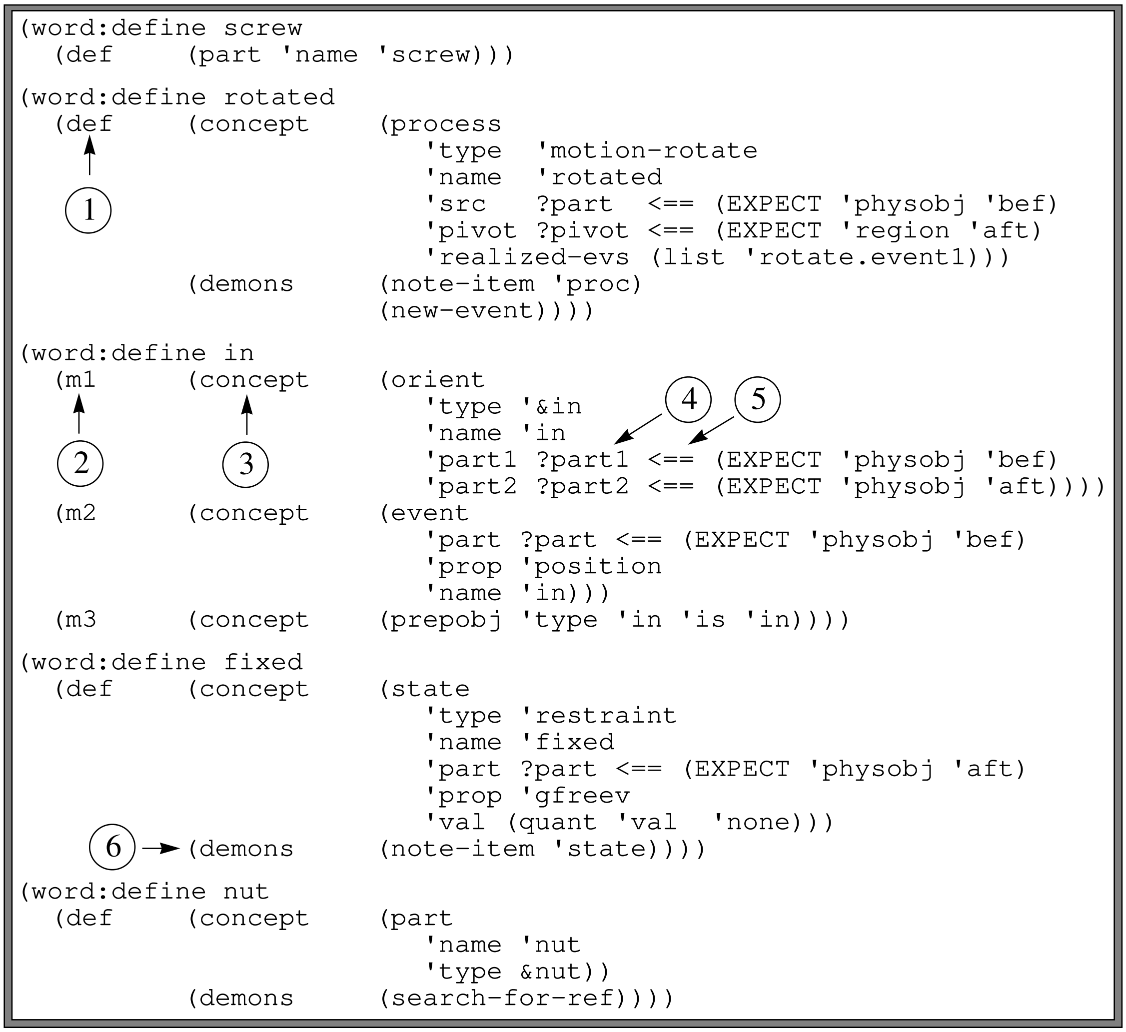
The demon interpreter (LDI) is the primary processing component coordinating text, lexical access, and memory. EDCA is a conceptual analyzer which uses McDypar [Dyer, 1983], a demon-based parser. Demon-based parsing involves the association of procedural entities with lexical entries. These procedural entities, called *demons* [Dyer, 1983], are context sensitive/specific, and are used to aid meaning disambiguation. McDypar processes words one at a time, or in phrases, places a token for the word onto working memory, and spawns any associated demons onto an agenda. The agenda is cycled through for every new word processed, during which all active demons are tested. Demon processing amounts to polling the demon against the currently instantiated working memory (WM) context. For example, if a word is read whose lexical entry has five possible meanings, then there may be five demons, say one for each meaning, placed on the agenda. When one demon's conditions are met, the associated word meaning is instantiated, and the demons associated with other meanings of the word are removed from the agenda. The meaning found is placed onto WM, which is a blackboard that all demons have access to. When an entire section of text is parsed in this way, the current concepts so associated are placed onto episodic process memory (EPM), where the current conceptualization is being constructed. It is in EPM that SLM is used to resolve and explain conflicts, and to make inferences necessary in filling gaps in device understanding.

7.3.5 Lexical Entries

Lexical entries are the link between words and their various conceptual patterns. The device (FONM) representation governs how devices and physical relationships are organized in semantic memory (SLM). In contrast, a lexical entry organizes the general conceptual patterns which a word might take. The pattern serves as a starting point for how the word, if a particular sense is chosen, will be interpreted. The manner in which meanings are chosen is controlled by the demon interpreter. Consider the first phrase in a second example, **Press** (from "How Things Work"; volume 1, page 38.):

> **Press**
>
> "A screw is rotated in a fixed nut, whereby a longitudinal force is developed in the spindle which thrusts against the part under it."

The lexical entries for the first phrase of **Press** are shown in Fig. 7.2. The notational semantics for lexical entries in the figure are defined as follows. The *def* slot (at 1) refers to the definition of the word, which is used when there is a single conceptual entry associated with the word. When there is more than one meaning for a word, the *m1, m2, m3,...,mn* notation (2) is used to define each meaning (e.g., "in" has three meanings, m1 - m3, represented in its lexical entry). Within a particular meaning definition, the *concept* slot (3) refers to the FONM representation associated with the concept. Fillers which are represented as *?bindingtoken* (e.g., ?part, at 4) are pattern variables which are filled during parsing. The <== *(EXPECT...)* notation (5) refers to a gap-filling demon which searches working memory looking for a concept with the class named and in the direction named.



```
(word:define screw
  (def     (part 'name 'screw)))

(word:define rotated
  (def     (concept    (process
                          'type  'motion-rotate
                          'name  'rotated
                          'src   ?part  <== (EXPECT 'physobj 'bef)
                          'pivot ?pivot <== (EXPECT 'region 'aft)
                          'realized-evs (list 'rotate.event1)))
           (demons     (note-item 'proc)
                       (new-event))))

(word:define in
  (m1      (concept    (orient
                          'type '&in
                          'name 'in
                          'part1 ?part1 <== (EXPECT 'physobj 'bef)
                          'part2 ?part2 <== (EXPECT 'physobj 'aft))))
  (m2      (concept    (event
                          'part ?part <== (EXPECT 'physobj 'bef)
                          'prop 'position
                          'name 'in)))
  (m3      (concept    (prepobj 'type 'in 'is 'in))))

(word:define fixed
  (def     (concept    (state
                          'type 'restraint
                          'name 'fixed
                          'part ?part <== (EXPECT 'physobj 'aft)
                          'prop 'gfreev
                          'val (quant 'val  'none)))
           (demons     (note-item 'state))))

(word:define nut
  (def     (concept    (part
                          'name 'nut
                          'type &nut))
           (demons     (search-for-ref))))
```

Figure 7.2 Lexical entries for "a screw is rotated in a fixed nut" within the McDypar parsing model and EDCA lexicon.

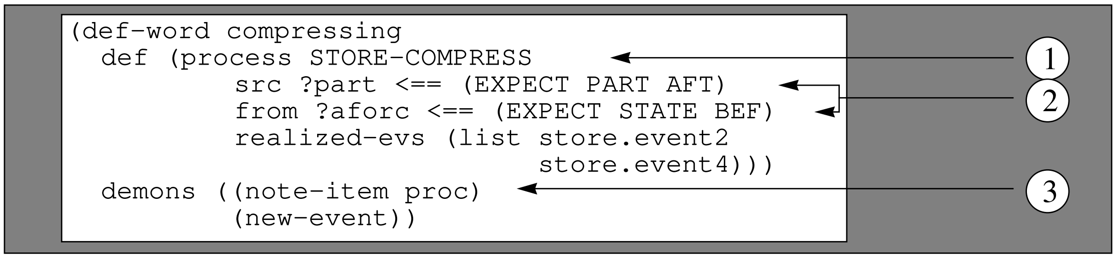
The *demons* slot (6) refers to demons which are associated with the concept-type in general, and are placed onto the parsing agenda when the lexical item is processed.

7.3.6 Textual Analysis and Disambiguation

The primary task of textual analysis is the disambiguation of word-meaning in working memory. Demons are used by EDCA to disambiguate text in four ways, by: (1) generating semantic (case) expectations, (2) resolving object references, (3) determining textually-related causal relations, and (4) structuring and focusing noun group references. The interaction of these demon types organizes enough content into WM so that in-depth reasoning between WM and the knowledge in SLM can occur. The description of **Toy Gun** demonstrates 1-3 of the demon processing types introduced above and is indicative of the difficulties arising in processing physical device descriptions.

Semantic Expectation

Physically-oriented text is often very specific in terms of word meaning. Words such as "barrel," "spring," "compressing," and "trigger" are words which map into specific conceptualizations. However, these words cannot be interpreted correctly without the surrounding context, because they have semantic dependencies which are resolved during processing. For example, "compressing" is represented as a process (BPP-STORE) which cannot itself be fully understood without its *src* and *from* roles being instantiated. Consider the lexical entry for "compressing" in Fig. 7.3:



```
(def-word compressing
  def (process STORE-COMPRESS
          src ?part <== (EXPECT PART AFT)
          from ?aforc <== (EXPECT STATE BEF)
          realized-evs (list store.event2
                             store.event4)))
  demons ((note-item proc)
          (new-event))
```

Figure 7.3 Lexical entry for the word "compressing."

If the word "compressing" is read but the roles are not identified, then the word can only be interpreted as a general reference to what is going on, because there is no direct way (through connections to other objects, which require knowing the regions where the behavior will occur) to relate the object to the associated context. The role fillers for a concept are resolved using a demon which searches for the role type in WM and indexes the two concepts appropriately. In the specific example shown, the EXPECT demon (2, also described below) matches the part and state roles necessary to understand "compressing" in **Toy Gun**:

Expect:

IF a CONCEPT with one of the given CLASSES is found when searching in the given direction

THEN the CONCEPT is bound to the given GAP

The word "compressing" cannot be completely understood in context until the given roles are resolved, even though the expectations for what the process might mean from different perspectives can be spawned. When "compressing" is read, the EXPECT demons for the *src* and *from* roles are spawned. In this case, the *src* role is filled by "spring," and an instance of the STORE process with "spring" instantiating the *src* role is placed onto WM. In addition, when the word "compressing" is parsed into an instance of the process STORE-COMPRESS, a reference to the process is saved (the demon *note-item*, at 3) as a cue for later semantic disambiguation. A *parse diagram*, illustrating the word processed by EDCA and how the expect demon works, is shown as Fig. 7.4.

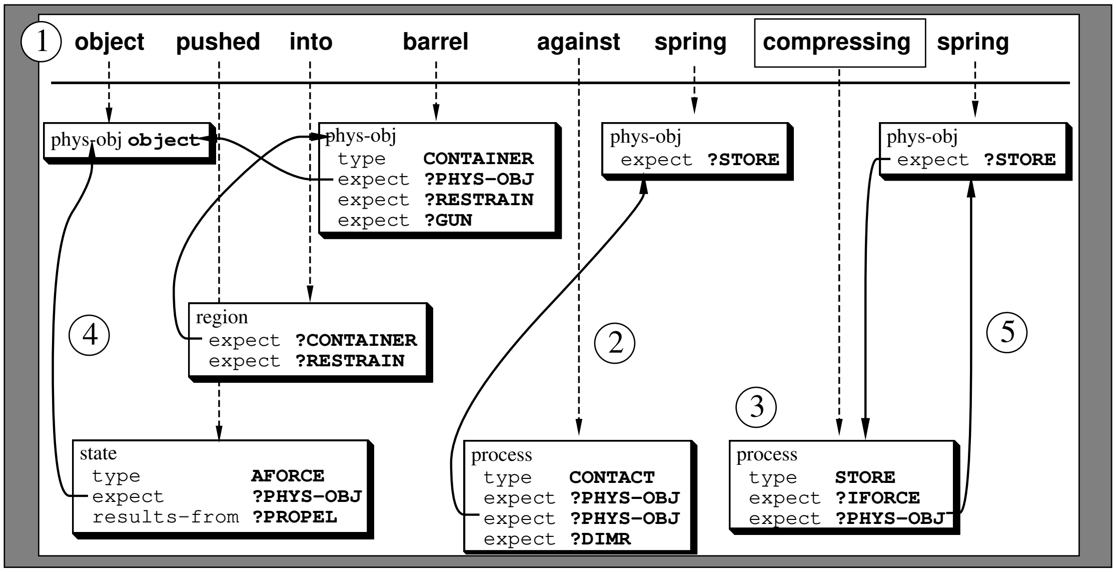


Figure 7.4 Partial parse diagram showing function of the EXPECT demon on "compressing."

Fig. 7.4 shows how the EXPECT demon is used during conceptual analysis to instantiate role fillers and hook conceptualizations together. The figure can be understood as follows. The words which are processed are shown above the solid horizontal line (e.g., "object" at 1). A dotted line is used to depict the correspondence between the word and the associated lexical entry in the EDCA lexicon (e.g., connecting the word "against" and the lexical entry `contact`, at 2). The conceptual representation of a lexical entry, e.g. the STORE process associated with "compressing" (3), shows the EXPECT demons which look into WM for appropriate role fillers. Finally, the solid curved lines represent where the demon eventually finds the filler for the concept and role it is associated with. One example is the line connecting the EXPECT demon for the object role in the AFORCE state with the PHYS-OBJ concept which preceded it (4). In this diagram, all role names, binding variable names, and direction identifiers associated with the EXPECT demon are left out for space considerations. Thus, the second EXPECT demon in the STORE process (5) refers to the demon associated with the *src* role illustrated in Fig. 7.3.

Reference Disambiguation

Object reference is the most common problem encountered in physical descriptions. **Toy Gun** has four object disambiguation instances of two types: (1) pronoun or backward reference, and (2) multiple object reference. The mechanisms used to resolve object references in EDCA are the same; an object is referenced and, depending on the context, it must somehow be remembered until it can be resolved.

1. **Object References:** Semantic expectation is used at the phrasal level to fill conceptual role expectations. However, often roles are not identified locally and must be retrieved from knowledge of previous phrases in EPM. This becomes a reference problem when multiple objects or processes are mentioned prior to the reference in question. In EDCA, this type of reference is associated with physical relations and processes, since they always depend on other processes and components. This problem is exemplified by considering the

second and third phrases of **Toy Gun**:

Toy Gun 2-3

"against a spring, compressing the spring"

The use of "against" and "compressing" in these phrases is indicative of a reference problem. If semantic expectation were to span phrasal boundaries, it would instantiate the first object that satisfied the search criterion. In this case, the *src* role for "against" would be filled by "barrel" while the *src* role for "compressing" would be filled by "object," indicating a breakdown in the use of expectation across phrasal boundaries. The resolution of this conflict has both syntactic and semantic elements.

Syntactic Object Disambiguation

In the first phrase of **Toy Gun 2-3**, the lexical item for the word "against" has an expectation for an object prior to its reference in the text. However, "against" is the first word in the phrase. Therefore, the reference object ("object") must be found from previously read phrases. This is accomplished using a syntactic rule for object reference, illustrated by **Rule OR1** below. Syntactic reference resolution is sometimes required when conceptual knowledge is unavailable or limited, such as in the current example. EDCA maintains a syntactic object focus using reference rules such as **OR1**, although semantic cues check, and can override, syntactic references, when they occur.

Rule OR1:

IF the current process' subject cannot be identified from the current phrase, and if there is a current subject from a previous phrase

THEN identify the current process' subject as the subject of the previous phrase

The application of syntactic rules in disambiguating object references is depicted in the parse diagram shown as Fig. 7.5. In this figure, the semantic expectations have been removed and the lexical items for the commas separating the phrases have been shown (e.g., at 1). When a comma or other lexical object is read, a number of demons are spawned (2) which are used to identify syntactic identifiers (such as subject and object) in the current conceptualization. The dotted line at (3) in Fig. 7.5 shows the effect of these demons on their respective phrases. For example, after the first phrase is parsed, the subject is remembered as "object," and the object is identified as "barrel" (3). When the second and third phrases are parsed, only the phrasal object ("spring") can be identified, so the syntactic rule for temporarily filling the syntactic subject identifies the local subject. Eventually this kind of reference is disambiguated by context, so the syntactic analysis provides placeholders for the role fillers until more information becomes available .

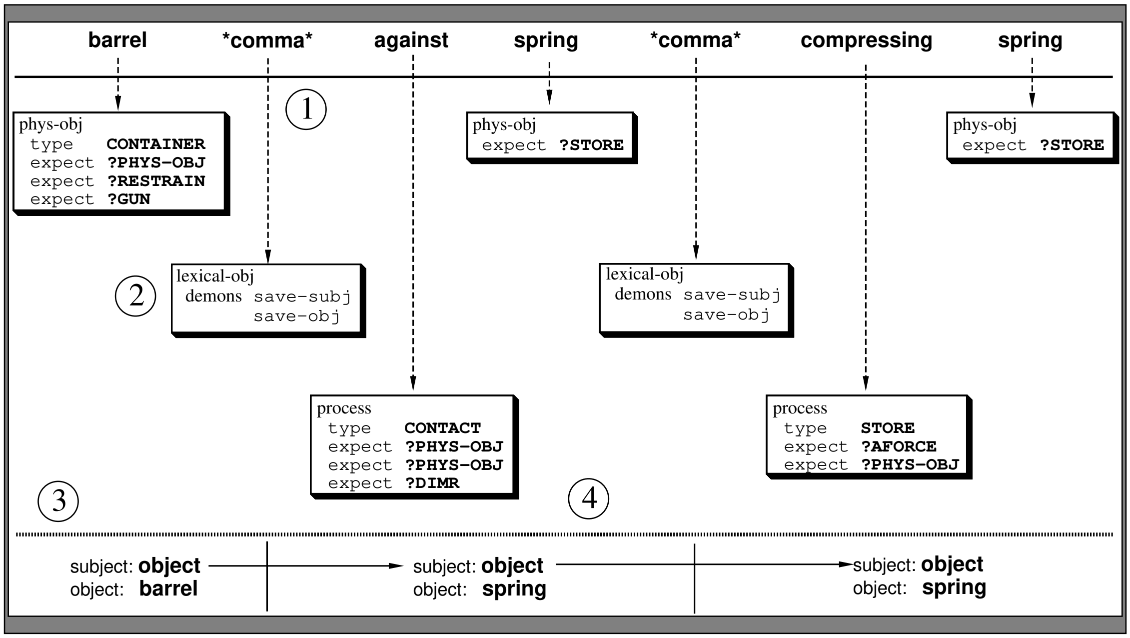


Figure 7.5 Syntactic rules and object reference disambiguation.

Semantic Object Disambiguation

The second phrase of **Toy Gun 2-3** can be understood from the sentence context, even though there is no explicit reference to the phrasal subject, because the physical relationships between the objects and their behavior provide semantic cues for disambiguating the phrasal subject. To understand the causal relationship which gives rise to "spring" compression, the "object" must possess some force that it can transmit to the "spring" in order to be instrumental in compressing the "spring." This knowledge is obtained as a result of knowing the enablement conditions for compression, and knowing what has already occurred in previous phrases. Since the STORE event is already realized, the STORE process enablements (applied force and contact) can be used to find the needed expectation ("object") using **Rule OR2**, thus disambiguating the part reference in "compressing."

Rule OR2:

IF the state resulting from an action (process) is recognized

THEN search memory for the corresponding satisfied enablements to the action (process) and index them into the episode.

This interaction is depicted in the parse diagram shown as Fig. 7.6. In this example, the thick horizontal dotted line distinguishes processing at the phrasal level (i.e., access to WM only) and retrieval from SLM.

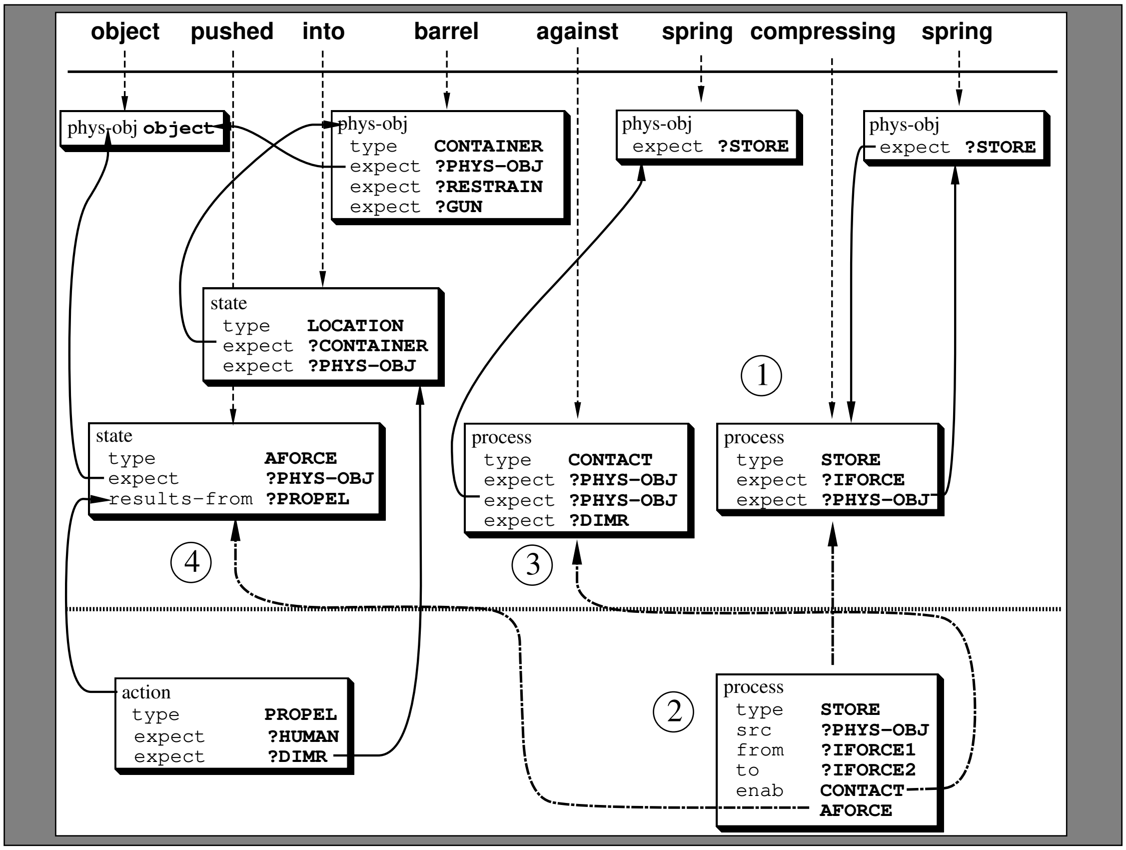


Figure 7.6 Role instantiation as a result of causal chain completion.

The STORE process which represents the lexical entry associated with "compressing" (1) is associated with the more general conceptual STORE schema in SLM (2). When "compressing" is parsed, a comparison to this schema is made possible. The dotted, arrowed lines emanating from the CONTACT and AFORCE process enablements, enable the filling of roles in STORE by finding the associated concepts in working memory (from 3 and 4) and mapping their role instantiations back to the current concept.

Anaphoric Object Disambiguation

The use of explicit pronoun references such as "it" as a reference to objects or processes poses additional problems to disambiguation. This is in part due to the lack of gender cues (such as he, hers, their) which aid in resolving pronouns. The remainder of the problem is that text describing objects and object interactions often references many parts. A reference to "it" may refer to any object or process seen in a discussion of objects and processes. The fourth phrase of **Toy Gun** (shown in **Toy Gun 4**) exemplifies this reference problem:

Toy Gun 4

"until it catches on a trigger"

In this phrase, "it" follows phrases which describe three parts: an "object," a "barrel," and a "spring." The object focus in EDCA combines the syntactic and semantic references as text is processed. However, there is no way to tell whether the part, that the RESTRAIN process associated with "catches" is expecting, is the subject or object of previous phrases. The "barrel" can be taken off the list of potential objects based on the syntactic subject and object bindings, but both the "object" and the "spring" must still be considered. EDCA currently has no mechanism for disambiguating this reference problem, so one of the objects is chosen with the anticipation that, downstream in the parse, semantic cues will help disambiguate the term.

2. **Multiple Object Reference:** Once again referring to **Toy Gun 2-3**, the word "spring" is introduced in the first phrase and reintroduced in the second. The reader has no difficulty understanding that both "spring" words refer to the same part. **Rule OR3** searches backwards for previous mention of objects and re-indexes current references to the original part.

Rule OR3:

IF this concept is mentioned previously in the text

THEN set the meaning of this node to that of the concept found

This methodology works as long as the words all refer to the same object.

Textually-Related Causal Disambiguation

Causal relationships are very prevalent in physical process descriptions. As a result, most every word is used to describe or infer aspects of the causality between parts and processes leading to a natural language conceptualization. One way in which causal ambiguity is resolved is through the use of functional determinants, called *causal keywords*. The first phrase of the second sentence in **Toy Gun** illustrates the use of "when" as a functional determinant:

Toy Gun S2-1

"When the trigger is pulled the spring is released..."

"When" is generally considered a conjunction. In the context of **Toy Gun S2-1,** "when" is a causal keyword implying that "trigger" pulling instrumentally causes "spring" release. Causal relationships are determined during lexical analysis by the use of words such as "when," "by," and "until" which directly imply functional causality. Three causal keyword groups have been identified in the texts analyzed by EDCA: (1) temporal enablement, (2) temporal disablement, and (3) instrumental support. "When" is represented as a temporal enablement conjunction, meaning that the second process (e.g., spring release in the example above) cannot occur until the first process has completed. The use of "until" in the fourth phrase of **Toy Gun** is an example of temporal disablement (see Fig. 7.7), while the use of "by" in the following phrase from **Press** illustrates instrumental support.

Press

"Presses provide a means of compressing and shaping components *BY* exerting high pressure on them."

The state resulting from force transmission ("exerting high pressure") in **Press** is the instrumental support. The causal relationship is recognized from event states of actions or processes. Causal keywords set up an expectation for a antecedent/consequent relationship of which there are two kinds: (1) before, and (2) between. The designation *before* or *between* is based on the relative placement of the keyword with respect to the surrounding events. When a keyword is found before two event states, the causal direction is forward (processA causes processB). If the keyword is *between* the event states the causal direction is backward (processA caused-by processB). For example, in the phrase "The can is opened *after* being punctured." the causal is between the two physical processes and the inference is that "being punctured" precedes "opening can" temporally. In the sentence "*After* the can is punctured it can be opened." the causal is before the two physical processes and the inference is, again, that "punctured" precedes "opening." Although the two sentences chosen have the same meaning, the position and type of causal are a significant aid in interpreting the relationship between the physical events. In the case of process associations, the resulting state of the causing process can then be indexed to an enablement of the other.

In the **Toy Gun** example shown in Fig. 7.7, the causal disablement is between the two process events. This means that the causal direction is backward: the latter (catching) event is recognized as the causal antecedent, while the former (compressing) event is recognized as the causal consequent. Thus a conceptualization can be built indexing the two events.

The process of "catching" disables "spring compression" (Fig. 7.7).

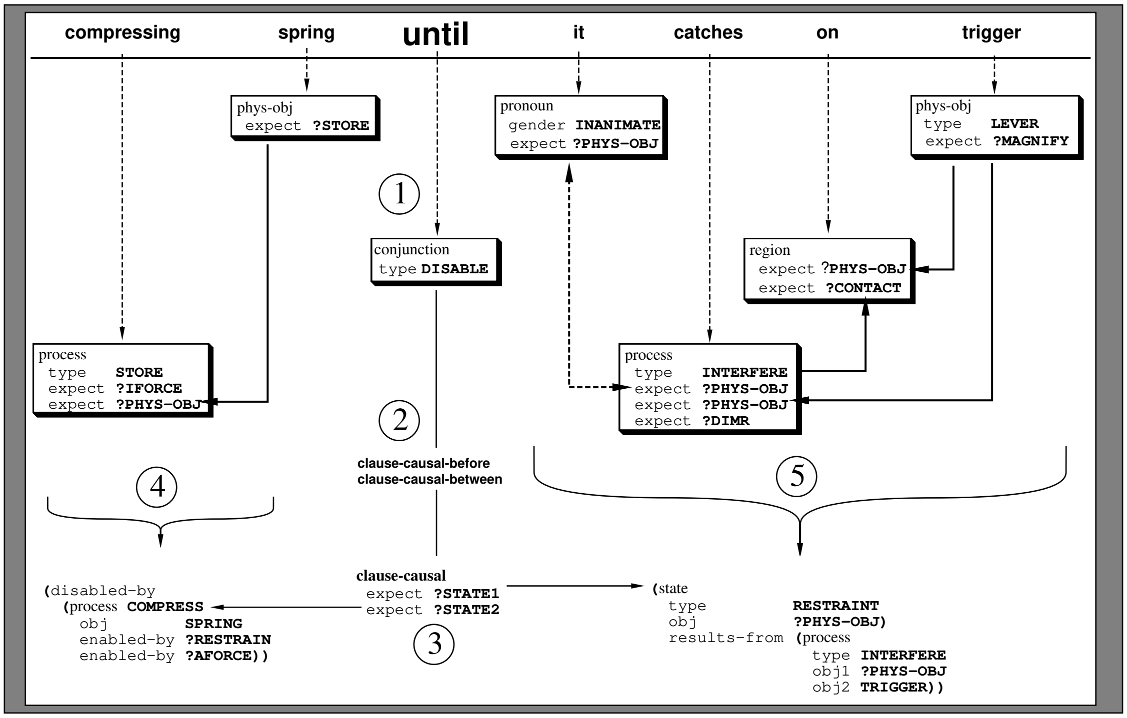


Figure 7.7 Causal enablement in **Toy Gun.**

Fig. 7.7 represents an example of textually-related causal disambiguation. In "compressing the spring until it catches on the trigger," the causal keyword is the conjunction "until," the processing of which creates expectations for the types of causal relationship between phrases previously described (before and between, enablement and disablement, at 1 and 2). As the text is analyzed, the phrasal events become known, and the causal expectations (antecedent and consequent, 3) are realized for the *between* case, resulting in a causal conceptualization (processB disables processA). The first phrase (processA) is recognized as the event resulting from the "compressing the spring" process. The second phrase (processB) is recognized as the event resulting from the "catches on the trigger" process. Some of the lexical entries and associated expectations have been left out of the figure.

WM Refinement via Expectation Failures

Instantiating a device in EPM requires an analysis of what is known about the device in question with respect to what is known about devices in general. Textual descriptions can describe devices which are slightly different, or broken, with respect to known devices in memory. Comparison of the known device to device schema in SLM helps to define the differences, which aids in their interpretation. The analysis associated with these comparisons leads to the reintegration of a new device into memory, and to rules describing device faults, both resulting in explanations.

Constructing Episodic Device Memory

The end product of reading a device description is a completely instantiated device episode. The episode reflects understanding gained through reading the description, comparing it to known knowledge sources and reasoning about the two, by making connections between the related knowledge structures. In order to do this two things must happen: (1) the objects and processes must be indexed in the new episode, and (2) device functions must be constructed. Constructing device functions and reintegrating the device into SLM are learning tasks which, along with the actual organization of memory used in EDCA, have not been fully addressed in this dissertation.

7.4 EDCA Implementation

The basis for EDCA is the McDypar demon-based parser, which was used extensively at the UCLA Artificial Intelligence lab until 1988. The first version of EDCA was written in T [Reese, 1984], a scheme-based Lisp dialect developed at Yale. The model was implemented on top of an AI development tool called RHAPSODY [Turner and Reeves, 1987] written at UCLA. RHAPSODY provided representational semantics, the McDypar parser, a generator, demon and agenda packages, and graphics facilities. The EDCA model uses approximately 150 demons to analyze the **Toy Gun** and **Press** device descriptions illustrated in this chapter as well as a few others.

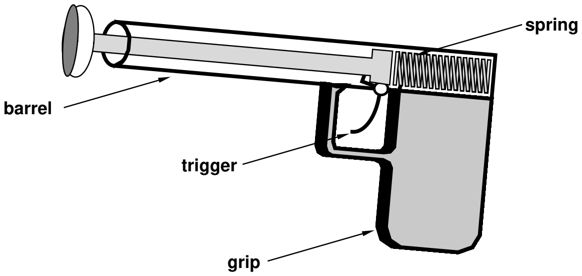
7.4.1 A Detailed Example: Toy Gun

The **Toy Gun** example has been used extensively to demonstrate various issues in device description comprehension and how they are addressed in EDCA from a conceptual standpoint. The processing of **Toy Gun** is described here in greater detail. The first sentence in Fig. 7.8a represents the *arming* function for a ’typical’ toy gun, such as the one depicted in Fig. 7.8b. This example has been selected to illustrate the issues encountered while processing and understanding physical device descriptions. EDCA has demons used in lexical analysis as well as for reasoning about partially instantiated conceptual structures. The processing of these somewhat different demon types is accomplished with two agendas, one for the textually-specific demons (agenda.4 in the trace), and one for the reasoning and domain-specific demons (agenda.7 in the trace). Agenda.4 is run after every word is processed, while agenda.7 is run at clause boundaries. A large amount of processing is done in just the first sentence of this example. As a result, a number of demons have been left out of the trace, as well as redundant processing (throughout) once it has been reviewed once.

After EDCA is loaded into memory the following text is processed:

(AN OBJECT IS PUSHED INTO A BARREL *COMMA* AGAINST A SPRING *COMMA* COMPRESSING THE SPRING UNTIL IT CATCHES ON A TRIGGER *PERIOD*)

(a)



(b)

Figure 7.8 (a) EDCA input for first sentence of **Toy Gun** text, (b) toy gun illustration.

```
---------------------------------
    Processing word: AN
---------------------------------

   Spawning demon: (IGNOR.1 #{WMN.1})
RUNNING AGENDA: #{AGENDA:PARSE-AGENDA.4}
RUNNING DEMON: IGNOR.1
   Executing +act: (IGNOR.1)
   Killing demon: IGNOR.1
```

Articles are ignored by the conceptual analyzer

```
---------------------------------
    Processing word: OBJECT
---------------------------------

Adding to *wm*: #{WMN.2}
Created concept:

(PHYSOBJ &PHYSOBJ.19  ◄────────────────────────────────────  OBJECT
 NAME OBJECT)

---------------------------------
    Processing word: IS
---------------------------------

   Spawning demon: (FIND-EVENT.1 #{WMN.3})
```

```
   FIND-EVENT.1
   =========================================================
   TEST:   If the OBJECT or PREPOSITIONAL PHRASE (for one
           of the given preps) can be interpreted as an
           EVENT in a higher level memory structure event
   ACT:    Let that event be the event for the sentence
   =========================================================

   Spawning demon: (TEST-OBJECT.1 #{WMN.3})

   TEST-OBJECT.1
   =========================================================
   TEST:   Test the OBJECT of a verb to see if it is one
           of the given CLASSES
   ACT:    Set the meaning for the node to the given
           meaning and execute the given procedure
   =========================================================
```

*The two meanings of ``is'' represented are as a reference to an action or process (and particularly to states of same), and as a object reference. In this text the ``is'' refers to the action (**PROPEL**), and the resulting containment of the ``object'' in the ``barrel'' which is caused by the ``push'' of the ``object''.*

```
---------------------------------
    Processing word: PUSHED
---------------------------------

Adding to *wm*: #{WMN.4}
Created concept:

(ACTION &ACTION.1
 TYPE &ACTION.13   ◄────────────────────────────────────   PROPEL
 NAME PUSHED
 OBJ ?PHYSOBJ)

   Spawning demon: (NEW-EVENT.1 #{ACTION.1})

   NEW-EVENT.1
   ============================================================
   TEST:   A new event or state is parsed
   ACT:    Run the rule-agenda to see what domain knowledge
           has to say.
   ============================================================

   Spawning demon: (EXPECT.1 #{WMN.4} #{ACTION.1} (OBJ) PHYSOBJ BEF)

   EXPECT.1
   =======================================================
   TEST:   A CONCEPT with one of the given CLASSES is
           found when searching in the given direction
   ACT:    The CONCEPT is bound to the given GAP
   =======================================================
```

```
RUNNING AGENDA: #{AGENDA:PARSE-AGENDA.4}

RUNNING DEMON: EXPECT.1
   Executing +act: (EXPECT.1)
  Slot "(PART)" in #{ACTION.1} <-- #{PHYSOBJ.19}
   Killing demon: EXPECT.1

RUNNING DEMON: NEW-EVENT.1
   Executing +act: (NEW-EVENT.1)

Adding to *wm*: #{WMN.5}
Created concept:
```

***PUSHED** is assumed to be an object state in which **PROPEL** action has occurred and the associated goals for **PROPEL** can be inferred. **NEW-EVENT** integrates domain-specific (DS) knowledge with the lexical analysis by spawning DS demons which are later analyzed. **EXPECT.1** locates the object of the **PROPEL** in **WM**.*

```
---------------------------------
    Processing word: INTO
---------------------------------

Adding to *wm*: #{WMN.6}
Created concept:

(PROCESS CONTACT.3
 TYPE         RESTRAIN
 SRC          ?PHYSOBJ1
 DST          ?PHYSOBJ2
 DIMR         ?DIM
 REALIZED-EVS (&EVENT.1 &EVENT.2))

   Spawning demon: (EXPECT.2 #{WMN.6} #{CONTACT.3} (SRC) PHYSOBJ BEF)

   EXPECT.2
   =======================================================
   TEST:   A CONCEPT with one of the given CLASSES is
           found when searching in the given direction
   ACT:    The CONCEPT is bound to the given GAP
   =======================================================

   Spawning demon: (EXPECT.3 #{WMN.6} #{CONTACT.3} (DST) PHYSOBJ AFT)

   EXPECT.3
   =======================================================
   TEST:   A CONCEPT with one of the given CLASSES is
           found when searching in the given direction
   ACT:    The CONCEPT is bound to the given GAP
   =======================================================

   Spawning demon: (EXPECT.4 #{WMN.6} #{CONTACT.3} (DIMR) DIM AFT)

   EXPECT.4
```

```
   =======================================================
   TEST:   A CONCEPT with one of the given CLASSES is
           found when searching in the given direction
   ACT:    The CONCEPT is bound to the given GAP
   =======================================================

RUNNING DEMON: EXPECT.2
   Executing +act: (EXPECT.2)
  Slot "(SRC)" in #{CONTACT.3} <-- #{PHYSOBJ.19}
   Killing demon: EXPECT.2

RUNNING DEMON: FIND-EVENT.1
   Executing +act: (FIND-EVENT.1)
   Killing demon: FIND-EVENT.1
```

*One meaning of **INTO** is containment, which is a type of connection for object groupings. Expectations are spawned for the associated parts as well as the dimension of connection. Since this process is within the scope of ``is'' the restraint and orientation events are taken as realized by **FIND-EVENT.1***

```
---------------------------------
    Processing word: BARREL
---------------------------------

Adding to *wm*: #{WMN.7}
Created concept:

(PHYSOBJ &PHYSOBJ.20
 TYPE     &PHYSOBJ.21     ◄──────────────────────────────  TUBE
 NAME     BARREL
 PHYS-ATS &PHYS-ATS.5)
RUNNING DEMON: EXPECT.3
   Executing +act: (EXPECT.3)
  Slot "(DST)" in #{CONTACT.3} <-- #{PHYSOBJ.20}
   Killing demon: EXPECT.3
```

*An instance of &tube is placed onto **WM**. **BARREL** has a functional region HOLE which is oriented along the barrel length. A DS demon, **PHYSOBJ:BARREL-DIM**, for regions matches the need for a dimension with the dimensional characteristic of the **BARREL** hole.*

```
---------------------------------
    Processing word: *COMMA*
---------------------------------

Adding to *wm*: #{WMN.8}
Created concept:

(LEXICAL-ITEM &LEXICAL-ITEM.7
 TYPE COMMA)

   Spawning demon: (SAVE-SO.1 #{WMN.8})

   SAVE-SO.1
```

```
   ======================================================
   TEST:   There is no process for a clause.
   ACT:    The earliest object is the subject, and
           the most recent object is the object.
   ======================================================

   Spawning demon: (SAVE-EV.1 #{WMN.8})

   SAVE-EV.1
   ======================================================
   TEST:   There is a local process for the clause,
           and the clause is not a relative clause.
   ACT:    The process is the local process.  Otherwise
           we forward the existing local process.
   ======================================================

   Spawning demon: (PROCESS-CLAUSE.1 #{WMN.8})

   PROCESS-CLAUSE.1
   ======================================================
   TEST:   When a SENTENCE or CLAUSE BOUNDRY has been
           reached...
   ACT:    Fire demons to EXPLAIN the concepts that
           have been recognized
   ======================================================
RUNNING DEMON: SAVE-SUBJ1.1
   Executing +act: (SAVE-SUBJ1.1)
[Binding *MOST-RECENT-SUB*]
   Killing demon: SAVE-SUBJ1.1

RUNNING DEMON: SAVE-EV.1
   Executing +act: (SAVE-EV.1)
[Binding *MOST-RECENT-PROC*]
   Killing demon: SAVE-EV.1

RUNNING DEMON: PROCESS-CLAUSE.1
   Executing +act: (PROCESS-CLAUSE.1)
```

*The demons **SAVE-SO**, **SAVE-SUB1**, **SAVE-SUB2**, **SAVE-SUB3**, **SAVE-OBJ1**, **SAVE-OBJ2**, **SAVE-OBJ3** (not shown), and **SAVE-EV** are reference placeholders for objects in different contexts. In this case **SAVE-SO**, **SAVE-SUBJ.1**, and **SAVE-EVENT** retain ordinal knowledge for future reference problems.*

*When a motion of an unspecified device is recognized a demon, **EP:PROC**, is spawned to create a new device instance. Later the validity of this assumption is checked. **OBJECT**, **BARREL** and the forced-state of **OBJECT** are placed onto EPM for future device reference. Later these constructs are incorporated into the new device representation.*

```
Adding to *ep*: #{EPN.1}
Put concept in EPISODIC MEMORY:
```

```
(PROCESS &PROCESS.3
 REALIZED-EVS (&$MOTION.EVENT1)
 SRC          &PHYSOBJ.19)   ◄──────────────────────────  OBJECT

   Spawning demon: (EP:PROC.1 #{PROCESS.3})

   EP:PROC.1
   ======================================================
   TEST:   A mechanical process is parsed.
   ACT:    Create a new d-mop in episodic memory.
   ======================================================

Adding to *ep*: #{EPN.2}
Put concept in EPISODIC MEMORY:

(STATE &STATE.12
 TYPE &STATE.13
 NAME PUSHED
 PART &PHYSOBJ.19   ◄───────────────────────────────────  OBJECT
 PROP FORCE
 VAL  &QUANT.13)

   Spawning demon: (PART:BARREL-DIM.1 #{PHYSOBJ.20})

   PART:BARREL-DIM.1
   ======================================================
   TEST:   A BARREL part is recognized.
   ACT:    See if HOLE attributes can be used in memory.
   ======================================================

   Spawning demon: (PART:SCRIPT-REC.1 #{PHYSOBJ.20})

   PART:SCRIPT-REC.1
   ======================================================
   TEST:   A part is placed into episodic memory.
   ACT:    Check partially instantiated scripts/mops
           for activation.
   ======================================================

   Spawning demon: (EP:PART.1 #{PHYSOBJ.20})

   EP:PART.1
   ======================================================
   TEST:   A part is placed into episodic memory.
   ACT:    Make it a part of the current device.
   ======================================================

   Spawning demon: (PART:LINKAGE-RECOG.1 #{PHYSOBJ.19})

   PART:LINKAGE-RECOG.1
   ======================================================
        IF the part has rod or slab ptype
        THEN the part is a linkage
   ======================================================
```

```
RUNNING AGENDA: #{AGENDA:RULE-AGENDA.7}

RUNNING DEMON: PART:BARREL-DIM.1
   Executing +act: (PART:BARREL-DIM.1)
Adding to *ep*: #{EPN.3}
Put concept in EPISODIC MEMORY:

RUNNING DEMON: EP:PROC.1
   Executing +act: (EP:PROC.1)
Adding to *ep*: #{EPN.3}
Put concept in EPISODIC MEMORY:

 (M-DEVICE &NULL-DEVICE
  DEVICE &DEVICE.1
  ACTOR  EDISON)

   Killing demon: EP:PROC.1
```

When a sentence or clause end is reached, DS demons are also fired to incorporate recognized knowledge into the current episodic representation for the unknown device. In this first phrase DS demons ***EP:PART.1****,* ***EP:PROC****,* ***PART:BARREL-DIM.1****, and* ***PART:LINKAGE-RECOG.1*** *are spawned (among others) to interpret what* ***OBJECT*** *is, and to aid in indexing the dimension from* ***BARREL*** *with* ***$MOTION.1****.*

PART:SCRIPT-REC.1 *is an example of one of many DS demons which are spawned onto the* ***RULE:AGENDA*** *when parts are placed onto* ***WM****. This demon looks for the processes and devices in which the part is a role filler and checks to see whether they are instantiated enough in* ***WM*** *to activate.* ***EP:PROC.1*** *is a demon which looks for new mentions of devices* *and creates dummy's for interpretive purposes.*

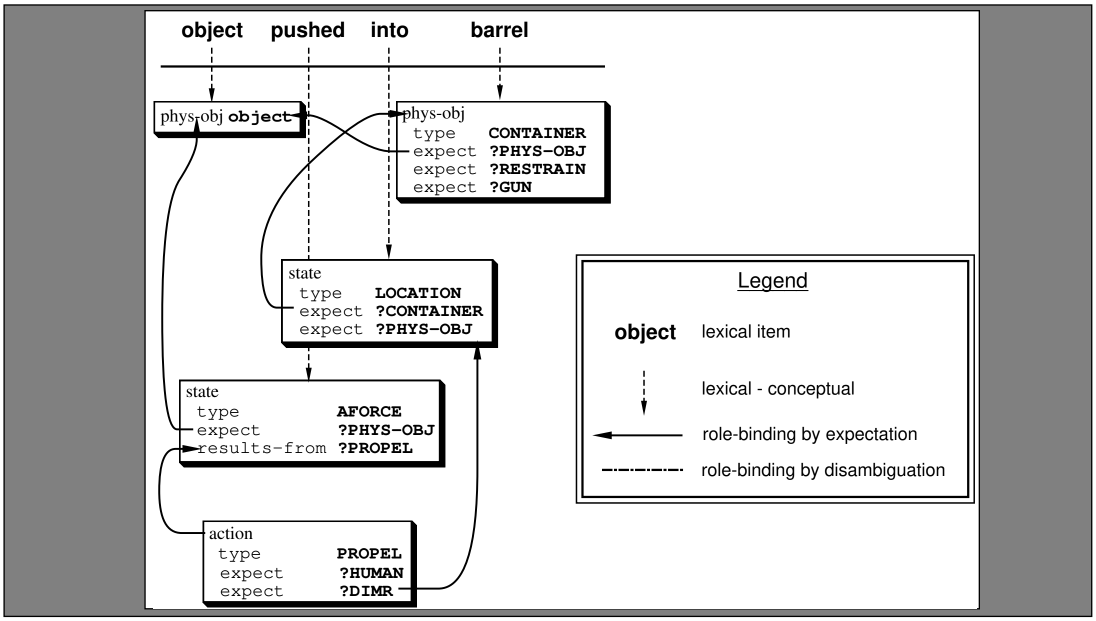


Figure 7.9 Parse diagram for the first phrase in **Toy Gun**, sentence 1.

```
---------------------------------
    Processing word: AGAINST
---------------------------------

Adding to *wm*: #{WMN.9}
Created concept:

(PROCESS &PROCESS.4
 NAME         AGAINST
 SRC          ?PHYSOBJ1
 DST          ?PHYSOBJ2
 REALIZED-EVS (&EVENT.5))

   Spawning demon: (EXPECT.2 #{WMN.9} #{PROCESS.4} (DST) PHYSOBJ AFT)

   EXPECT.2
   =======================================================
   TEST:   A CONCEPT with one of the given CLASSES is
           found when searching in the given direction
   ACT:    The CONCEPT is bound to the given GAP
   =======================================================

   Spawning demon: (EXPECT.3 #{WMN.9} #{PROCESS.4} (SRC) PHYSOBJ BEF)

   EXPECT.3
   =======================================================
   TEST:   A CONCEPT with one of the given CLASSES is
           found when searching in the given direction
   ACT:    The CONCEPT is bound to the given GAP
```

```
   =======================================================
```

*The **EXPECT** demons are spawned to locate the roles in the connection instance. The connection **AGAINST** satisfies the restraint expectation from the first clause. The realized event in **AGAINST** helps to satisfy the orientation between **OBJECT** and **BARREL**, as well as to fill an inference that the spring is also in the **BARREL** (orientation is transitive, i.e. if A is in C, and A is connected to B, then B is in C).*

```
---------------------------------
    Processing word: SPRING
---------------------------------

Adding to *wm*: #{WMN.11}
Created concept:

(PHYSOBJ &PHYSOBJ.22
 TYPE &SPRING
 NAME SPRING)

RUNNING DEMON: EXPECT.2
   Executing +act: (EXPECT.2)
  Slot "(DST)" in #{PROCESS.4} <-- #{PHYSOBJ.22}
   Killing demon: EXPECT.2
---------------------------------
    Processing word: *COMMA*
---------------------------------

   Spawning demon: (SAVE-SUBJ3.2 #{WMN.12})

   SAVE-SUBJ3.2
   ==================================================================
   TEST:   A subject of a process doesn't exist for the current phrase,
           and there is a preceding phrase which has a subject.
   ACT:    Remember the subject from the previous phrase as the most
           recent subject.
   ==================================================================

RUNNING AGENDA: #{AGENDA:PARSE-AGENDA.4}

RUNNING DEMON: SAVE-SUBJ3.2
   Executing +act: (SAVE-SUBJ3.2)
   Killing demon: SAVE-SUBJ3.2

RUNNING DEMON: SAVE-OBJ1.2
   Executing +act: (SAVE-OBJ1.2)
   Killing demon: SAVE-OBJ1.2

Adding to *ep*: #{EPN.4}
Put concept in EPISODIC MEMORY:
```

```
(PROCESS &PROCESS.4
 NAME         AGAINST
 SRC          &PHYSOBJ.19    ◄──────────────────────────── OBJECT
 DST          &PHYSOBJ.22    ◄──────────────────────────── SPRING
 REALIZED-EVS (&EVENT.5))
```

*In this phrase the use of multiple reference resolution techniques shows itself. In this case a process and local referents exist from the previous clause, as well as not existing (local referents) in the present clause. The result is that the previous subject is brought forward into the present one. However, since **AGAINST** was expected from the previous clause and instantiated in the present one, the subject of the previous clause is brought forward via context anyway. To reiterate a prior point, we have intentionally left out a number of processing items for the sake of brevity.*

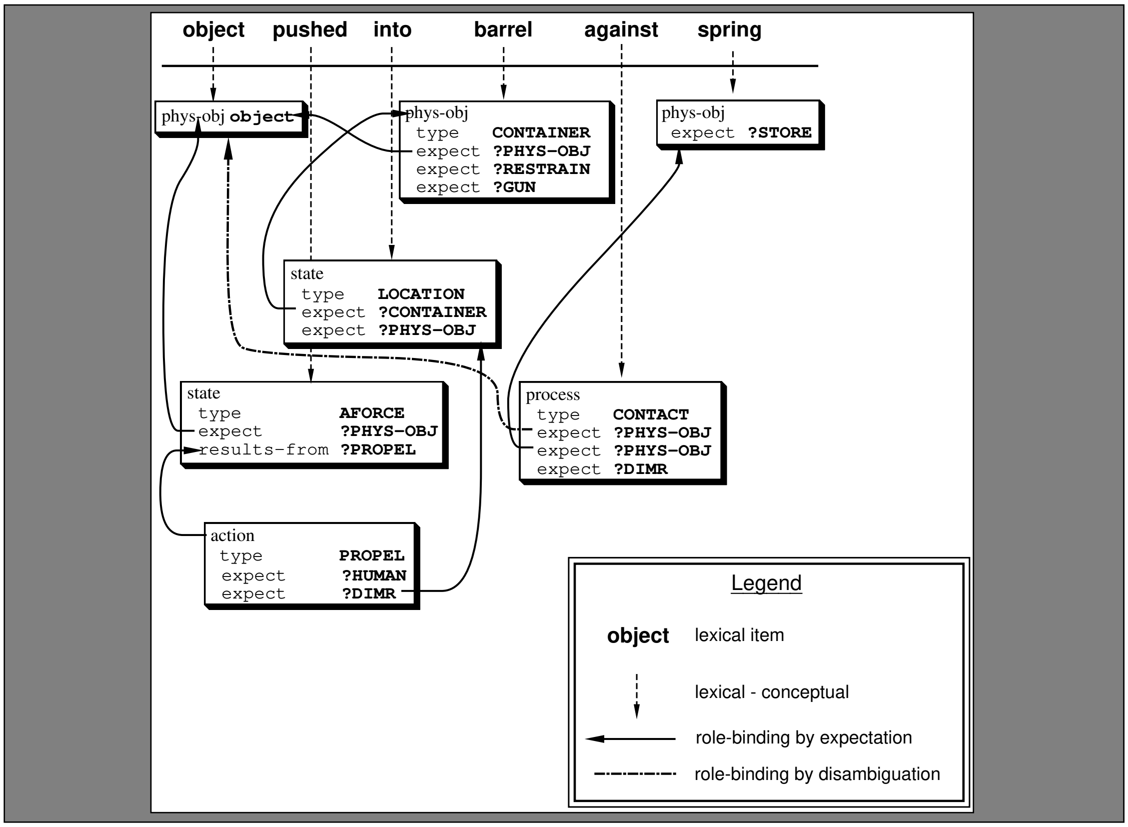


Figure 7.10 Parse diagram for the first two phrases of **Toy Gun**, sentence 1.

```
---------------------------------
    Processing word: COMPRESSING
---------------------------------

Adding to *wm*: #{WMN.13}
Created concept:

(PROCESS &PROCESS.5
 NAME         COMPRESSING
 SRC         ?PHYSOBJ
 REALIZED-EVS (&EVENT.6 &EVENT.7))

   Spawning demon: (EXPECT.5 #{WMN.13} #{PROCESS.5} (SRC) PHYSOBJ AFT)

   EXPECT.5
   =======================================================
   TEST:   A CONCEPT with one of the given CLASSES is
           found when searching in the given direction
   ACT:    The CONCEPT is bound to the given GAP
   =======================================================

---------------------------------
    Processing word: SPRING
---------------------------------

Adding to *wm*: #{WMN.15}
Created concept:

(PHYSOBJ &PHYSOBJ.23
 TYPE &SPRING
 NAME SPRING)

   Spawning demon: (SEARCH-FOR-REF.4 #{WMN.15})

   SEARCH-FOR-REF.4
   =======================================================
   TEST:   Search for a referent for this concept
   ACT:    Set the meaning for the node to the
           concept found
   =======================================================

RUNNING DEMON: EXPECT.5
   Executing +act: (EXPECT.5)
  Slot "(SRC)" in #{PROCESS.5} <-- #{PHYSOBJ.23}
   Killing demon: EXPECT.5

RUNNING DEMON: SEARCH-FOR-REF.4
   Executing +act: (SEARCH-FOR-REF.4)
   Spawning demon: (REPLACE-TO-BNDRY.1 #{WMN.15} #{PHYSOBJ.22})

   REPLACE-TO-BNDRY.1
   ================================================================
   TEST:   A concept is a forward reference to a previous concept
   ACT:    Replace all concepts which referred to the existing
```

```
           concept prior to its redesignation
   ================================================================

RUNNING DEMON: REPLACE-TO-BNDRY.1
   Executing +act: (REPLACE-TO-BNDRY.1)
   Killing demon: REPLACE-TO-BNDRY.1
```

*When the word **COMPRESSING** is read, a **STORE** process is instantiated when it finds the object **SPRING**. At the same time, **SPRING** has been mentioned before. **SEARCH-FOR-REF** is spawned to find and assimilate previous instances of spring (or any part) in ***WM***. Having found the previous instance the demon reindexes ***WM*** so that the second instance doesn't point to anything. This part is later removed altogether.*

*Likewise there is also a reference problem in this clause. The **STORE** process has no subject but cannot be initiated without some kind of contact and transmission. Because the **STORE** event is recognized, the enablement conditions must be met. A DS demon, **EP:-PROC-PRECOND2**, is needed to survey the enablements to **STORE** and confirm. Since the **OBJECT** possesses force, is in motion, and is in contact with the **SPRING**, all of the **STORE** enablements are met. As a result **OBJECT** must be the cause of the compression.*

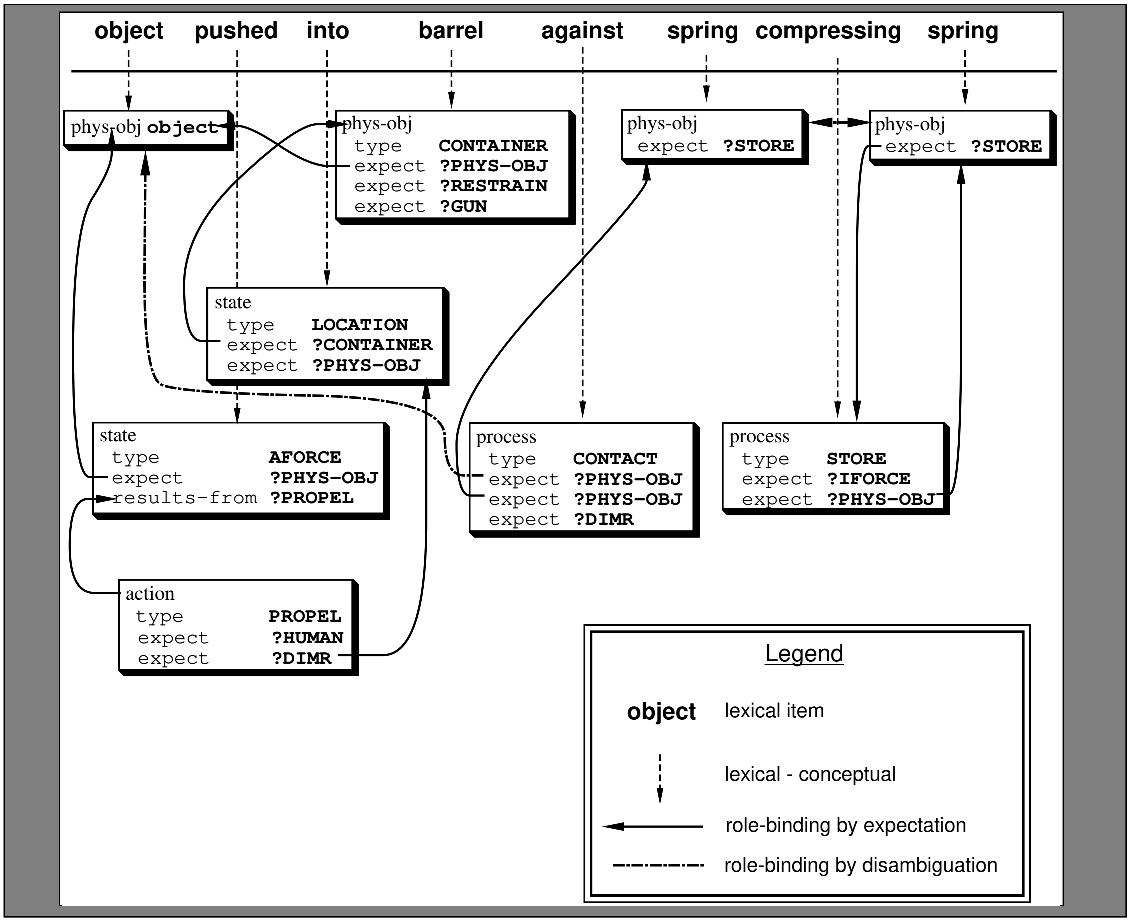


Figure 7.11 Parse diagram for the first three phrases of **Toy Gun**, sentence 1.

```
---------------------------------
    Processing word: UNTIL
---------------------------------

Adding to *wm*: #{WMN.16}
Created concept:

(PREPOBJ &PREPOBJ.8
 TYPE CAUSAL
 IS   UNTIL)

RUNNING DEMON: PROCESS-CLAUSE.3
   Executing +act: (PROCESS-CLAUSE.3)

Adding to *ep*: #{EPN.5}
Put concept in EPISODIC MEMORY:

(PROCESS &PROCESS.5
 NAME         COMPRESSING
 SRC          &PHYSOBJ.22   ◄────────────────────── SPRING (original)
 REALIZED-EVS (&EVENT.6 &EVENT.7))

   Spawning demon: (CLAUSE:CAUSAL2-BET.1 #{WMN.16})

   CLAUSE:CAUSAL2-BEF.1
   ==============================================================
   TEST:   A causal disablement keyword is found while parsing.
   ACT:    Search for an event preceding, an event
           following, and spawn demons to infer the
           relationship between the respective clauses.
   ==============================================================
```

UNTIL *signals the potential for an* ***antecedent/consequent*** *relationship. Because these usually occur between high level knowledge structures they are preferred over process interpretation (and get some preference as a result).* ***CLAUSE:CAUSAL-BET*** *and* ***CLAUSE:CAUSAL-BEF*** *(not shown) are spawned from the functional-keyword meaning of* ***UNTIL****.*

```
(LEXICAL-ITEM &LEXICAL-ITEM.9
---------------------------------
    Processing word: IT
---------------------------------

Adding to *wm*: #{WMN.17}
Created concept:

(LEXICAL-ITEM &LEXICAL-ITEM.9
 TYPE PRON
 IS   IT)

   Spawning demon: (FIND-REF.1 #{WMN.17})
```

```
   FIND-REF.1
   ====================================================
   TEST:   OUR-NODE occurs late in a clause.
   ACT:    Set OUR-NODE's concept to the most
           recently mentioned object, otherwise to
           to the most recently mentioned subject.
   ====================================================

---------------------------------
    Processing word: CATCHES
---------------------------------

Adding to *wm*: #{WMN.18}
Created concept:

(PROCESS &PROCESS.5
 NAME         CATCHES
 SRC          ?PHYSOBJ1
 DST          ?PHYSOBJ2
 REALIZED-EVS (&EVENT.8))

RUNNING DEMON: EXPECT.7
   Executing +act: (EXPECT.7)
  Slot "(SRC)" in #{PROCESS.5} <-- #{PHYSOBJ.19}
   Killing demon: EXPECT.7

---------------------------------
    Processing word: ON
---------------------------------

Adding to *wm*: #{WMN.19}
Created concept:

(ORIENT &ORIENT.5
 NAME  ON
 TYPE  ()
 PART1 ?PHYSOBJ1
 PART2 ?PHYSOBJ2)

   Spawning demon: (EXPECT.8 #{WMN.19} #{ORIENT.5} (PART2) PHYSOBJ AFT)

   EXPECT.8
   =======================================================
   TEST:   A CONCEPT with one of the given CLASSES is
           found when searching in the given direction
   ACT:    The CONCEPT is bound to the given GAP
   =======================================================

   Spawning demon: (EXPECT.9 #{WMN.19} #{ORIENT.5} (PART1) PHYSOBJ BEF)

   EXPECT.9
   =======================================================
   TEST:   A CONCEPT with one of the given CLASSES is
           found when searching in the given direction
```

```
   ACT:    The CONCEPT is bound to the given GAP
   =======================================================

RUNNING DEMON: EXPECT.9
   Executing +act: (EXPECT.9)
  Slot "(PART1)" in #{ORIENT.5} <-- #{PHYSOBJ.19}
   Killing demon: EXPECT.9

---------------------------------
    Processing word: TRIGGER
---------------------------------

Adding to *wm*: #{WMN.21}
Created concept:
(PART &PHYSOBJ.24
 TYPE &PHYSOBJ.25    ◄───────────────────────  simple lever mechanism
 NAME TRIGGER)

RUNNING DEMON: EXPECT.8
   Executing +act: (EXPECT.8)
  Slot "(PART2)" in #{ORIENT.5} <-- #{PHYSOBJ.24}
   Killing demon: EXPECT.8

RUNNING DEMON: EXPECT.6
   Executing +act: (EXPECT.6)
  Slot "(DST)" in #{PROCESS.5} <-- #{PHYSOBJ.24}
   Killing demon: EXPECT.6

---------------------------------
Processing word: *PERIOD*
---------------------------------

Adding to *wm*: #{WMN.22}
Created concept:

(LEXICAL-ITEM &LEXICAL-ITEM.10
 TYPE PERIOD)

RUNNING DEMON: PROCESS-CLAUSE.4
   Executing +act: (PROCESS-CLAUSE.4)

Adding to *ep*: #{EPN.6}
Put concept in EPISODIC MEMORY:

(ORIENT &ORIENT.5
 NAME  ON
 TYPE  ()
 PART1 &PHYSOBJ.19    ◄───────────────────────────────────  OBJECT
 PART2 &PHYSOBJ.24)

   Spawning demon: (EP:ORIENT.1 #{ORIENT.5})
```

```
   EP:ORIENT.1
   =====================================================
   TEST:   An orient is parsed onto wm.
   ACT:    Place it in working memory (ep),
           and put it into the local device.
   =====================================================

Adding to *ep*: #{EPN.7}
Put concept in EPISODIC MEMORY:

(PROCESS &PROCESS.5
 NAME         CATCHES
 SRC          &PHYSOBJ.19    ◄────────────────────────  OBJECT
 DST          &PHYSOBJ.24    ◄────────────────────────  TRIGGER
 REALIZED-EVS (&EVENT.8))

Adding to *ep*: #{EPN.8}
Put concept in EPISODIC MEMORY:

(PROCESS &PROCESS.6
 NAME         COMPRESSING
 SRC         &PHYSOBJ.22     ◄────────────────────────  SPRING
 REALIZED-EVS (&EVENT.6 &EVENT.7))
```

*There are two causal demons, a 'before' and a 'between', monitoring *wm* for process event sequences. In this example we show only the 'between' because that is the one that gets instantiated. The clausal finds the previous event (compressing the spring) right away, but keeps looking for the other.*

*The processing of **IT** occurs in parallel to the causal analysis. **FIND-REF.1** is spawned to look for referents to the word. What is known is that **IT** represents the subject of a connection process **CATCHES** (this is known because the local phrase is processed unsuccessfully first). Since both the **OBJECT** and **SPRING** are in motion it is difficult to resolve the problem. The **SPRING** is the object of compression in the last process read. However, the **OBJECT** is the subject of that phrase. In the present example the running object reference approach was used for the sake of a complete example. The other alternatives have not been fully addressed at the time of writing.*

*The remainder of the phrase is analyzed in much the same way as previous phrases and the **OBJECT**, **TRIGGER**, and **SPRING** are all indexed into **EPM** appropriately. At this point the second event sequence is available to the keyword search and processing continues.*

```
RUNNING DEMON: CLAUSE:CAUSAL2-BET.1
   Executing +act: (CLAUSE:CAUSAL2-BET.1)
   killing demon: CLAUSE:CAUSAL2-BET.1
```

*The +act of **CLAUSE:CAUSAL2-BET.1** is to check the resulting states of the antecedent process against the enablement states of the consequent process. In this example the resulting state of catching the object on the trigger is that the object is restrained from further motion and motion stops (processes watch for their disablement conditions). Since an en-*

ablement condition for compression is motion of the object, and since the restraint of the object transitively feeds through to the spring, the test is positive and the causal relationship can be constructed in EPM.

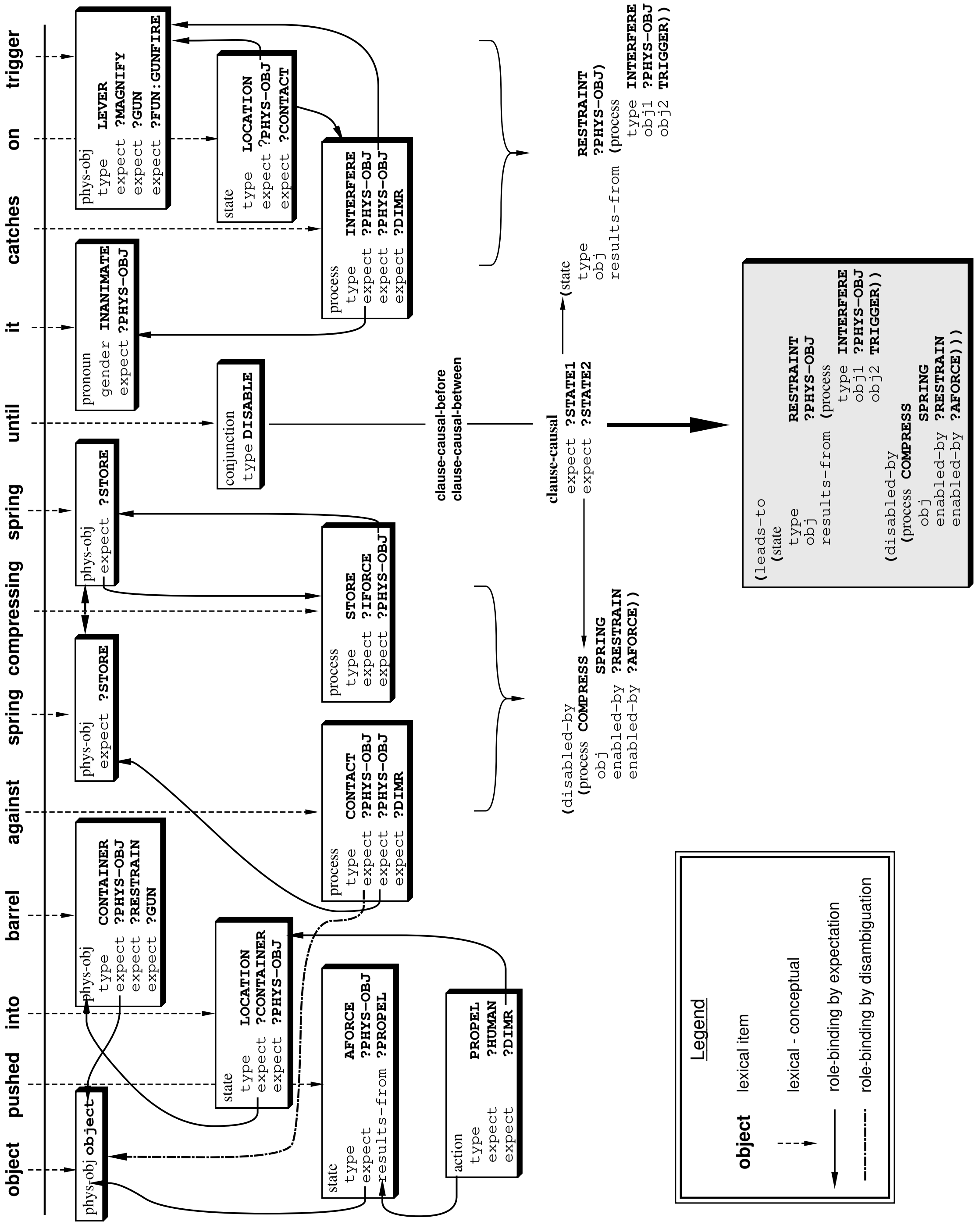


Figure 7.12 Parse diagram for the first sentence of **Toy Gun**

RESULT OF PARSE:

```
(ACTION &ACTION.1
 TYPE          &ACTION.13    ◄─────────────────────────── PROPEL
 NAME          PUSHED
 OBJ           &PHYSOBJ.19   ◄─────────────────────────── OBJECT
 REALIZED-STS  (&STATE1))    ◄─────────────────────────── MOTION

(PROCESS &PROCESS.3
 NAME          CONTAIN
 SRC           &PHYSOBJ.19   ◄─────────────────────────── OBJECT
 DST           &PHYSOBJ.20   ◄─────────────────────────── BARREL
 REALIZED-EVS  (&EVENT.4))

----------------------------

(PROCESS &PROCESS.4
 NAME          AGAINST
 SRC           &PHYSOBJ.19   ◄─────────────────────────── OBJECT
 DST           &PHYSOBJ.22   ◄─────────────────────────── SPRING
 REALIZED-EVS  (&EVENT.5))

----------------------------

SCENE1 of causal

(PROCESS &PROCESS.5
 NAME          COMPRESSING
 SRC           &PHYSOBJ.22   ◄─────────────────── SPRING (original)
 REALIZED-EVS  (&EVENT.6 &EVENT.7))

 (enabled-by process.4, action.1, and some unknown connection)

----------------------------

SCENE2 of causal disables process

(PROCESS &PROCESS.6
 NAME          CATCHES
 SRC           &PHYSOBJ.19   ◄─────────────────────────── OBJECT
 DST           &PHYSOBJ.24   ◄─────────────────────────── TRIGGER
 REALIZED-EVS  (&EVENT.8))

----------------------------

(PHYSOBJ &PHYSOBJ.19
 NAME OBJECT)

(PHYSOBJ &PHYSOBJ.20
 TYPE      &PHYSOBJ.21   ◄─────────────────────────────── TUBE
 NAME      BARREL
 PHYS-ATS &PHYS-ATS.5)   ◄────────── REGION: HOLE - ALONG LENGTH
```

```
(PHYSOBJ &PHYSOBJ.22
 TYPE &SPRING
 NAME SPRING)

(PHYSOBJ &PHYSOBJ.24
 TYPE &PHYSOBJ.25     ◄────────────────── simple lever mechanism
 NAME TRIGGER)

(ORIENT &ORIENT.5
 NAME  ON
 TYPE  ()
 PART1 &PHYSOBJ.19    ◄────────────────────────────── OBJECT
 PART2 &PHYSOBJ.24)   ◄────────────────────────────── TRIGGER
```

Sentence interpretation results as follows. The primary structure is the causal which shows that the last connection (**CATCHES**) disables the elastic (**COMPRESSION**). The *compression* was analyzed as a success state but is known to be enabled by the action (**PUSH**) of the *object*, and some unknown restraint of **SPRING**. We also know that **OBJECT** is restrained (**CONTAINED**) by a **BARREL**. Since the primary process is elastic (**COMPRESSING**), the use of the device for this scene is taken to be the same use of the *compressing* scene, or energy storage.