An Approach to Automated Architectural Floor Layout Generation with Case-based Reasoning

by

Yoshihiro Ichioka, Paul Teicholz, Paul Chinowsky

TECHNICAL REPORT
Number 44 A

March, 1991

Stanford University

If you would like to contact the authors please write to:

c/o CIFE, Civil Engineering,
Stanford University,
Terman Engineering Center
Mail Code: 4020
Stanford, CA 95305-4020

An Approach to Automated Architectural Floor Layout Generation with Case-based Reasoning

by

Yoshihiro Ichioka[1], Paul M. Teicholz[2],
Paul Chinowsky[3]

1 Systems Engineer, Obayashi Corporation and Visiting Research Fellow, *Center for Integrated Facility Engineering,* Stanford University.

2 Director, *Center for Integrated Facility Engineering*, Stanford University.

3 Research Assistant, Civil Engineering, Stanford University.

TABLE OF CONTENTS

[illegible]

Appendices :

1. Abstract

Research into automated floor layout generation has been a problem in which many Architectural / Engineering / Construction (A/E/C) researchers have taken an interest since the 1960s. These researchers have applied many approaches to solve this problem, however, few results which are considered acceptable by architectural designers have been achieved. Since the approaches required quite a few restrictions for analysis, or were focused on a narrow problem domains, the assumption has been that automated layout generation is still not practical for designer's purposes.

The previous approaches were based on mathematical methods dependent on the processing ability of specific computers. In contrast, a recent approach to the layout generation problem has evolved incorporating knowledge-based paradigms such as rules and frames. Teicholz and Chinowsky (Chinowsky 90, Working Paper) have been studying this problem in the "Knowledge-based System for Architectural Layout Generation" project at the Center for Integrated Facility Engineering at Stanford University. In this project, they developed a prototype automated layout generation program for the domain of university facilities. They proposed the idea of storing space relationships and layout configurations in slots within an instance of an applicable frame, and representing design principles with an IF-THEN rule form.

Chinowsky pointed out the difficulty of representing layout case histories as one of the major risks in his project (Chinowsky 90, Research Proposal). The objective of my research is to study a case-based reasoning technique which has begun to attract notable attention from AI researchers, and to examine the possibility of applying the technique to the automated floor layout generation problem.

In this paper I will introduce the basic functions of case-based reasoning systems including Memory Organizing Packages (MOPs) and Thematic Organization Points (TOPs), and some case-based reasoning applications. In addition to this fundamental study of a case-based reasoning system, I will introduce a Case-Based Reasoning Environment (CBRE) developed for this research. CBRE is the development environment for the case-based reasoning application written in Allegro Common Lisp Version 1.3.2 and FrameKit Version 2.0. The reasoning functions of CBRE are based on Micro MOPs, introduced by Riesbeck and Schank (Riesbeck and Schank 89). I translated the function of a frame in Micro MOPs into FrameKit for the purpose of attaching a demon and other basic frame functions. CBRE contains a color, interactive graphical display window to show MOPs hierarchy networks and their slots, in addition to a tracing window to show MOPs created during the reasoning process. Furthermore, I will propose the idea of applying a case-based reasoning technique to the generation of a residential house layout. Since this is still a prototype version

developed in CBRE, only the room size and the building area are taken into account as design constraints.

Finally, I refer to the difficulty of extracting design knowledge from a designer, and the issue of applying a case-based reasoning technique to a generic automated floor layout generation problem.

2. Introduction

2.1. Objectives

The objective of the CIFE project, entitled "A Knowledge-based System for Architectural Layout Generation", is to develop a knowledge-based layout generation system which can work in cooperation with a designer at the conceptual stage of design. In this project, Teicholz and Chinowsky propose the idea of using a frame representation for space relationships and layout configurations, and an IF-THEN rule form for design principles. Chinowsky addresses several issues which could present risks to the project results (Chinowsky 90, Research Proposal). Specifically, he points out the potential difficulty of representing layout case histories within a computer representation, and obtaining a consensus concerning the requirements for case histories.

The objective of my research is to examine the above issues through the application of case-based reasoning techniques. Case-based reasoning provides advantages for this projects as follows:

- Case-based reasoning provides a reduction in the time needed to derive solutions in a search space, compared to rule-based reasoning.
- Case-based reasoning has the possibility to provide solutions in domains which a designer doesn't understand completely.
- Previous experiences provide insights into previous mistakes, and possibilities for eliminating these mistakes in similar situations and also abilities to recall good solutions for similar situations from history.
- Cases provide an understanding of what features are important in a problem domain.

Within my research period at CIFE*, I developed CBRE, a case-based reasoning environment, in Allegro Common Lisp. The case-based reasoning technique was then applied to the generation of residential layouts. Due to the limited development time, the prototype system contains only adapting and storing functions.

* This research was investigated when the principle author, Yoshihiro Ichioka, studied as a Visiting Research Fellow from Obayashi Corporation in Center for Integrated Facility Engineering Stanford University from Septemper 1989 to February 1991.

2.2. A Background of the Layout Generation Problem

Research into automated floor layout generation has been a problem in which many Architectural / Engineering / Construction (A/E/C) researchers have taken an interest since the 1960s. In this chapter, three classical methods : graph networks, shape grammars, and TCR (Total Closeness Rating), are introduced for the purpose of understanding the potential of automated floor layout generation.

2.2.1. Graph Networks

Several researchers including Mitchell *etal.* (Mitchell, Steadman, and Liggett 76), Rinsma (Rinsma 87), Flemming (Flemming 78), Gilleard (Gilleard 78), and Eastman (Baybars and Eastman 80), have used graph network theories to model room adjacencies. As shown in Figure 2.1.1, the simple illustrated layout (left) contains the adjacency graph (top right). The adjacency graph of a room layout is planar*, and has vertices corresponding to the rooms and edges corresponding to their connections. Given a connected planar graph, we can define its corresponding dual graph (bottom right). The dual graph represents the boundaries of the rooms.

The dual graph is the plausible method to transfer from the rooms and their adjacencies to rectangular dissections. However, the graph does not represent a unique dissection , because the alignment of dual edges produces alternative dissections.

2.2.2. Shape Grammars

Shape grammars were introduced as a useful method of generating designs about ten years ago (Stiny 80). Since then, researchers including Chase (Chase 88), Coyne, and Gero (Coyne and Gero 85), have proposed shape grammar methods based on their own design knowledge and experience. Shape grammars use rules related to the shape of rooms to generate layouts.

* A graph G is **planar** if it can be drawn in the plane in such a way that no two edges meet each other except at a vertex to which they are both incident.

Figure 2.2.2 shows a set of design rules for generating simple building forms from walls and columns (Coyne and Gero 85). A dot indicates the top of the room.

Rule(2) represents the following:

IF [room a(1) and short wall(2)]

THEN [copy: room a(1), copy: room c(2), copy: short wall(4), copy: short wall(5), copy: long wall(5), copy: open wall(2)].

Figure 2.2.3 shows the representation of the shape, and Figure 2.2.4 shows a tree structure resulting from the application of the rules given in Figure 2.2.2.

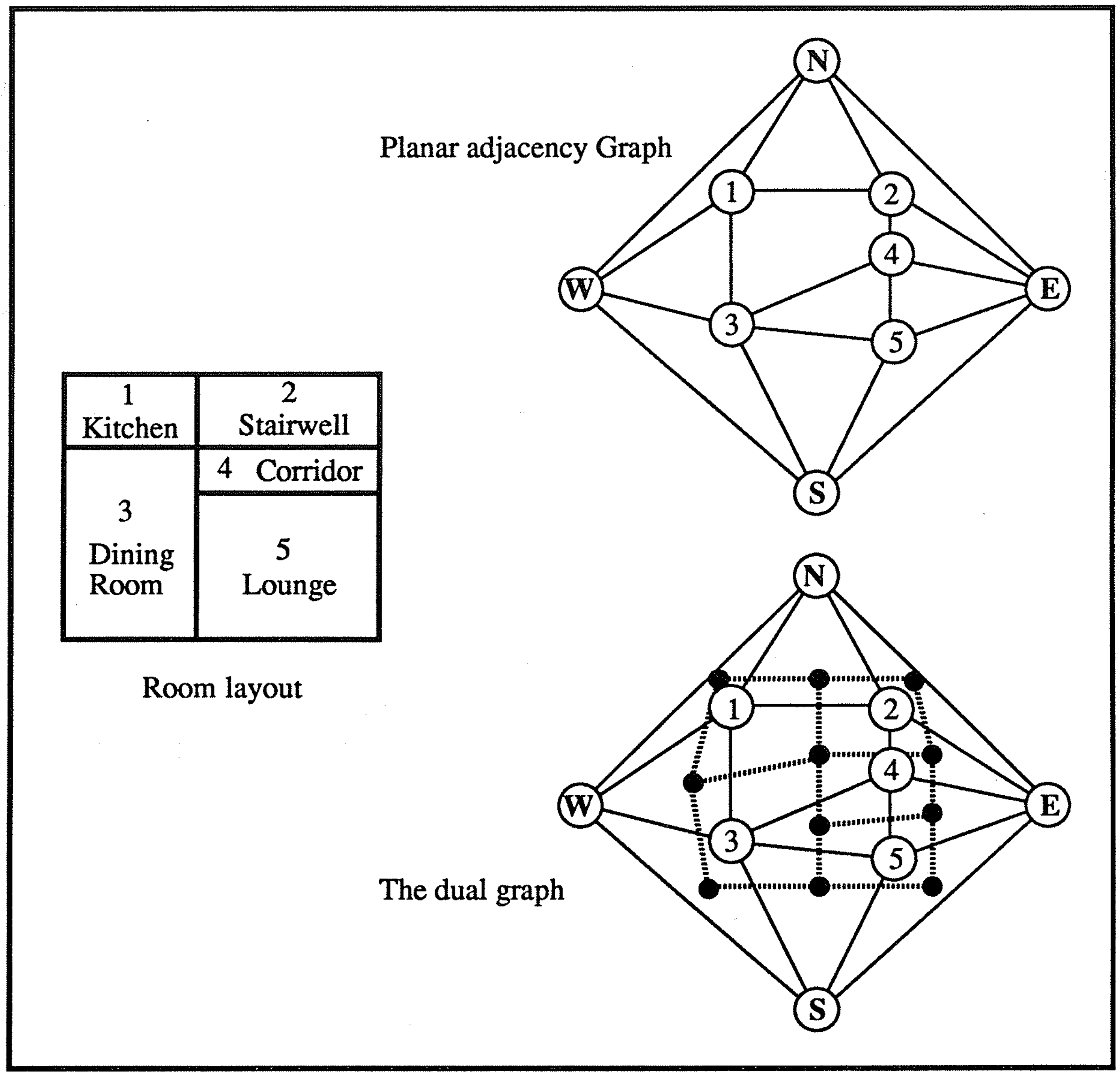


Figure 2.2.1 : Graph Networks Example

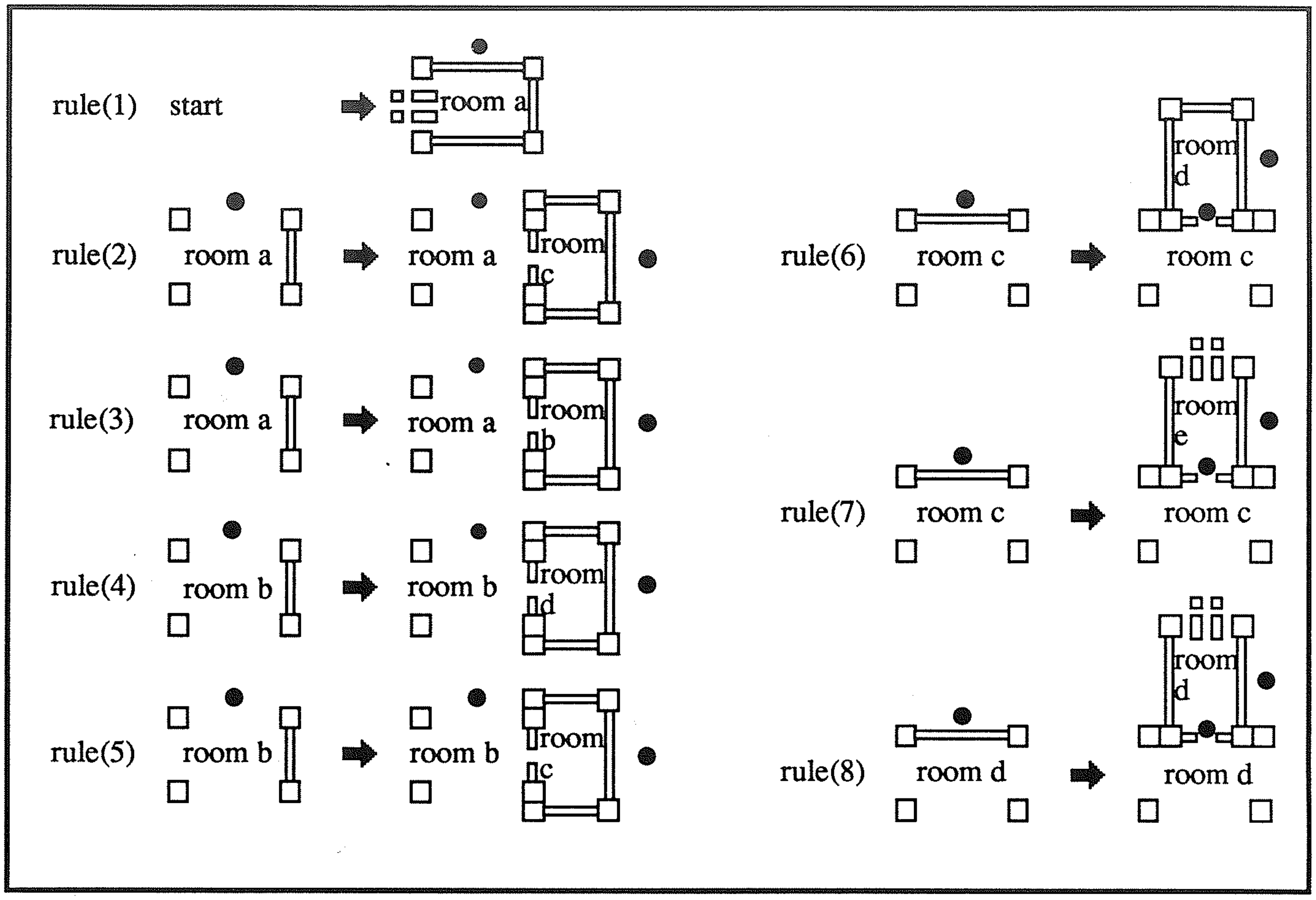


Figure 2.2.2 : Design Rule Examples in Shape Grammars

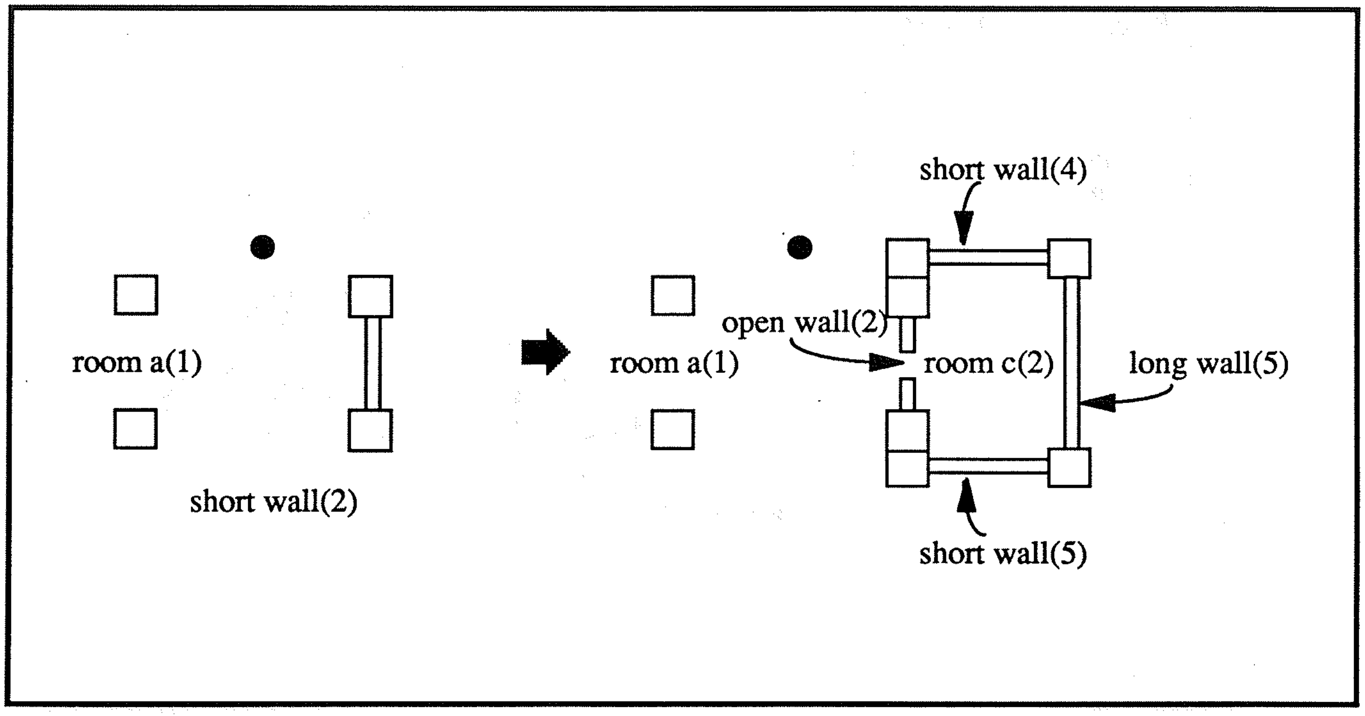


Figure 2.2.3 : The Representation of The Room Shape

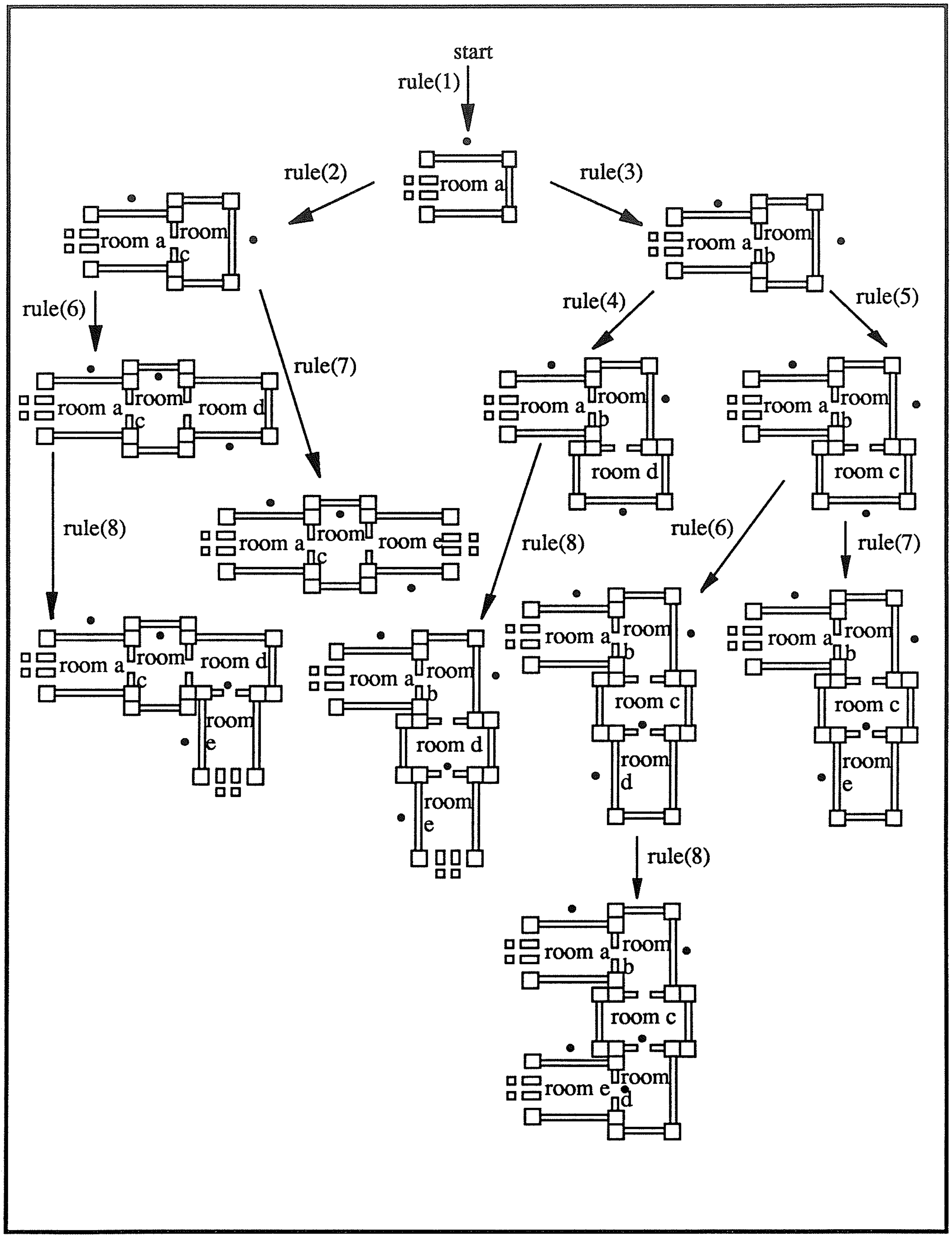


Figure 2.2.4 : The Result Tree for The Rules in Figure 2.2.1

2.2.3. TCR (Total Closeness Rating)

Total Closeness Rating is a computational method based on a Computer-aided Facility Planning algorithm. In this chapter I will introduce CORELAP, an acronym for COmputerized RElationship LAyout Planning (Tompkins and White 84).

In this method, the closeness relationships between a department and all the departments are represented in an activity relationship chart with six alphabetical values (shown in Figure 2.2.6). The TCR is the sum of the numerical values assigned to the closeness relationships (A=6, E=5, I=4, O=3, U=2, X=1). Table 2.2.7 shows the calculation of TCRs for the example problem in Figure 2.2.6. When we make a layout of these departments, the department that has the highest TCR is placed in the center of the layout. If an equal TCR exists, one of the following tie-breaking rules is applied: select the department having the largest area or select the department having the lowest department number. Next, the relationship chart is searched, and if a department is found with an "A" relationship to the selected department, it is added to the layout. If nothing is found, the relationship chart is searched for "E" relationships, "I" relationships, and so forth until a match is found. If there are two or more departments found, the department with the highest TCR is selected. If the department which has an "A" relationship to the first department is found among the unassigned departments, that department is selected as the third department. If there are still two or more candidates, TCR and the tie-breaking rule are utilized. If no unassigned department exists which has an "A" relationship to the second selected department, the searching procedure is repeated in order, for "E" relationships, "I" relationships, and so forth. Whenever a tie is found, TCR and the tie-breaking rule are applied. The searching procedure is continued until all departments are selected.

The location of the selected department in the layout is calculated with the placing rating score. The placing rating score is the sum of weighted closeness ratings between the department being put in the layout and its neighbors. Let's see the example shown in Figure 2.2.5. Now we have to put department 2 into the layout consisting of departments 1 and 7 (Figure (a)). We assumed that department 2 is the next to enter the layout and it has an "A" relationship with department 1 and an "E" relationship with department 7. If an "A" relationship is weighted 64 and an "E" relationship is 16, then the placing rating scores are shown in Figure (b), (c), and (d). If the same placing rating scores exist, the shared boundary lengths are compared. The boundary length is the number of unit square sides which are shared with the department to enter the layout and its neighbors. In Figure (B) and (C) have the boundary length 2 respectively, and (d) has 3.

CORELAP evaluates the result by calculating the layout score. The layout score is defined as follows:

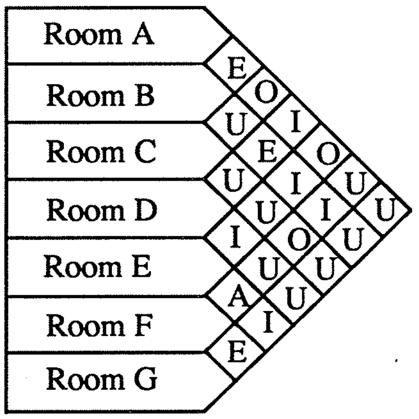
$$\text{Layout score} = \sum_{\text{all department}} \begin{matrix}\text{numerical}\\ \text{closeness}\\ \text{rating}\end{matrix} \times \begin{matrix}\text{length of}\\ \text{shortest}\\ \text{path}\end{matrix}$$

Figure 2.2.6 shows the relation chart of the example. Table 2.2.7 shows the calculation of the TCRs for the example. Figure 2.2.8 shows one of the results of the example calculated by CORELAP, and finally, Table 2.2.9 shows its layout score.

```
                          2 2
1 1 1                   1 1 1                 Placing Rating
1 1 1 7 7 7             1 1 1 7 7 7           Score = 64
   (a)                     (b)

1 1 1   2 2             Placing Rating        1 1 1 2 2        Placing Rating
1 1 1 7 7 7             Score = 16            1 1 1 7 7 7      Score = 80
   (c)                                           (d)
```

Figure 2.2.5 : The Placing Rating Score (CORELAP)



Activity Relationship Chart

Value	Closeness	Weight
A	Absolutely Necessary	243
E	Especially Important	81
I	Important	27
O	Ordinary Closeness OK	9
U	Unimportant	1
X	Undesirable	

Closeness Rating

Figure 2.2.6 : Relationship Chart and Closeness Values

Room Name	Department No.	Relationships	TCR
Room A	1	E, O, I, O, U, U	19
Room B	2	E, U, E, I, I, U	22
Room C	3	O, U, U, U, O, U	14
Room D	4	I , E, U, I, U, U	19
Room E	5	O, I, U, I, A, I	23
Room F	6	U, I, O, U, A, E	22
Room G	7	U, U, U, U, I, E	17

Table 2.2.7 :Calculation of TCRs

```
0  0  0  0  0  0  0  0        0  0  0  0  0  0  0  0
0  0  0  16 16 0  0  0        0  0  0  16 16 0  0  0
0  0  0  0  15 0  0  0        0  0  17 17 15 0  0  0
0  0  0  0  0  0  0  0        0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0        0  0  0  0  0  0  0  0
         (a)                           (b)

0  0  0  0  0  0  0  0        0  14 11 11 13 0  0  0
0  0  12 16 16 0  0  0        0  14 12 16 16 0  0  0
0  0  17 17 15 0  0  0        0  17 17 15 0  0  0  0
0  0  0  0  0  0  0  0        0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0        0  0  0  0  0  0  0  0
         (c)                           (d)
```

Figure 2.2.8 : The Layout Result for Example Figure 2.2.6

Relationship	Preset Value	From	To	Distance	Product of Preset Value and Distance
A	6	E	F	0	0
E	5	A	B	0	0
E	5	B	D	0	0
E	5	F	G	0	0
I	4	A	D	0	0
I	4	B	E	2	8
I	4	B	F	0	0
I	4	D	E	3	12
I	4	E	G	0	0
O	3	A	C	0	0
O	3	C	F	0	0
O	3	A	E	2	6
U	2	A	F	0	0
U	2	A	G	1	2
U	2	B	C	2	4
U	2	B	G	0	0
U	2	C	D	2	4
U	2	C	E	1	2
U	2	C	G	2	4
U	2	D	F	2	4
U	2	D	G	1	2
				Layout score	48

Table 2.2.9 : Calculation of Layout Score

3. Case-based Reasoning System

3.1. Case-based Reasoning

If we examine our ordinary thinking, we use case-based reasoning to solve problems without noticing it. We remember the result and problem solving processes of old cases, and try to apply them to the new problem.

On this year's Thanksgiving holiday, my wife decided to make a traditional pumpkin pie just like Americans do. Since before this, we ate a fabulous pumpkin pie which we bought at the farmer's market, she tried to make one exactly like that.

First, she used some mashed pumpkin, several spoonfuls of granulated sugar and some cream as filling ingredients. However, the result of this case wasn't good enough to satisfy us. The pumpkin filling was more yellowish than the one we got at the farmer's market, and its flavor was less than we expected. This failure reminded my wife of previous cases involving an apple pie and a fruit cake. The apple pie case reminded her of using brown sugar to make the pie brown, and the case of the fruit cake reminded her that she used caramel sauce to make the cake more flavorsome and brown. So, the second time she used brown sugar and caramel sauce instead of granulated sugar. These two memories brought us a good result. They worked effectively and successfully. Something about the first failure reminded her of the previous trials. As I mentioned above, case-based reasoning is used in the various kinds of everyday common-sense reasoning.

Generally, we can solve a problem easier the second time than the first, because we can adapt the previous experience to the new one.

A case-based system consists of four major parts (Figure 3.3.1).

These are as follows:

(1) Retrieve

(2) Adapt

(3) Evaluate

(4) Store

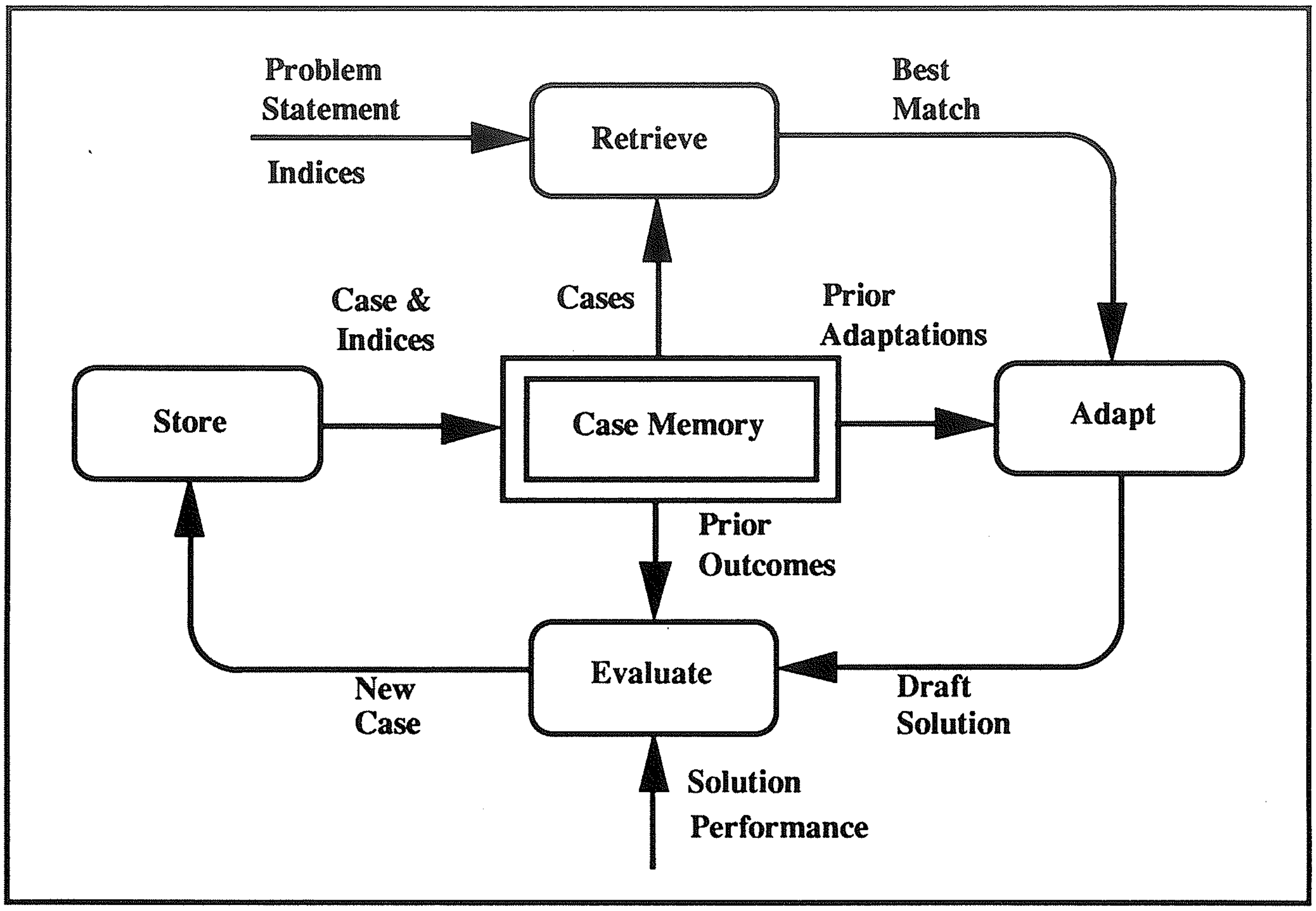


Figure 3.1.1 : Case-based Reasoning System Basic Flow

3.2. Retrieve

During the retrieving stage, the system finds the most appropriate case in its case-memory to adapt. Specifically, if when the system can not find a solution which satisfies all of the requirements of a new case, then it tries to find the one which covers the most requirements.

The ability to understand the new problem in terms of old experiences has two parts : recalling old experiences and interpreting the new situation in terms of the recalled experiences. The first is called the **indexing problem**. An index is assigned when a case is stored in case memory for the purpose of retrieving the case under appropriate conditions. The second is the process of comparing the new problem to the previous experiences.

When we implement case-base reasoning systems on a computer, we have to think about the organization of its case memory and the retrieving algorithm. With regard to this issue, several methods have been implemented including flat memory, hierarchical memory, prioritized discrimination nets, and redundant discrimination nets (Kolodner and Riesbeck 90).

3.3. MOPs

In this chapter I will introduce MOPs ; Memory Organization Packages, as an implementation of hierarchy memory. Shank described the basic MOP concept in his book titled "Dynamic Memory" (Shank 82) .

According to his representation ;

In order to *account for reconstructive memory, and the ability to generalize and learn from past experience, I am proposing that a memory structure exists that I call a MOP. It follows from what I have said so far that a MOP is also a processing structure. As a memory structure, the role of a MOP is to provide a place to store new inputs. As a processing structure, the role of a MOP is to provide expectations that enable the prediction of future inputs or inference of implicit events on the basis of previously encountered, structurally similar, events. A MOP processes new inputs by taking the aspects of those inputs that relate to that MOP and interpreting those aspects in terms of the past experiences most closely related to them. Many different high-level memory structures can be relevant at any given time in processing an input, i.e., any of a number of different MOPs may applicable at one time.*

He also defined the idea of scene and script which constitute MOPs.

A *scene defines a setting, an instrumental goal, and actions that take place in that setting in service of that goal. These actions are defined in terms of specific and generalized memories relating to that setting and goal. For example, ordering in a restaurant or getting your baggage in an airport are scenes. As long as there is an identifiable physical setting and a goal being pursued with that setting, we have a scene. Two kinds of information are present in a scene. First, we have physical information about what the scene looks like. Information about what was in one's line of sight can be part of one's remembrance of a scene. Second, we have information about the activities that go on in a scene.*

A script, in *our new, narrower, scene, is a sequence of actions that take place within a scene. Many scripts can encode the various possibilities for the realization of a scene. That is a script instantiates or color a screen. Scenes contain general information; scripts provide the specifics.*

Schank explained MOPs functions using the example of visiting a doctor's office.

I went to the doctor's yesterday. While I was reading a magazine I noticed that a patient who arrived after me was being taken ahead of me. The doctor will probably still overcharge me!

For this example he chose two kinds of MOPs, M-PROFESSIONAL OFFICE VISIT and M-CONTRACT (Figure 3.3.1). Each of these MOPs organizes scenes and scripts in terms of visiting a doctor's office. Waiting Room, for example, is one of the scenes in M-PROFESSIONAL OFFICE VISIT.

The primary function of M-PROFESSIONAL OFFICE VISIT is to provide the correct sequence of scenes when executing this action. Now let's see how this MOP works. The phrase "went to a doctor" refers to "doctor", which is something about which we have information. Doctor is a token in memory. For every token in memory, there exists information attached to that token which indicates where to look for further information.

M-PROFESSIONAL OFFICE VISIT has information attached to it concerning what other MOPs might also be active when it is active. When we imagine the scene of visiting a doctor, first we have to wait our turn in the waiting room, after our turn comes we see the doctor, and we pay some money for medical treatment before we get back home.

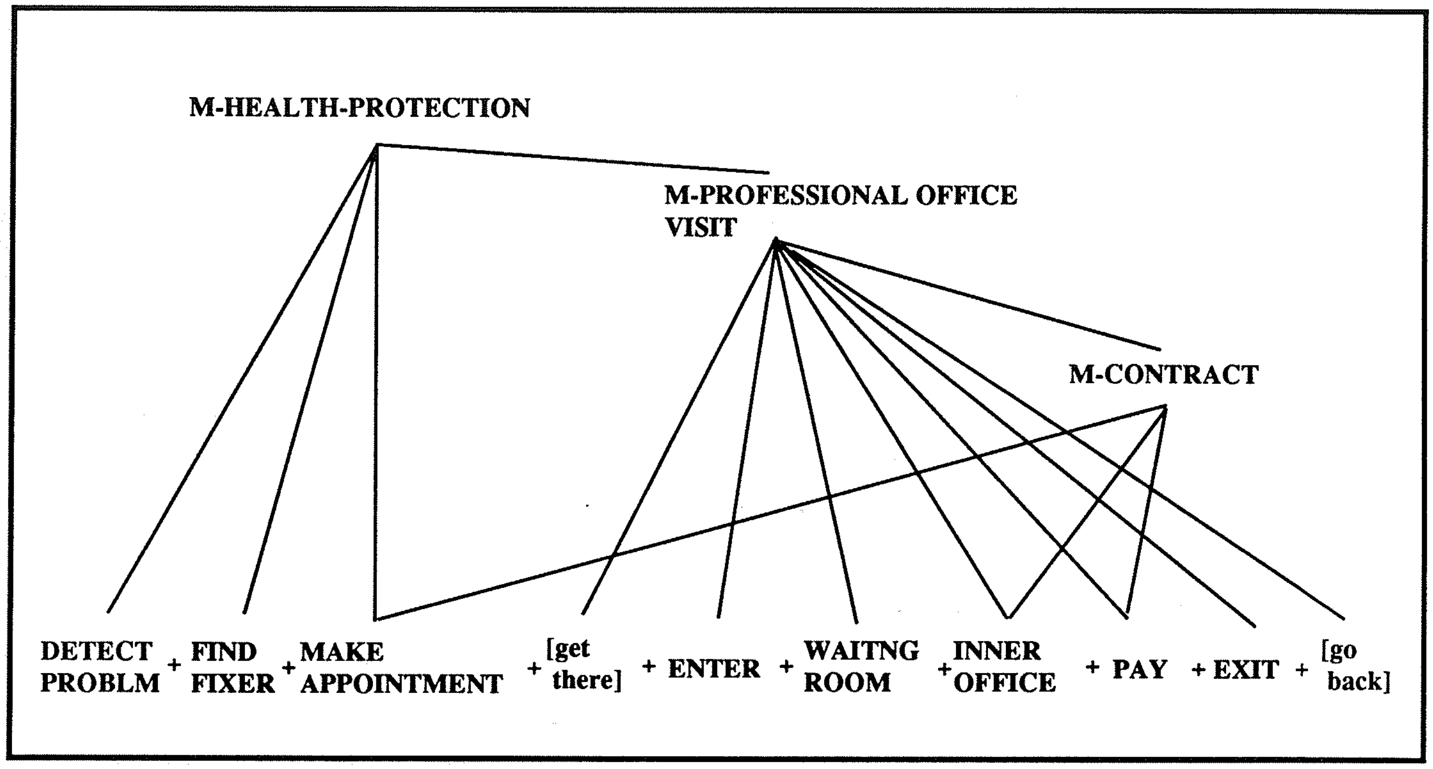

M-HEALTH-PROTECTION
M-PROFESSIONAL OFFICE VISIT
M-CONTRACT
DETECT PROBLM + FIND FIXER + MAKE APPOINTMENT + [get there] + ENTER + WAITNG ROOM + INNER OFFICE + PAY + EXIT + [go back]

Figure 3.3.1 : Schank's MOPs Example

The MOP M-CONTRACT is invoked when M-PROFESSIONAL OFFICE VISIT is activated. This represents the unexpressed contract between a doctor and a patient. The doctor gives a bill to the patient for his or her medical examination. If the patient refuses to pay for it, the patient will be sued. All information about this should be in the MOP M-CONTRACT, not M-PROFESSIONAL OFFICE VISIT. Hence, a MOP has two sorts of information : One is a sequential set of scenes and the other is a relationship with the other MOPs.

A MOP organizes a set of scenes and scripts which lead to a goal in memory. Namely M-PROFESSIONAL OFFICE VISIT organizes a set of information about visiting a professional office, it would not be involved in any higher level MOPs. Generally, people go to see a doctor to take care of their health. However, this is not a scene or a script in M-PROFESSIONAL OFFICE VISIT, but a goal someone wants to achieve. Nevertheless, that information is not included in M-PROFESSIONAL OFFICE VISIT, that is also invoked by the phrase "went to a doctor". Thus we define the MOP M-HEALTH PROTECTION as a higher level MOP. As you can see in Figure 3.3.1 there are three activity MOPs : M-HEALTH PROTECTION, M-PROFFESSIONAL OFFICE VISIT, and M-CONTRACT.

M-PROFESSIONAL OFFICE VISIT has information as follows:

[get there] + WAITING ROOM + GET SERVICE + PAY + [get back]

M-CONTRACT forms:

[get contract] + NEGOTIATE + AGREE + DELIVER + PAY

The following items are organized by MOPs:

Items organized by MOPs	Denoted	Example
Scripts	$XXX	$HERTZ-COUNTER
Scenes	XXX	RENT-A-CAR COUNTER
Place Holders	[xxx]	[get transportation]

Here, Place Holder is a parameter which comes from other knowledge. For example, M-PROFESSIONAL OFFICE VISIT has the Place Holder [get there]. Quite a few MOPs, scenes, or scripts can fill up that area. If someone takes a bus to visit a doctor, then the MOP M-BUS will fit that spot.

Schank explained the function of Place Holders:

The most important aspect of the structures organized by MOP is that they be general enough to be used by other MOPs. This is a key point. Scenes and scripts organized by one MOP can also be organized by others. For example, the PAY scene is used by a great many MOPs in addition to M-POV. If you lost your wallet you might attempt to figure out where you had it last. You might remember putting it down near a cash register while paying. The problem then would be differentiate one PAY event from another. The fact that this is difficult indicates that PAY is a shared structure.

As he pointed out, the first sentence, "*I went to the doctor's yesterday,*" gets us to look at what we know about doctors. In this example, those are M-PROFESSIONAL OFFICE VISIT and M-CONTRACT. When the following sentence "*While I was reading a magazine I noticed that a patient who arrived after me was being taken ahead of me*" is matched, it can be interpreted as a WAITING ROOM in the MOP M-PROFESSIONAL OFFICE VISIT. What happened here, is the failure of the script, named "customer queueing" which is organized by WAITING ROOM. Thus, the actual situation is explained by way of M-PROFESSIONAL OFFICE VISIT, WAITING ROOM, and $CUSTOMER QUEUE. The last sentence "T*he doctor will probably still overcharge me!*" brings up the scene PAY of M-CONTRACT.

We have three MOPs in a story of "doctor's visit", each MOP has a linked line to the many kinds of scenes which contain action information. Thus, a MOP is an organizer of scenes.

3.4. TOPs

In Dynamic Memory (Shank 82) Schank gave the idea of TOPs; Thematic Organization Points.

The key to remindings, memory organizing and generalization is the ability to create new structures that coordinate or emphasize the abstract significance of a combination of episodes. Structures that represent this abstract, domain-independent, information we call Thematic Organization Points or TOPs. Tops are responsible for our ability to:

1. *Get reminded of a story that illustrates a point.*
2. *Come up with adages such as, "A stitch in time saves nine" or "Neither a borrower nor a lender be" at an appropriate point.*

3. *Recognize an old story in new trappings.*
4. *Notice co-occurrences of seemingly disparate events and draw conclusions from their co-occurrence.*
5. *Know how something will turn out because the steps leading to it have been seen before.*
6. *Learn information from one situation that will apply in another.*
7. *Predict an outcome for a newly encountered situation.*

Schank addressed his idea in the example of the similarity between the stories Romeo and Juliet and West Side Story. In this example he pointed out that both stories are similar because of the following key factors:

1. young lovers
2. objections of parents
3. an attempt to get together surreptitiously
4. a false report of death
5. the false report causes a real death of one of the lovers

He categorized both stories as cases of young lovers with a **mutual goal pursuit against outside opposition** which correspond to the TOP MG; OO. Thus, a TOP has both a goal type and a condition on that goal. Figure3.4.1 shows some TOPs that Schank proposed in his book: Dynamic Memory.

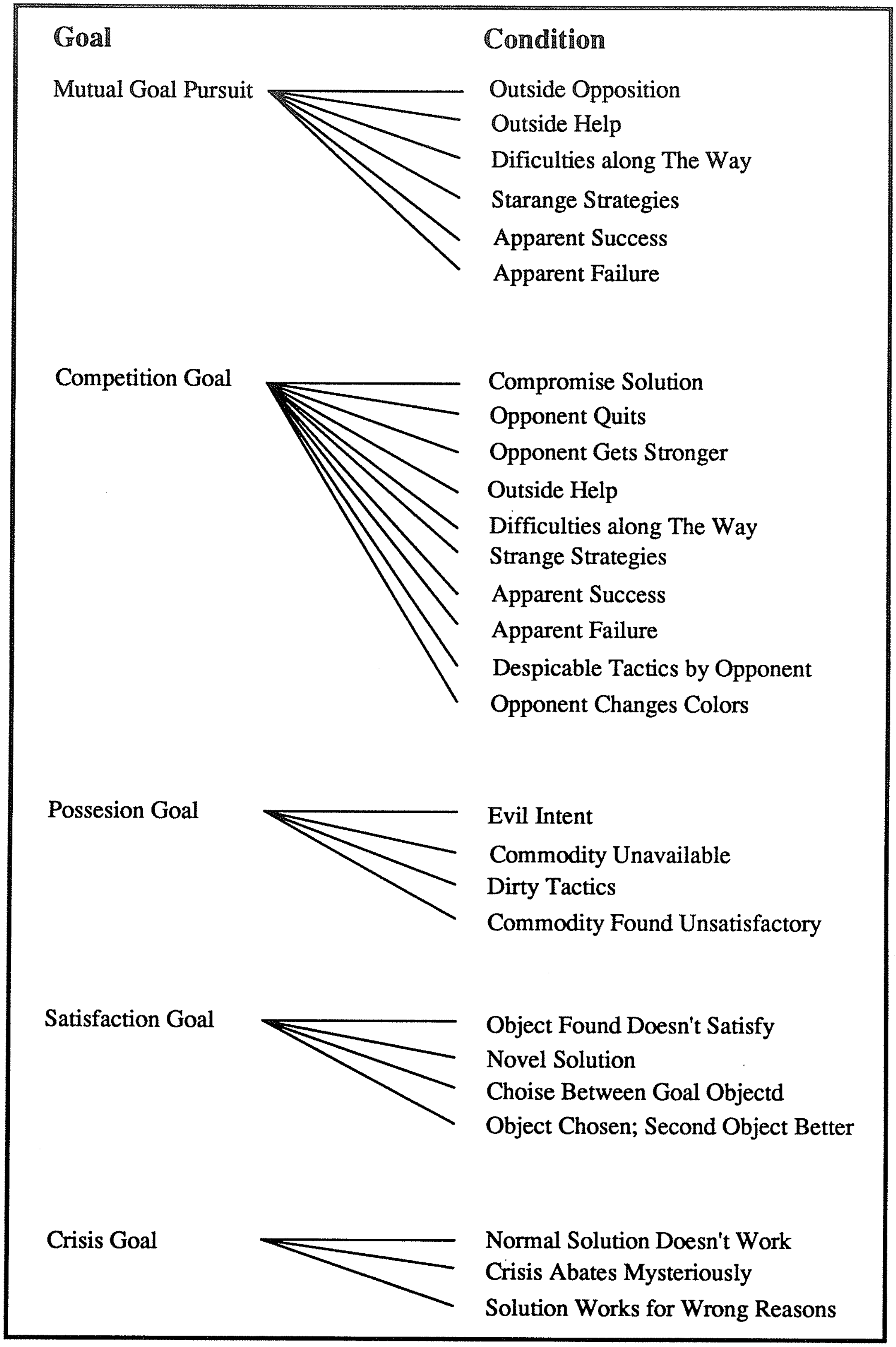


Figure 3.4.1 : Schank's TOPs Example

In another example, Hammond used her own idea of TOPs in the case-based reasoning program called CHEF. She proposed the use of TOPs for planning problems in the book: Case-Based Planning (Hammond 89). CHEF stores its TOPs in a discrimination network, indexed by the features that describe the problem that they correspond to. In searching for a TOP, CHEF extracts these features from its explanation of the current problem and the TOP suggests strategies to solve that problem. CHEF's TOPs have two components: the feature of the problem used to index to it and strategies that it suggests to deal with that problem.

CHEF uses sixteen TOPs as follows:

SIDE-EFFECT (SE)

TOPs in this group describe the problems with the side-effects of one step interfering with the overall plan in one way or another.

- SE: DISABLED-CONDITION: CONCURRENT
- SE: DISABLED-CONDITION: SERIAL
- SE: DISABLED-CONDITION: BALANCE
- SE: BLOCKED-PRECONDITION
- SE: GOAL-VIOLATION

DISIRED-EFFECT (DE)

TOPs in this group describe the situations in which the desired effect of a step, that is a state that either directly satisfies a goal or enables the sat isfaction of a goal, causes problems later in the plan.

- DE: DISABLED-CONDITION: CONCURRENT
- DE: DISABLED-CONDITION: SERIAL
- DE: DISABLED-CONDITION: BALANCE
- DE: BLOCKED-PRECONDITION

SIDE-FEATURE (SF)

TOPs in this group relate to the situations in which the features of new items that have been added to plans interfere with the goal that the plans are trying to achieve.

- SF: ENABLEDS-BAD-CONDITION
- SF: BLOCKED-PRECONDITION
- SF: GOAL-VIOLATION

DESIRED-FEATURE (DF)

TOPs in this group describe the situations in which a feature of an object that actually satisfies a goal causes trouble.

- DF: DISABLED-CONDITION
- DF: BLOCKED-PRECONDITION

STEP-PARAMETER (SP)

TOPs in this group relate to the problems involving the amount of time a step takes and the tools it uses.

- SP: TIME
- SP: TOOL

Figures 3.4.2-1 to 3.4.2-3 illustrate the use of the CHEF system.

```
Searching for plan that satisfies -
        Include beef in the dish.
        Include broccoli in the dish.
        Make a stir-fry dish.

Found recipe -> REC2 BEEF-WITH-GREEN-BEANS

Recipe exactly satisfies goals ->
        Make a stir-fry dish.
        Include beef in the dish.

Recipe partially matches ->
        Include broccoli in the dish.
                in that the recipe satisfies:
                        Include vegetables in the dish.

Building new name for copy of BEEF-WITH-GREEN-BEANS
Calling recipe BEEF-AND-BROCCOLI

Modifying recipe: BEEF-AND-BROCCOLI
to satisfy: Include broccoli in the dish.

Placing some broccoli in recipe BEEF-AND-BROCCOLI

- Considering ingredient-critic:
  Before doing step: Stir fry the -Variable-
   do: Chop the broccoli into pieces the size of chunks.
- ingredient-critic applied.

Checking goals of recipe -> BEEF-AND-BROCCOLI

Recipe -> BEEF-AND-BROCCOLI has failed goals.

The goal: The broccoli id now crisp.
is not satisfied.
It is instead the case that: The broccoli is now soggy.
```

Figure 3.4.2-1 : Hammond's CHEF TOPs Example

```
Unfortunately: The broccoli is now a bad texture.
In that: The broccoli is now soggy.

Changing name of recipe BEEF-AND-BROCCOLI
to BAD-BEEF-AND-BROCCOLI

Found TOP TOP1 -> SIDE-EFFECT: DISABLED-CONDITION:
CONCURRENT
TOP -> SIDE-EFFECT: DISABLED-CONDITION: CONCURRENT has
3 strategies associated with it:
          SPLIT-AND-REFORM
          ALTER-PLAN: SIDE-EFFECT
          ADJUNCT-PLAN


Applying TOP -> SIDE-EFFECT: DISABLED-CONDITION:
CONCURRENT
to failure it it not the case that: The broccoli is now crisp.
in recipe BAD-EFFECT-AND-BROCCOLI

Asking questions needed for evaluating strategy:
SPLIT-AND-REFORM

ASKING -> Can plan
          Stir fry the sugar, soy sauce, rice wine, garlic,
          corn starch, broccoli and beef for three minutes.
          or split and rejoined

Found plan: Instead of doing step: Stir fry the sugar,
soy sauce, rice wine, garlic, corn starch, broccoli
and beef for three minutes
do:
S1 = Stir fry the broccoli for three minutes.
S2 = Remove the broccoli from the result of action S1.
S3 = Stir fry the sugar, soy sauce, rice wine, garlic,
corn starch and beef for three minutes,
S4 = Add the result of action S2 to the result of action S3.
S5 = Stir fry the result of action S4 for a half minute.

Asking questions needed for evaluating strategy:
     ALTER-PLAN-SIDE-EFFECT
```

Figure 3.4.2-2 : Hammond's CHEF TOPs Example (continued)

```
ASKING -> Is there an alternative to
              stir fry the sugar, soy sauce, rice wine, garlic,
              corn starch, broccoli and beef for three minutes.
              that will enable
              The dish now tastes savory.
              which does not cause
              There is thin liquid in the pan from the beef
              equaling 4.8 teaspoons.

No alternative plan found

Asking questions needed for evaluating strategy: ADJUNCT-PLAN

ASKING -> Is there an adjunct plan that will disable
              There is thin liquid in the pan from the beef
              equaling 4.8 teaspoons,
              that can be run with
              Stir fry the sugar, soy sauce, rice wine, garlic,
              corn starch, broccoli and beef for tree minutes.

No alternative plan found

Deciding between modification plans suggested by strategies:
Only one modification can be implemented -> SPLIT-AND-REFORM

Implemented plan -> Instead of doing step: Stir fry the sugar,
soy sauce, rice wine, garlic, corn starch, broccoli
and beef for three minutes.
do:
S1 = Stir fry the broccoli for three minutes.
S2 = Remove the broccoli from the result of action S1.
S3 = Stir fry the sugar, soy sauce, rice wine, garlic,
corn starch and beef for three minutes,
S4 = Add the result of action S2 to the result of action S3.
S5 = Stir fry the result of action S4 for a half minute.
Suggested by strategy SPLIT-AND-REFORM.
```

Figure 3.4.2-3 : Hammond's CHEF TOPs Example (continued)

3.5. Adapt

In the adapting stage, the system modifies the retrieved case to fit the current requirements. When we looked at the previous applications using case-based reasoning, they used the following adaptations in their adapting process.

- Inserting values and structures
- Deleting values and structures
- Substituting values and structures for new ones
- Splitting a problem into a parts

Kolodner and Riesbeck introduced several adapting methods (Koldner and Riesbeck 90) as follows:

(1) Reinstantiation
(2) Parameter Adjustment
(3) Local Search
(4) Query Memory
(5) Specialized Search
(6) Transformation
(7) Critic Application
(8) Derivational Replay

(1) Reinstantiation

Reinstantiation is a method to substitute a slot's values in the old case for the ones in a new case when using frame representations for organizing case memory.

(2) Parameter Adjustment

Parameter Adjustment is a method to change parameters in the old case to the new one by means of using adjustment heuristics which are derived from the relationship between problem features and solution parameters.

(3) Local Search

Local Search is a method to substitute other values by searching for a close relative in its abstract hierarchies.

(4) Query Memory

Query Memory is a method to search memory for the described element when Local Search is inapplicable or yields no answer. With this method a description of the needed element must be available after Local Search.

(5) Specialized Search

Specialized Search is a method to give memory instructions about how to look for an alternate.

(6) Transformation

Transformation is a method to transform the element of the old case to fit the new problem using one of the following heuristics:

- Delete a secondary feature
- Substitute a feature
- Add a feature
- Adjust the amount of a feature

(7) Critic Application

Critic Application uses critics which are indexed by the combinations of features that trigger them, and initiates the critics whose features are present in a problem or solution.

(8) Derivational Replay

Derivational Replay is a method to recall how the old value was computed and replay that computation to get the solution for the new case.

3.5. Evaluate

This stage is one of the most important for a case-based reasoner. In evaluation, the results of the adapted case are tested in the real world. If the results were as expected, it is accepted and no more analysis is done. On the other hand, if the results were different than expected, explanation would be needed.

3.6. Store

Finally, in the storing stage, the executed reasoning result is stored in case memory for future use. The most important process at this stage is indexing the new case in case memory. An index must be selected for recalling the case when it is needed at a later time.

4. CBRE (Case-Based Reasoning Environment)

4.1. CBRE Functional Overview

The case-based reasoning environment introduced in this chapter is implemented in Allegro Common Lisp and FRAMEKIT on a Macintosh. FRAMEKIT is a frame-based knowledge representation environment written in Common Lisp. It was first developed by Jamie Carbonell and enhanced by Eric Nyberg. FRAMEKIT provides the basic functions of frames, instances, inheritance, and demons.

Fundamentally, the case-based reasoning environment is based on Micro MOPs which was introduced in Riesbeck and Schank's book "INSIDE CASE-BASED REASONING" (Schank and Riesbeck 89). Since the part of frame-based data handling in Micro MOPs is translated in FRAMEKIT, the case-based reasoning environment will be more flexible than Riesbeck and Schank's.

Figure 4.1.1 shows the case-based reasoning environment main menu and submenus. "**CBR-Menu**", shown at upper left, is the main menu and has five menu items, "**MOP-File**", "**MOP-Define**", "**MOP-Utility**", "**MOP-Edit**", and "**Quit-CBR**". The following descriptions introduce the menu items and submenu items. "Micro MOPs CHEF" is used as the example in each description.

MOP-File When main menu "**MOP-File**" is clicked, the small submenu which has "**New**" and "**Update**" items pops up. The user can choose either a new file or a previous file. Both "**New**" and "**Update**" items invoke Macintosh's "**File Allocation**" window.

MOP-Define Main menu item "**MOP-Define**" pops up "**MOP Definition**" window shown as Figure 4.1.2. The user can define easily MOP and instance with their slots.

MOP-Utility Main menu item "**MOP-Utility**" invokes submenu "**Display**" which has seven menu items, "**MOPs-List**", "**SLOTs-List**", "**MOP-Hierarchy**", "**SLOT-Hierarchy**", "**Cases**", "**New-MOPs**", and "**Quit-Display**" shown as Figure 4.1.1.

MOPs-List	Submenu item “MOPs-List” returns an abstract hierarchy list below a MOP keyed in after the prompt “enter MOP name ==>”. An abstract hierarchy is the context network structure in terms of the inheritance among MOPs and instances shown as Figure 4.1.3.
SLOTs-List	Submenu item “**SLOTs-List**” returns a slot hierarchy list below a MOP or an instance keyed in after the prompt “enter MOP name ==>”. A slot hierarchy list is the context network structure in terms of the relationship of slot roles and their fillers shown as Figure 4.1.4.
MOP-Hierarchy	Submenu item “**MOP-Hierarchy**” pops up “Hierarchy Window” window shown as Figure 4.1.5. This window shows an abstract hierarchy list below a MOP or an instance keyed in after the prompt “enter MOP name ==>” graphically. At this window the user can select several functions from submenu “**MOPs**”: submenu item “**Scroll**” scrolls the abstract hierarchy on the window up, down, right, and left easily, “**Show-slots**” pops up the slots window of MOPs or instances on “Hierarchy Window” window, “**Zoom-in**” and “**Zoom-out**” increases and shrinks the abstract hierarchy on the window re spectively, “**Quit**” deletes “Hierarchy Window” window and returns the control to “**Display**” submenu.
SLOT-Hierarchy	Submenu item “**SLOT-Hierarchy**” pops up graphical window which shows the network of slot roles and their fillers below a MOP and an instance keyed in after the same prompt , “enter MOP name ==>”, as I mentioned above . As shown in Figure 4.1.6 this window supports the same functions as “**MOP-Hierarchy**” except “**Show-slots**” function. Figure 4.1.6 shows “i-m-recipe-steps.10”, the filler of role “steps” in “i-m-chicken-and peanuts” MOP.

Cases	Submenu item "Cases" shows "Cases in memory" radio-button window on the screen shown as Figure 4.1.7. The cases in memory, the same as the instances below "m-case" MOP, are displayed whenever he or she clicks "Cases" submenu item.
New-MOPs	Submenu item "**New-MOPs**" pops up the scroll window shown as Figure 4.1.8. This window shows all MOPs and instances which are created automatically during processing. When the user clicks the name on this window, its slot hierarchy list is returned.
Quit-Display	Submenu item "**Quit-Display**" returns the control to main menu "**CBR-Menu**".

MOP-Edit	Main menu item "**MOP-Edit**" pops up Macintosh's File Edit window of the file chosen at "**MOP-File**" menu item.
Quit-CBR	Main menu item "**Quit-CBR**" quits Case-based reasoning environment and returns the control to general Allegro Common Lisp environment.

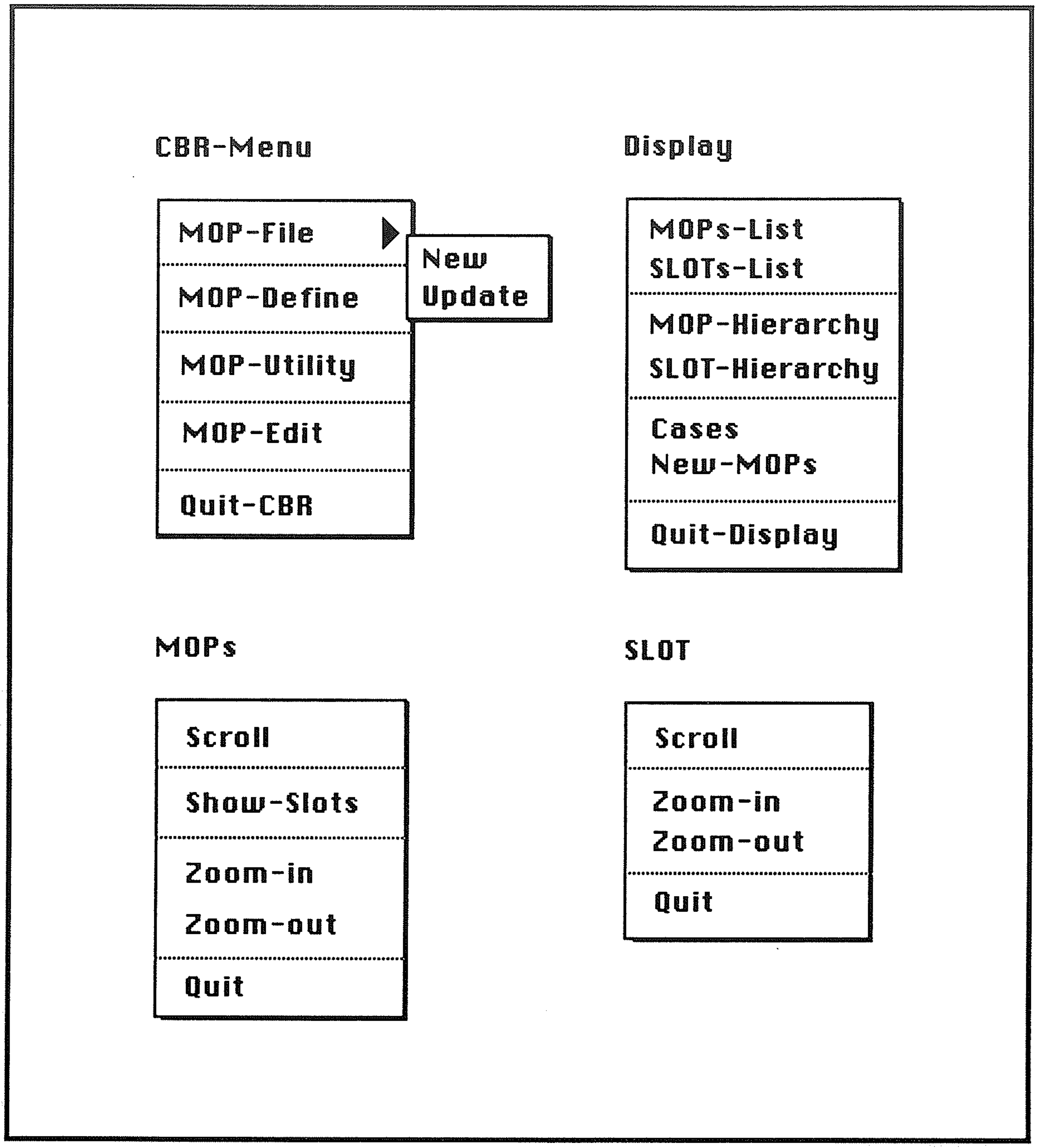
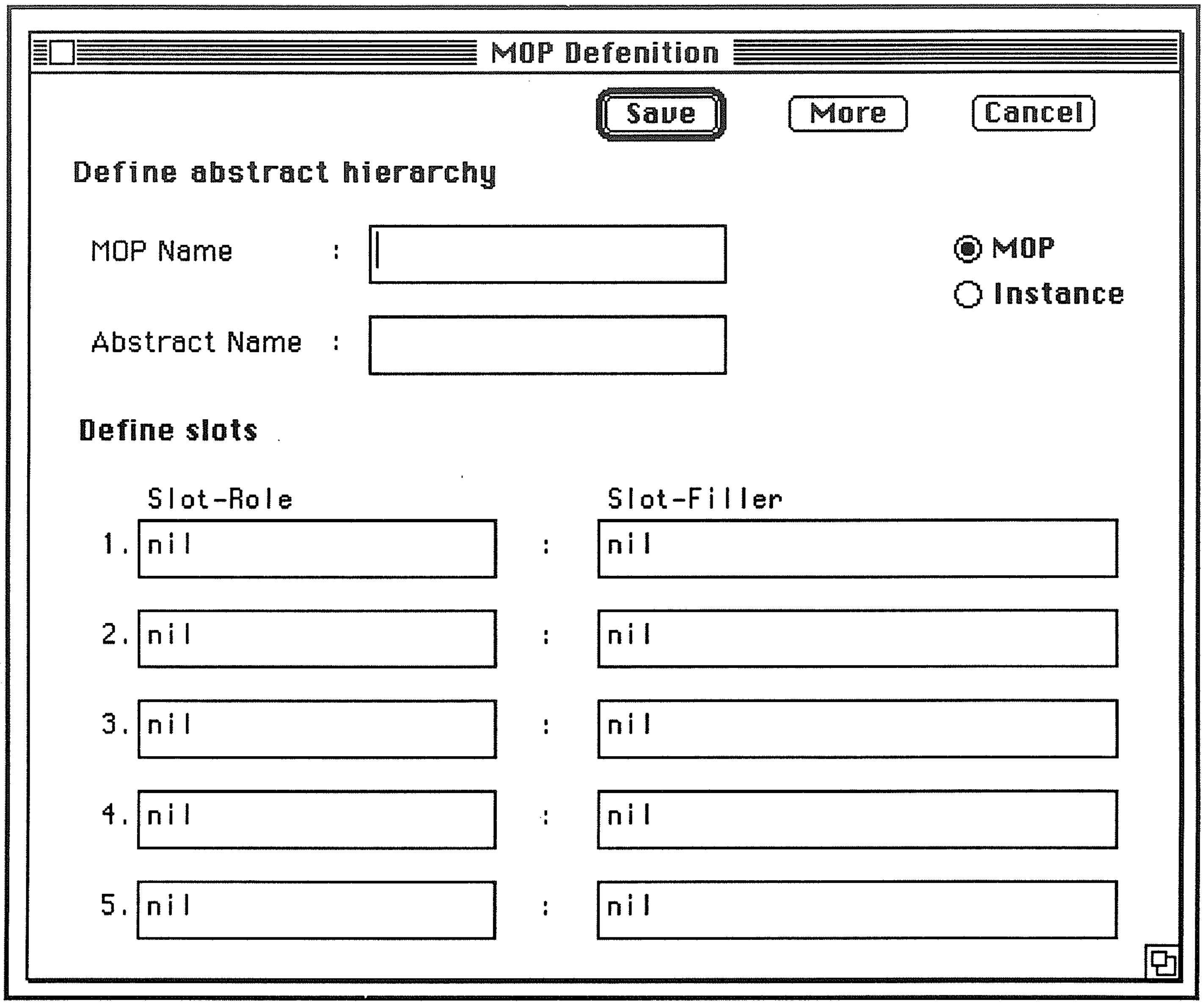


Figure 4.1.1 : CBRE Main Menu and Submenu



Define abstract hierarchy

MOP Name :

Abstract Name :

[✓ MOP] Instance

Define slots

	Slot-Role		Slot-Filler
1.	nil	:	nil
2.	nil	:	nil
3.	nil	:	nil
4.	nil	:	nil
5.	nil	:	nil

Figure 4.1.2 : MOP-Define Window

```
? enter MOP name ==> m-ingred

( M-INGRED    ( M-SPICES      ( I-M-SPICES ))
              ( M-SMALL-INGRED     ( I-M-PEANUTS ) ( I-M-GREEN-BEANS )
              ( I-M-SNOW-PEAS ))
              ( M-LARGE-INGRED     ( I-M-GREEN-PEPPERS ) ( I-M-BROCCOLI )
              ( I-M-CHICKEN ) ( I-M-BEEF ))
              ( M-VEGETABLE
                ( M-STURDY-VEGETABLE    ( I-M-PEANUTS ) ( I-M-GREEN-PEPERS )
                        ( I-M-GREEN-BEANS ))
                        ( M-CRISP-VEGETABLE  ( I-M-SNOW-PEAS ) ( I-M-BROCCOLI )))
               ( M-MEAT     ( M-BONY-MEAT                ( I-M-CHICKEN ))
                            ( M-BONELESS-MEAT        ( I-M-BEEF ))))
```

Figure 4.1.3 : MOPs-List Return Example

```
? enter MOP name ==> m-bone-step

( M-BONE-STEP      ( ACTION      I-M-BONE )
                   ( OBJECT      M-BONY-MEAT ))
```

Figure 4.1.4 : SLOTs-List Return Example

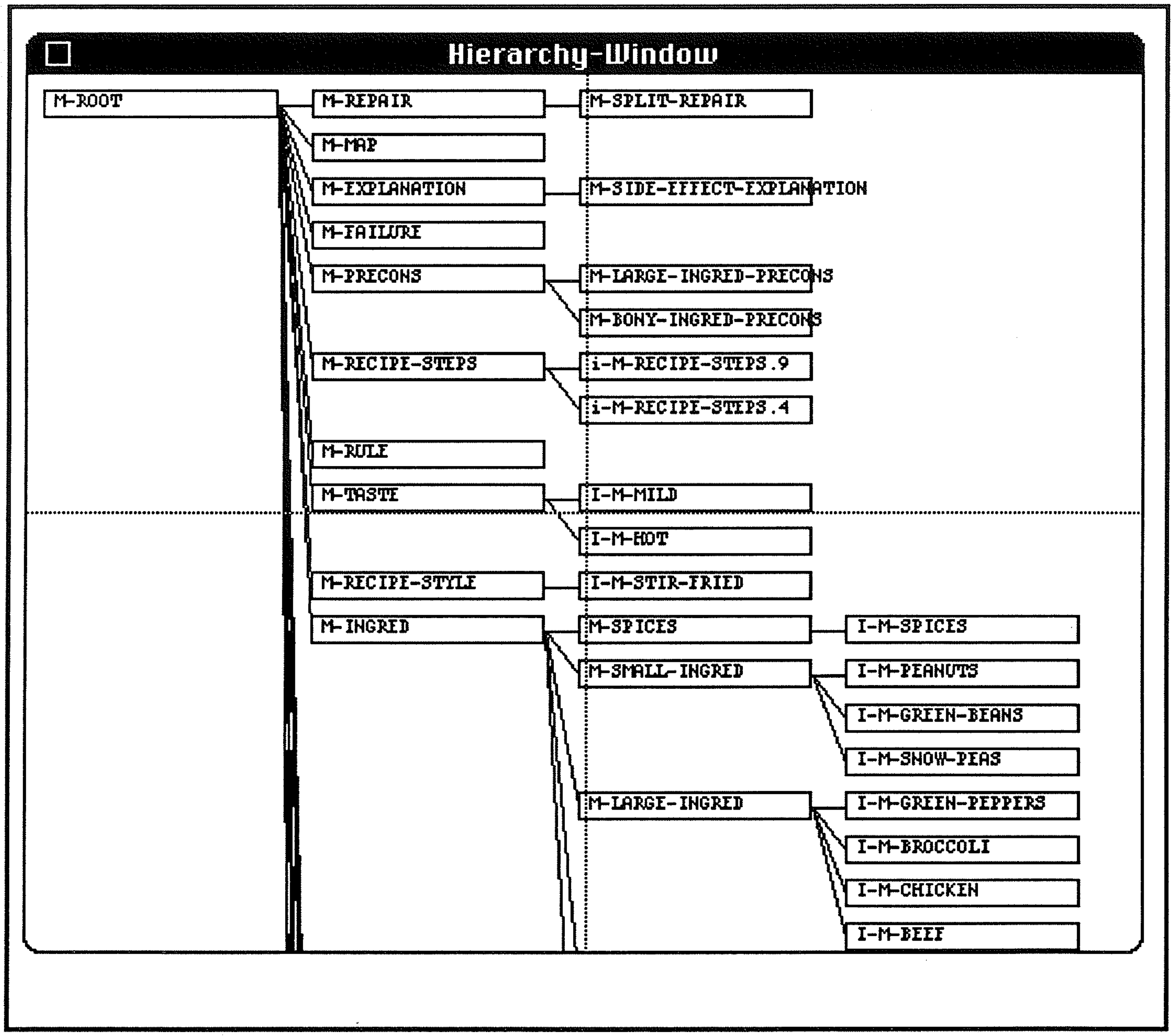
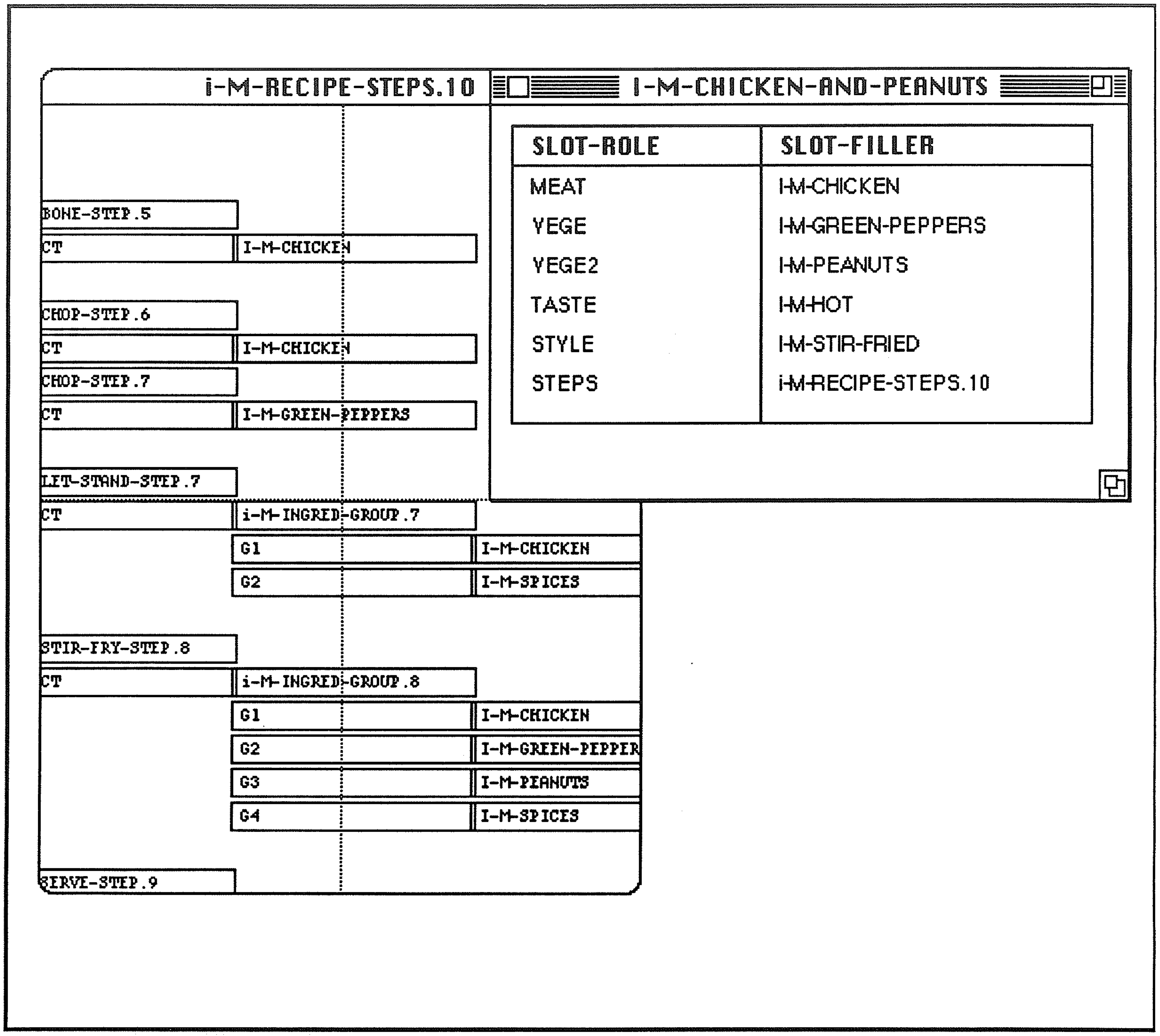


Figure 4.1.5 : MOP-Hierarchy Window



i-M-RECIPE-STEPS.10

BONE-STEP.5

CT	I-M-CHICKEN

CHOP-STEP.6

CT	I-M-CHICKEN

CHOP-STEP.7

CT	I-M-GREEN-PEPPERS

LET-STAND-STEP.7

CT	i-M-INGRED-GROUP.7	
	G1	I-M-CHICKEN
	G2	I-M-SPICES

STIR-FRY-STEP.8

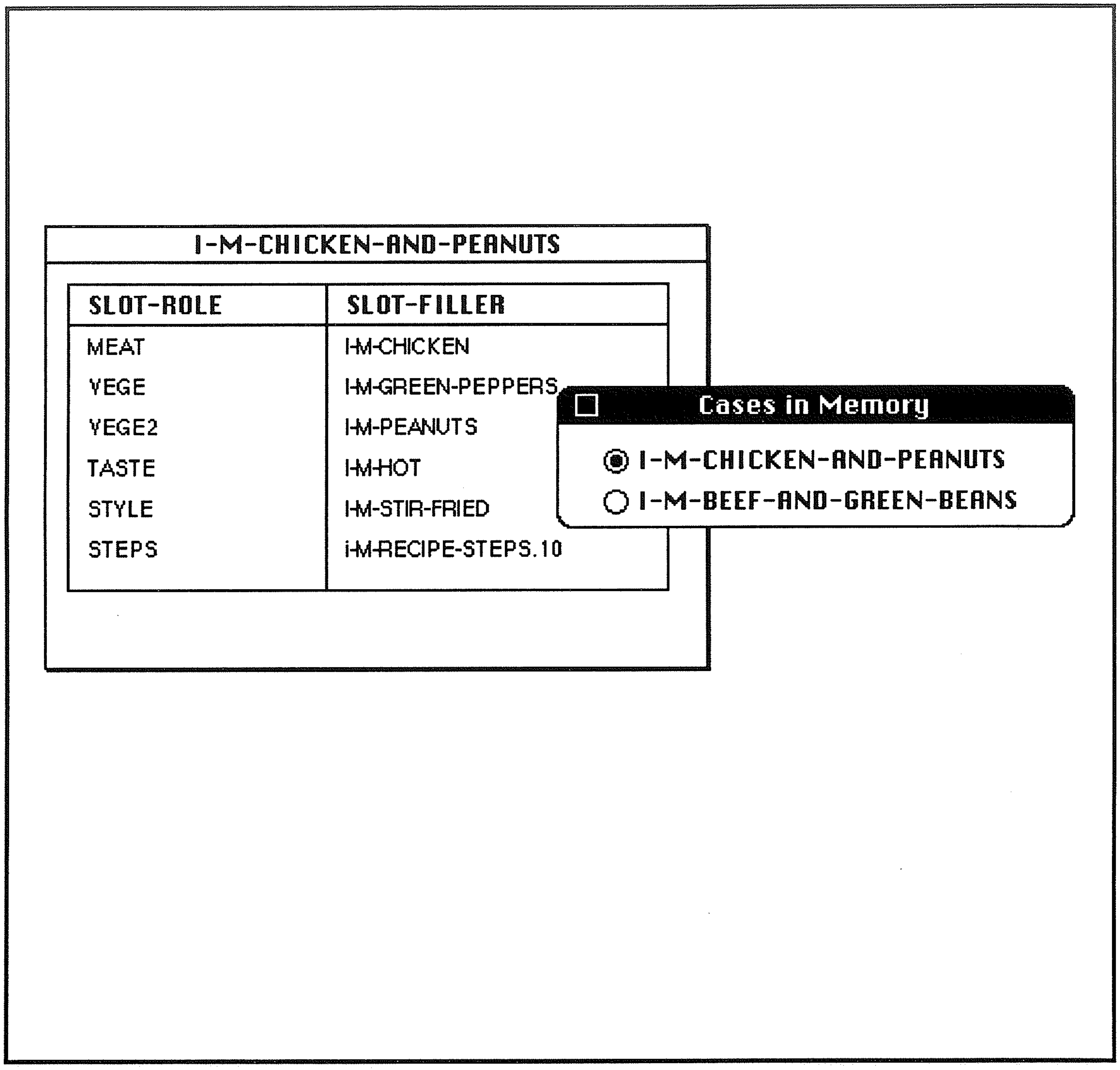
CT	i-M-INGRED-GROUP.8	
	G1	I-M-CHICKEN
	G2	I-M-GREEN-PEPPER
	G3	I-M-PEANUTS
	G4	I-M-SPICES

SERVE-STEP.9

I-M-CHICKEN-AND-PEANUTS

SLOT-ROLE	SLOT-FILLER
MEAT	I-M-CHICKEN
VEGE	I-M-GREEN-PEPPERS
VEGE2	I-M-PEANUTS
TASTE	I-M-HOT
STYLE	I-M-STIR-FRIED
STEPS	i-M-RECIPE-STEPS.10

Figure 4.1.6 : SLOT-Hierarchy Window



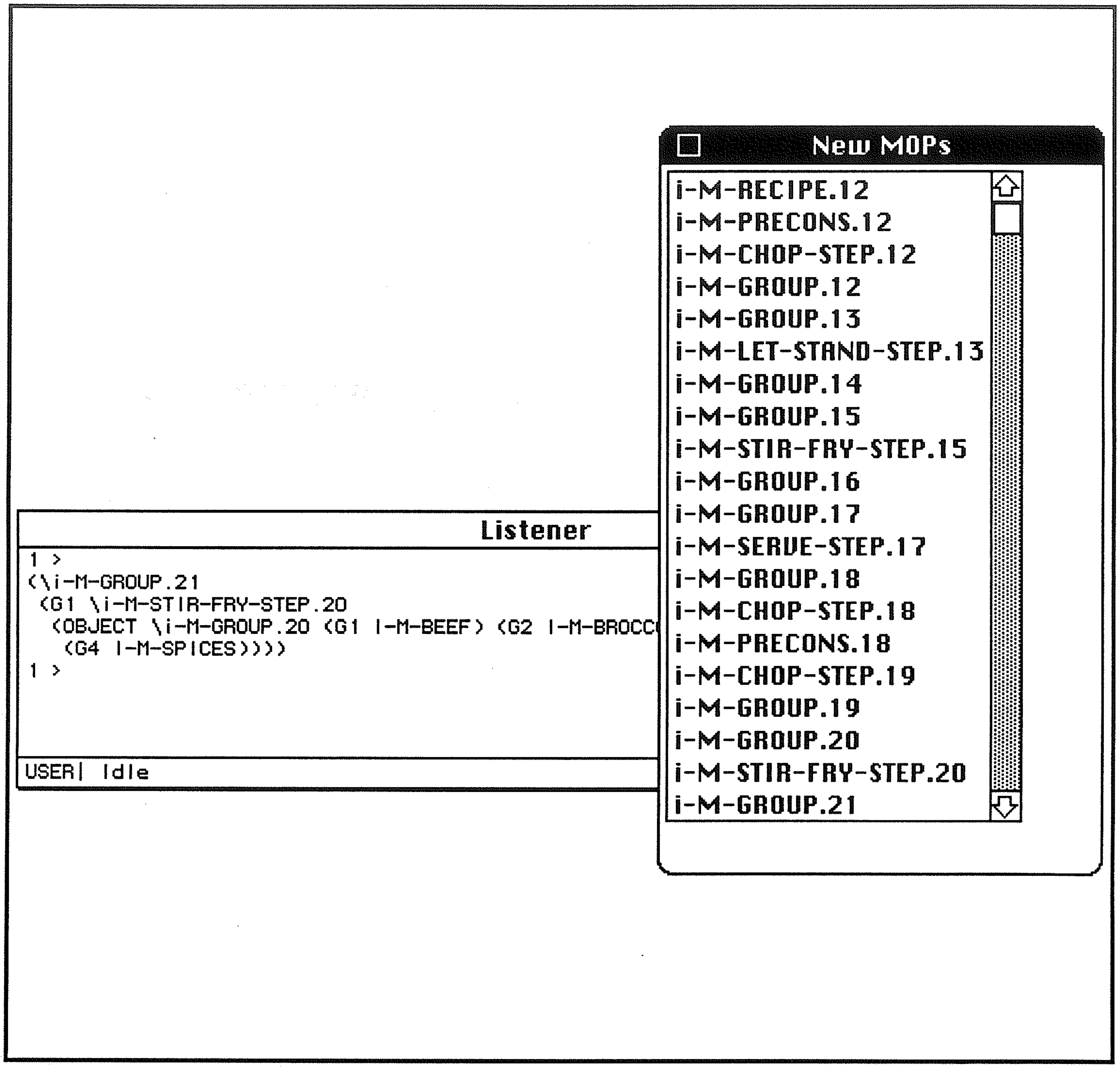
I-M-CHICKEN-AND-PEANUTS

SLOT-ROLE	SLOT-FILLER
MEAT	I-M-CHICKEN
VEGE	I-M-GREEN-PEPPERS
VEGE2	I-M-PEANUTS
TASTE	I-M-HOT
STYLE	I-M-STIR-FRIED
STEPS	i-M-RECIPE-STEPS.10

Cases in Memory

[✓ I-M-CHICKEN-AND-PEANUTS]
I-M-BEEF-AND-GREEN-BEANS

Figure 4.1.7 : Cases Return Example



```
1 >
(\i-M-GROUP.21
 (G1 \i-M-STIR-FRY-STEP.20
  (OBJECT \i-M-GROUP.20 (G1 I-M-BEEF) (G2 I-M-BROCC
   (G4 I-M-SPICES))))
1 >
```

```
USER| Idle
```

```
i-M-RECIPE.12
i-M-PRECONS.12
i-M-CHOP-STEP.12
i-M-GROUP.12
i-M-GROUP.13
i-M-LET-STAND-STEP.13
i-M-GROUP.14
i-M-GROUP.15
i-M-STIR-FRY-STEP.15
i-M-GROUP.16
i-M-GROUP.17
i-M-SERVE-STEP.17
i-M-GROUP.18
i-M-CHOP-STEP.18
i-M-PRECONS.18
i-M-CHOP-STEP.19
i-M-GROUP.19
i-M-GROUP.20
i-M-STIR-FRY-STEP.20
i-M-GROUP.21
```

Figure 4.1.8 : New-MOPs Return Example

4.2. How to Define and Access the MOPs

Figure 4.2.1 shows a simple example of the MOP definitions and instances. The function “clear-memory” in the first statement creates the foundation MOP “m-root”. In Micro MOPs, all MOPs should be defined under “m-root”. Example 1 creates a world with three students and their characteristics.

```
( clear-memory )
( defmop     m-nationality  ( m-root )            mop )
( defmop     i-m-American   ( m-nationality )     instance )
( defmop     i-m-Japanese   ( m-nationality )     instance )
( defmop     i-m-German     ( m-nationality )     instance )
( defmop     m-eye-color    ( m-root )            mop )
( defmop     i-m-blue       ( m-eye-color )               instance )
( defmop     i-m-grey       ( m-eye-color )               instance )
( defmop     i-m-brown      ( m-eye-color )               instance )
( defmop     i-m-green      ( m-eye-color )               instance )
( defmop     i-m-student    ( m-root )            mop )
( defmop     i-m-Bob        ( m-student )         instance
                            ( nationality   i-m-American )
                            ( age           26 )
                            ( height        6.2 )
                            ( eye-color     i-m-grey ))
( defmop     i-m-Mark       ( m-student )         instance
                            ( nationality   i-m-German )
                            ( age           24 )
                            ( height        5.7 )
                            ( eye-color     i-m-green ))
( defmop     i-m-Kazu       ( m-student )         instance
                       ( nationality   i-m-Japanese )
                            ( age           30 )
                            ( height        5.6 )
                            ( eye-color     i-m-brown ))
```

Figure 4.2.1 : MOPs Definition for Example 1

```
( defmop        name mop-list type slot-form1 slot-form2 ... )

        name          : the name of new MOPs
        mop-list      : the name of immediate upper level abstraction  MOPs
        type          : type of MOPs, it should be either "mop" or "instance"
        slot-form     : the form of slots
```

Defmop defines a MOP named *name* as *type* with slots specified as *slot-form* under the abstraction MOPs named *mop-list*.

In Micro MOPs, the words starting with "m-" are used as abstractions and the words starting with "i-m-" are used as instances. Furthermore, "slot" and "slot value" are referred to in a frame system as "role" and "role filler" respectively. A slot in Micro MOPs has the form "(role filler)".

```
( role-filler     role slot-source )

        role          : the name of role
        slot-source   : the list of slots or the name of MOP
```

Role-filler returns the filler of *role* in *slot-source*. For example, if we want to get the filler of nationality-role in i-m-Bob and the filler of eye-color in i-m-Kazu.

```
? ( role-filler   'nationality    'i-m-Bob )
I-M-AMERICAN
? ( role-filler   'eye-color      'i-m-Kazu )
I-M-BROWN
```

```
( add-role-filler        role mop filler )

        role          : the name of role
        mop           : the name of MOP
        filler        : the filler of role
```

Add-role-filler adds the slot (*role filler*) to the slots of *mop*. For example, if we want to add the slot (hair-color i-m-black) to i-m-Kazu, then

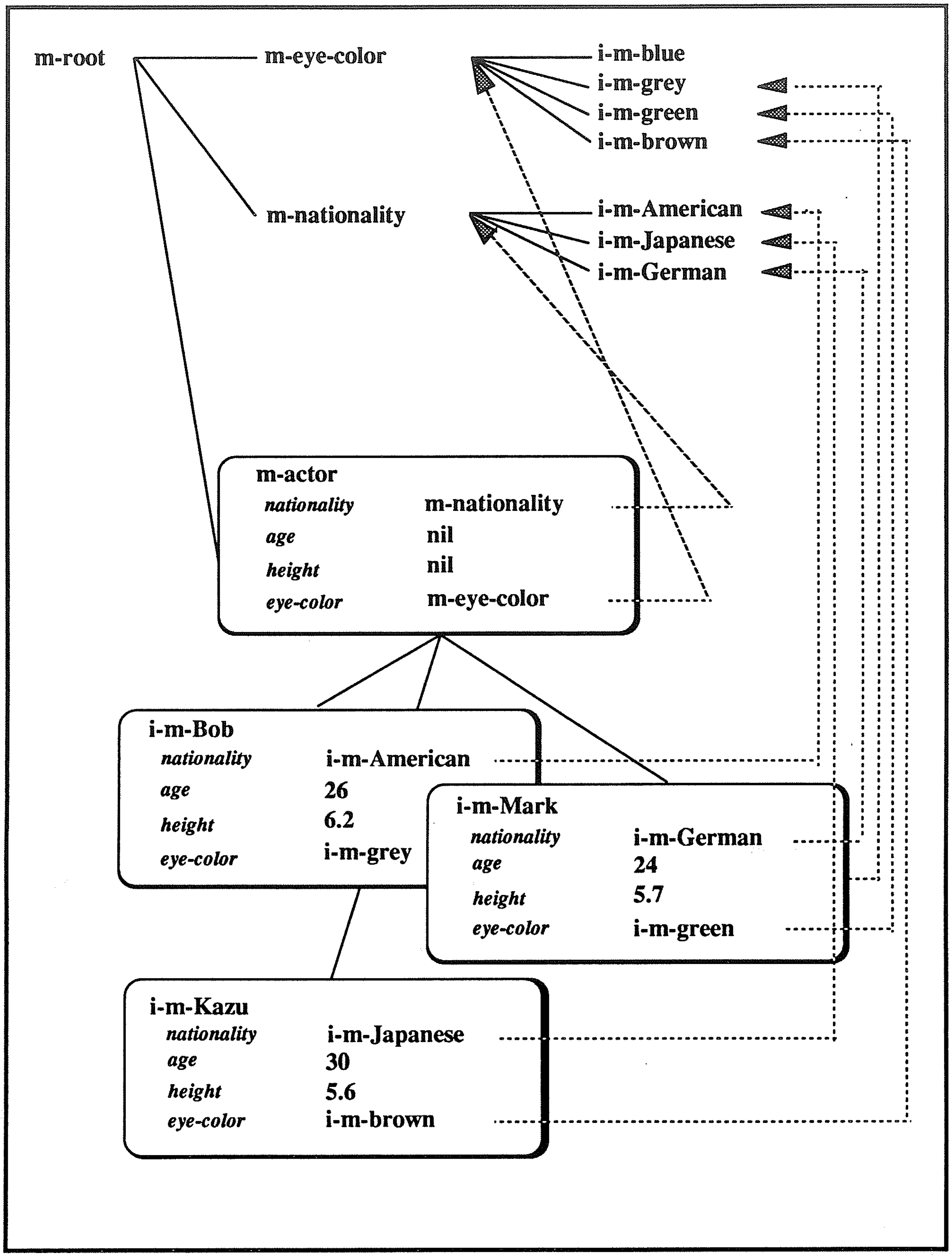


Figure 4.2.2 : Example 1

```
? ( add-role-filler     'hair-color     'i-m-Kazu     'i-m-black )
I-M-KAZU : HAIR-COLOR  <=  I-M-BLACK
I-M-BLACK
```

```
( new-mop     name mop-list  type slot-list)
```

name	: the name of new MOP or instance
mop-list	: the name of immediate abstraction MOPs which you want to link to
type	: either mop or instance
slot-list	: list of slots

New-mop makes a MOP named *name* as its type of *type* with slot of *slots-list* under immediate abstraction MOPs named *mop-list.* For example, if we make a new instance i-m-Jim with some new slots, then

```
? ( new-mop  'i-m-Jim       '( m-student )  'instance
'(( nationality  i-m-American )
  ( age          32 )
  ( height       6.2 )
  ( eye-color    i-m-grey )))
I-M-JIM
```

In Figure 4.2.2, the words except a numerical number used as the filler of role should be defined as either MOP or instance before being referred. Figure 4.2.3 is an "m-root " abstraction hierarchy network graph which is displayed by sub-menu function "**MOP-Hierarchy**", in the case-based reasoning environment main menu "**MOP-Utility**".

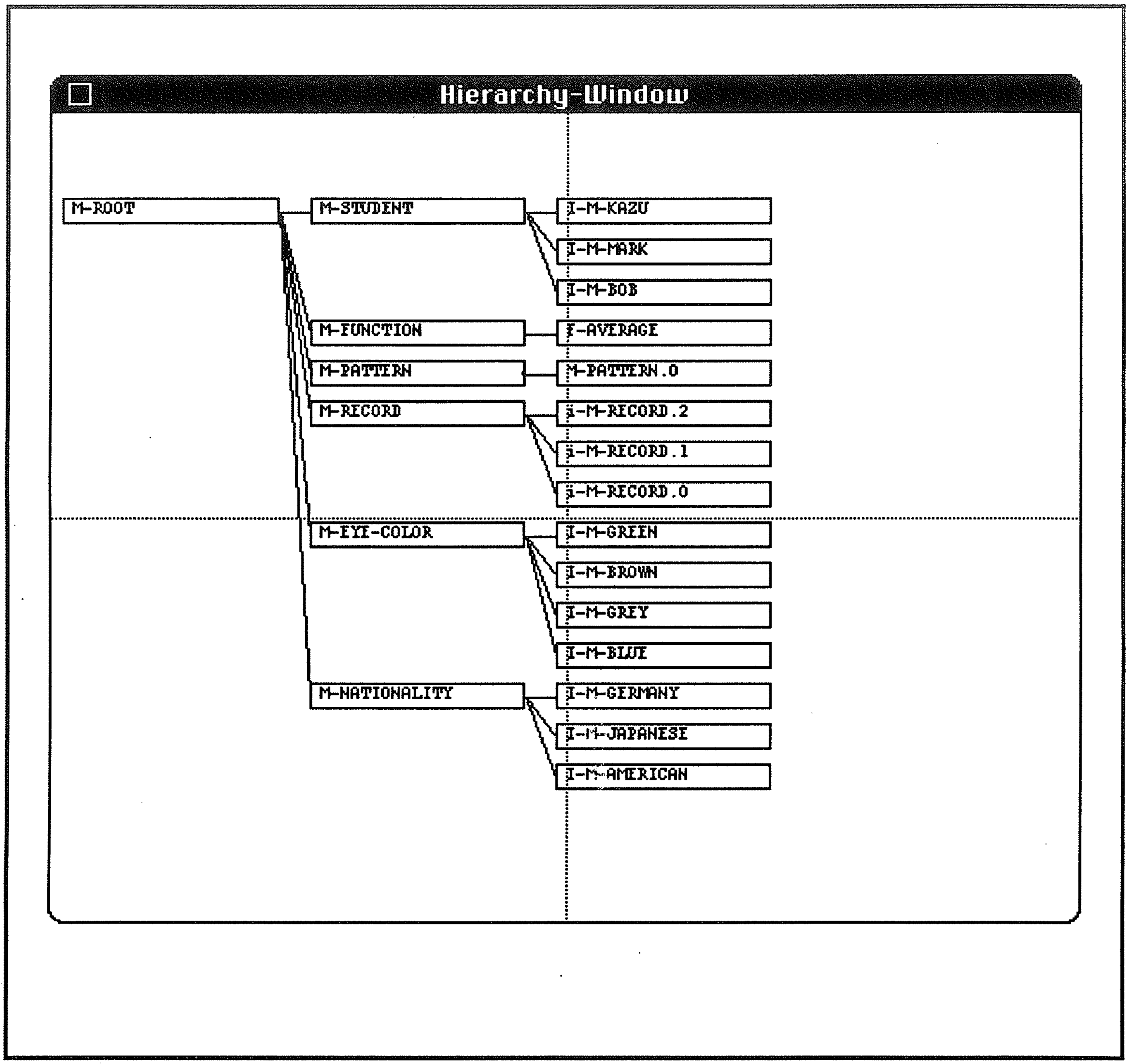


Figure 4.2.3 : MOPs Hierarchy Graph for Example 1

Figures 4.2.4-1 and 4.2.4-2 illustrate additional functions in Micro MOPs. In this example, three students' Math and English scores are added to the previous example.

```
( clear-memory )
( defmop    m-nationality  ( m-root )          mop )
( defmop    i-m-American   ( m-nationality )   instance )
( defmop    i-m-Japanese   ( m-nationality )   instance )
( defmop    i-m-German     ( m-nationality )   instance )
( defmop    m-eye-color    ( m-root )          mop )
( defmop    i-m-blue       ( m-eye-color )             instance )
( defmop    i-m-grey       ( m-eye-color )             instance )
( defmop    i-m-brown      ( m-eye-color )             instance )
( defmop    i-m-green      ( m-eye-color )             instance )
( defmop    m-quarter      ( m-root )          mop )
( defmop    i-m-fall              ( m-quarter )        instance )
( defmop    i-m-winter     ( m-quarter )       instance )
( defmop    i-m-spring     ( m-quarter )       instance )
( defmop    i-m-summer     ( m-quarter )       instance )
( defmop    m-record       ( m-root )          mop )
                           ( Math        nil )
                           ( English     nil )
                           ( year        1990 )
                           ( quarter     i-m-fall ))
( defmop    m-pattern      ( m-root )          mop )
( defmop    m-function     ( m-root )          mop )
( defmop    f-average      ( f-function ))
( defmop    i-m-student    ( m-root )          mop
                           ( nationality m-nationality )
                           ( age         nil )
                           ( height      nil )
                           ( eye-color   m-eye-color )
                           ( record      m-record )
                           ( average     m-pattern
                                        ( calc-fn       f-average )))
( defmop    i-m-Bob        ( m-student )       instance
                           ( nationality i-m-American )
                           ( age         26 )
                           ( height      6.2 )
                           ( eye-color   i-m-grey )
                           ( record      m-record
                                         ( Math         70)
                                         ( English      86)))
```

Figure 4.2.4-1 : MOPs Definitions for Example 2

```
( defmop     i-m-Mark     ( m-student )         instance
                          ( nationality   i-m-German )
                          ( age           24 )
                          ( height        5.7 )
                          ( eye-color     i-m-green )
                          ( record        m-record
                                          ( Math          85)
                                          ( English       77)))
( defmop     i-m-Kazu     ( m-student )         instance
                    ( nationality   i-m-Japanese )
                          ( age           30 )
                          ( height        5.6 )
                          ( eye-color     i-m-brown )
                          ( record        m-record
                                                  ( Math          98)
                                                  ( English       52)))
;
( defun f-average       ( pattern mop )
        ( declare       ( ignore pattern ))
        ( let    (( math-score  ( path-filler '( record Math )    mop ))
                  ( english-score  ( path-filler '( record English ) mop )))
                 ( * 0.5 ( + math-score english-score ))))
```

Figure 4.2.4-2 : MOPs Definitions for Example 2 (continued)

In Figure 4.2.4, the slot whose role named "record" in every student's MOP has the unique filler that has the form of "i-m-record.*i*". In Figures 4.2.4-1 and 4.2.4-2, the slot "record" has the different filler from the other's: this slot has the network structure which consists of MOP and its slots. In this case, the system automatically generates instances with the slots under MOP, attached unique number to the tail of their instance names to distinguish respectively.

(path-filler   *path mop*)

path	: the list of roles to search
mop	: the name of MOP

Path-filler returns the filler for *path* in *mop*. For example, to retrieve Bob's Math score, then

```
? ( path-filler   '( record Math )              'i-m-Bob )
70
```

```
( inherit-filler  role mop )

        role            : the name of role to get
        mop             : the name of MOP
```

Inherit-filler returns the first filler of *role* found in either *mop* or one of its abstraction MOP. For example, to retrieve the year of Bob's record, then

```
? ( inherit-filler      'year        ( role-filler     'record 'i-m-Bob ))
1990
```

"(role-filler 'record 'i-m-Bob)" returns "i-m-record.0", despite the fact that "i-m-record.0" doesn't have role "year" and its filler. Otherwise, "1990" can be returned after inheriting "m-record" from its immediate abstraction MOP.

```
( get-filler     role mop )

        role            : the name of role to get
        mop             : the name of MOP
```

Get-filler returns the filler of the *role* in *mop*. The filler is either an inherited instance or the value calculated by the function. The filler is calculated when a role inherits a pattern MOP with a slot "(calc-fn *function*)". For example, to retrieve Bob's average examination score, then

```
? ( get-filler   'average      'i-m-Bob )
I-M-BOB      : AVERAGE <= 78.0
78.0
? ( get-filler   'average      'i-m-Mark )
I-M-MARK     : AVERAGE <= 75.0
75.0
? ( get-filler   'average      'i-m-Kazu )
I-M-KAZU     : AVERAGE <= 81.0
81.0
```

Figure 4.2.5 shows a case memory after the "get-filler" function is executed. Figure 4.2.6 shows a SLOT-Hierarchy graph network which is displayed by the "**SLOT-Hierachy**" submenu function. The graph network shows slot "(average 78.0)" is attached to i-m-Bob.

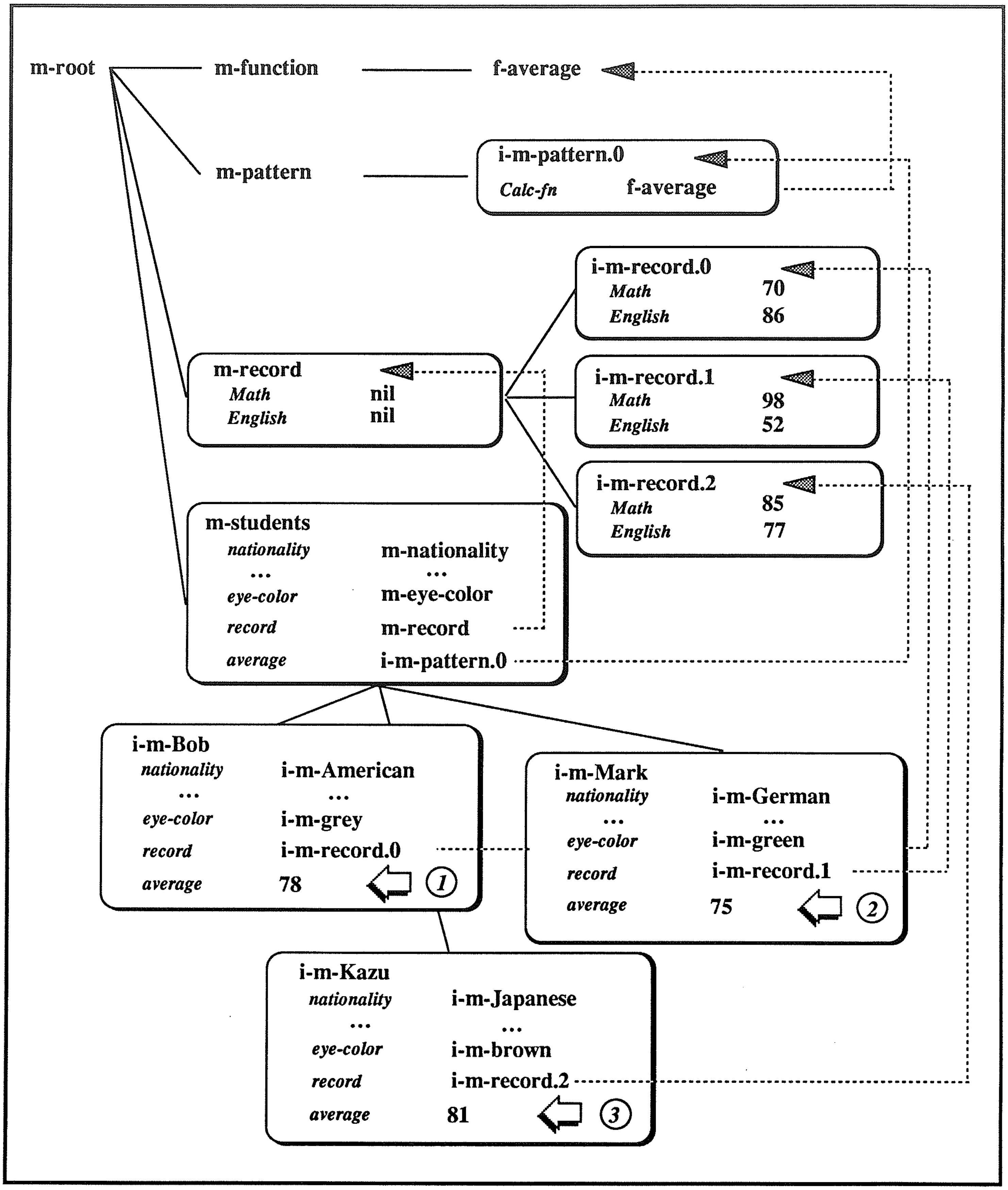
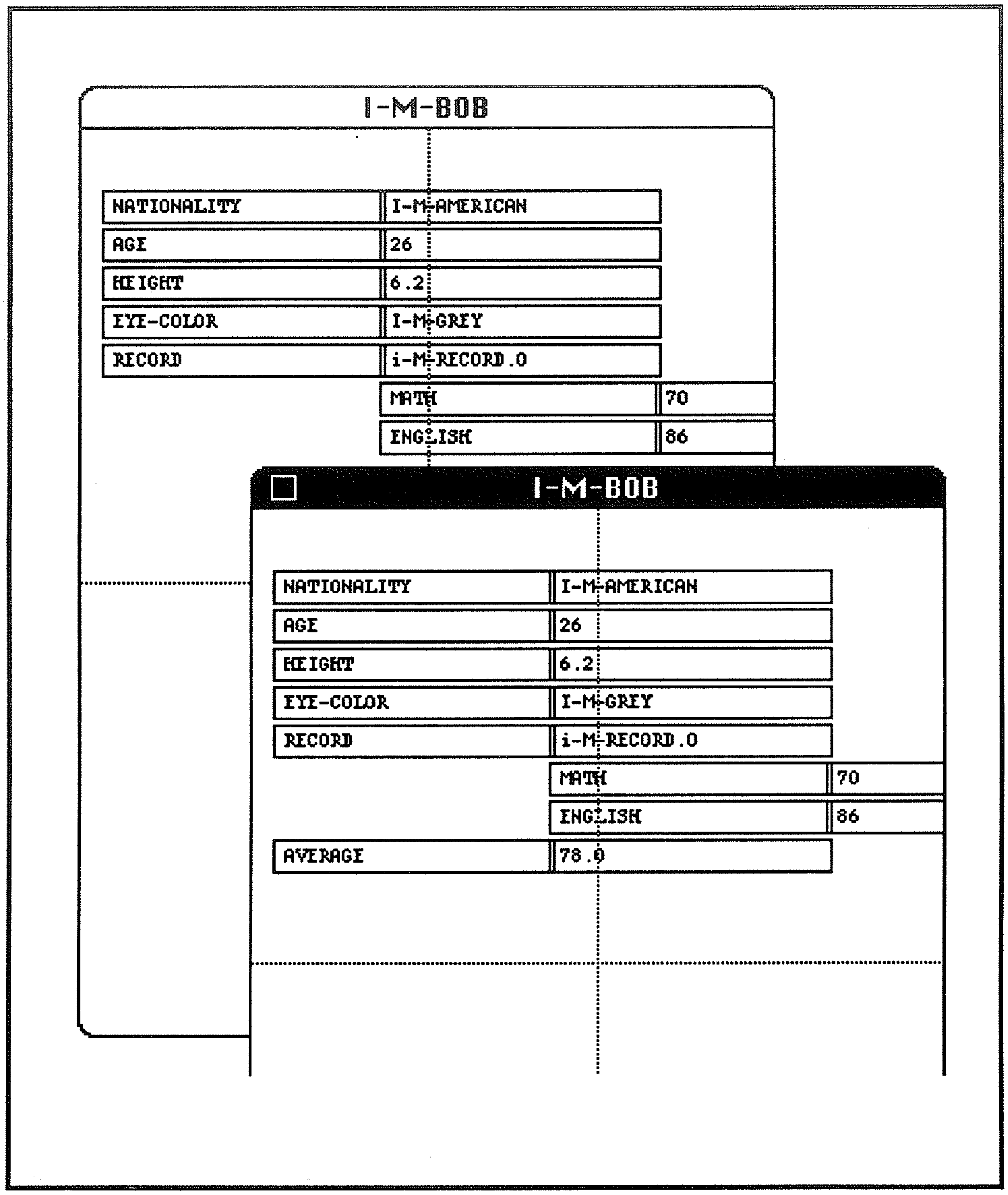


Figure 4.2.5 : "get-filler" at Example 2



I-M-BOB

NATIONALITY	I-M-AMERICAN	
AGE	26	
HEIGHT	6.2	
EYE-COLOR	I-M-GREY	
RECORD	i-M-RECORD.0	
	MATH	70
	ENGLISH	86

I-M-BOB

NATIONALITY	I-M-AMERICAN	
AGE	26	
HEIGHT	6.2	
EYE-COLOR	I-M-GREY	
RECORD	i-M-RECORD.0	
	MATH	70
	ENGLISH	86
AVERAGE	78.0	

Figure 4.2.6 : SLOT-Hierarchy Graph for Example 2

5. Case-based Reasoning Approach to A Residential House Design

5.1. System Architecture

As I introduced in chapter 2, researchers have applied case-based reasoning techniques to various domains. In this chapter, I will introduce an effort to apply this technique to the residential house design domain. Figure 5.1.1 shows the system architecture and its data flow. This prototype system consists of five parts : **Layout CAD, Case-based Reasoning Platform, Function Library, Case Memory,** and **Design Knowledge Database.**

• Layout CAD

Layout CAD has three major features. The first one is the **Design Tool Box.** The **Design Tool Box** has eleven designing functions as shown in Figure 5.1.2.

(a) is a tool for shrinking a room size.

(b) is a tool for extending a room size.

(c) is a tool for rotating a room location.

(d) is a tool for extending all rooms.

(e) is a tool for shrinking all rooms.

(f) is a tool for eliminating a room.

(g) is a tool for drawing an area plan bubble diagram.

(h) is a tool for rotating a house orientation.

(i) is a tool for changing a room adjacency.

(j) is a tool for drawing a building area.

(k) is a tool for attaching a new room.

The **Design Tool Box** is used when a designer inputs an area plan with bubble diagrams at the planning stage, and when the designer accumulates knowledge about using design techniques to fit the new case in **Case Memory** and **Design Knowledge Database.**

• Case-based Reasoning Platform

At **Case-based Reasoning Platform Case-based Reasoner** works as a supervisor. It watches the current situation in the system and decides which one of reasoning functions, **Retriever, Adapter, Evaluater, Storer** and **Layout Output,** should receive control corresponding to the situation.

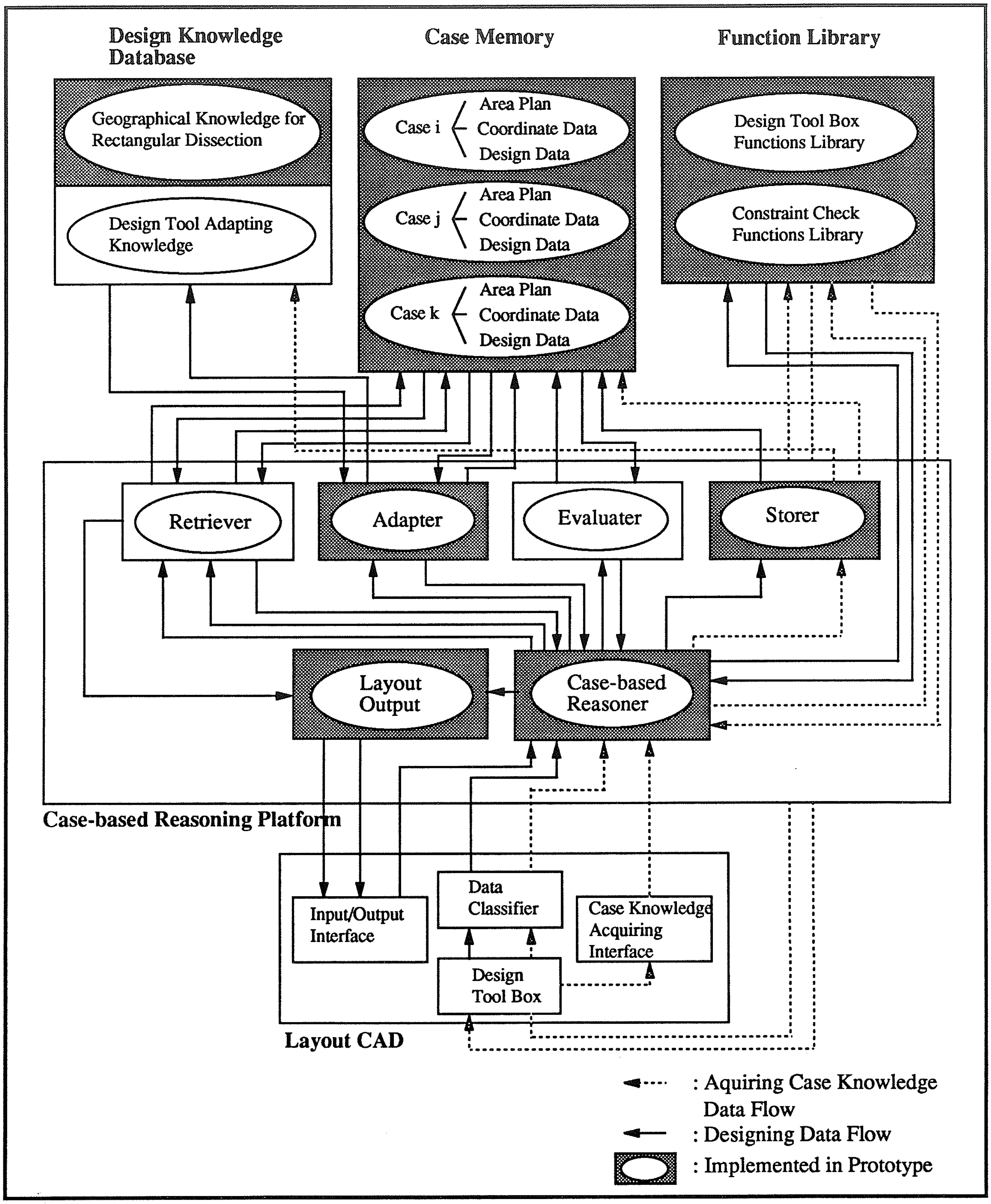


Figure 5.1.1 : System Architecture

• **Function Library**

Function Library is the library consisting of the functions necessary for checking design constraints such as a room size, sun light, noise, and privacy. It works in cooperation with the **Case-based Reasoner** to adapt stored design procedures and with the **Design Tool Box** to acquire case knowledge.

• **Case Memory**

Case Memory deals with two type of cases, the area plan bubble diagram and design procedure. The bubble diagram contains the area adjacency information, and is stored as an adjacency matrix. First, **Retriever** searches **Case Memory** with the area plan as a search index. Another is the design procedure. The design procedure is a set of methods which are applied when changing a room layout, e.g. shrinking all the rooms or eliminating unnecessary room. This kind of data is collected as a part of **Design Data** in Figure 5.1.1. Furthermore, **Design Data** contains room design attributes, *e.g.* room length, room width, and room name. **Coordinate Data** is a set of x-y coordinate values of rectangular room vertexes. **Coordinate Data** is necessary when **Layout Output** draws a rectangular dissection room layout on the screen.

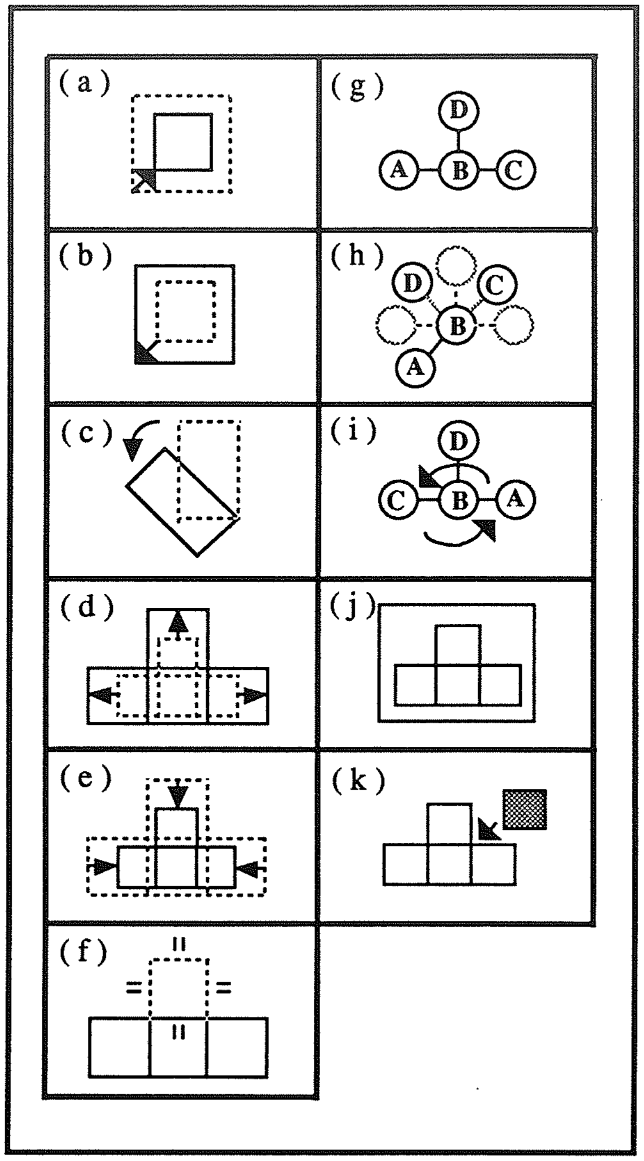


Figure 5.1.2 : Design Tool Box

• **Design Knowledge Database**

Design Knowledge Database consists of **Geographical knowledge for Rectangular Dissection** and **Design Tool Adapting Knowledge.** The former is the Euclidean relationship which comes from the rectangular room adjacencies. The latter is the knowledge about the adapting technique used in Design Tools. This is the information about when and how each tool was used, or which combination of tools was efficient to resolve this difficulty.

Figure 5.1.2 shows a data flow of the first retrieving stage. At this stage, the user enters the area plan by means of a bubble diagram. After that, **Retriever** returns some alternatives corresponding to the entered area plan from **Case Memory**. Finally, **Layout Output** shows these alternatives graphically on the screen.

Figure 5.1.3 shows a data flow of the second retrieving stage. At this stage, the user chooses one of the alternatives, enters the building area by the **Design Tool Box**, answers some overall questionnaires and prioritizes the design requirements. First, all constraints are checked with the functions in the library. Next, the difference between the new case and the case picked from the alternatives is passed to **Case-based Reasoner**. Thus, at this stage, the case is indexed with a list of satisfied constraints and unsatisfied constraints. **Retriever** searches **Case Memory** and then returns the closest layout case. If **Retriever** finds a matching case, **Adapter** adapts the design process of the retrieved case to the new case. If **Retriever** finds an analogous case, then **Case-based Reasoner** searches the resolution which can repair the difference between the new and retrieved cases. In each case, evaluator evaluates the result. Finally, **Layout Output** draws the graphical layout which is customized by **Adapter**, and **Storer** stores it in **Case Memory**.

Figure 5.1.4 shows a data flow of acquiring knowledge about the case customization process. The user can accumulate the customizing process with the **Design Tool Box**.

Due to the limited development time, a part of the CBRE has not been developed. The example introduced in the next chapter illustrates the current system capabilities.

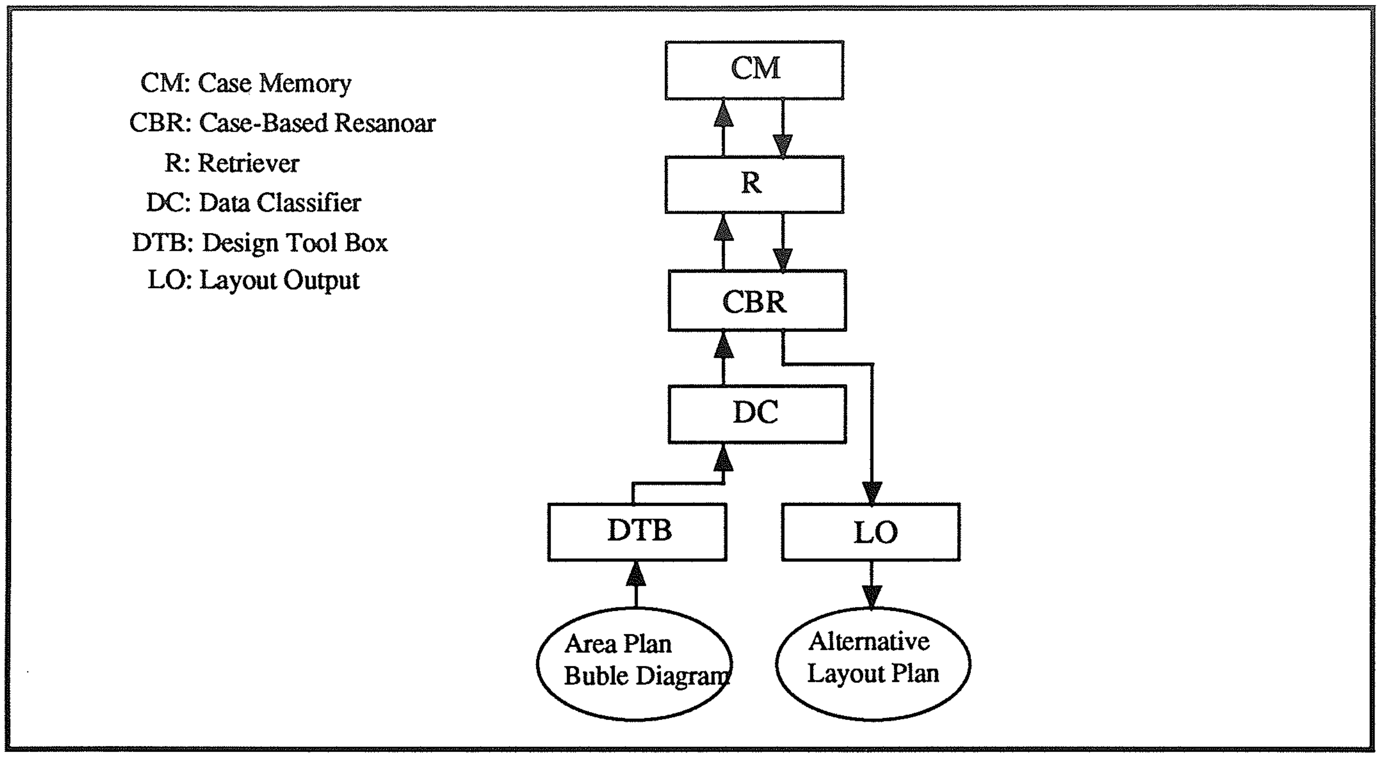


Figure 5.1.3 : Data Flow Stage 1

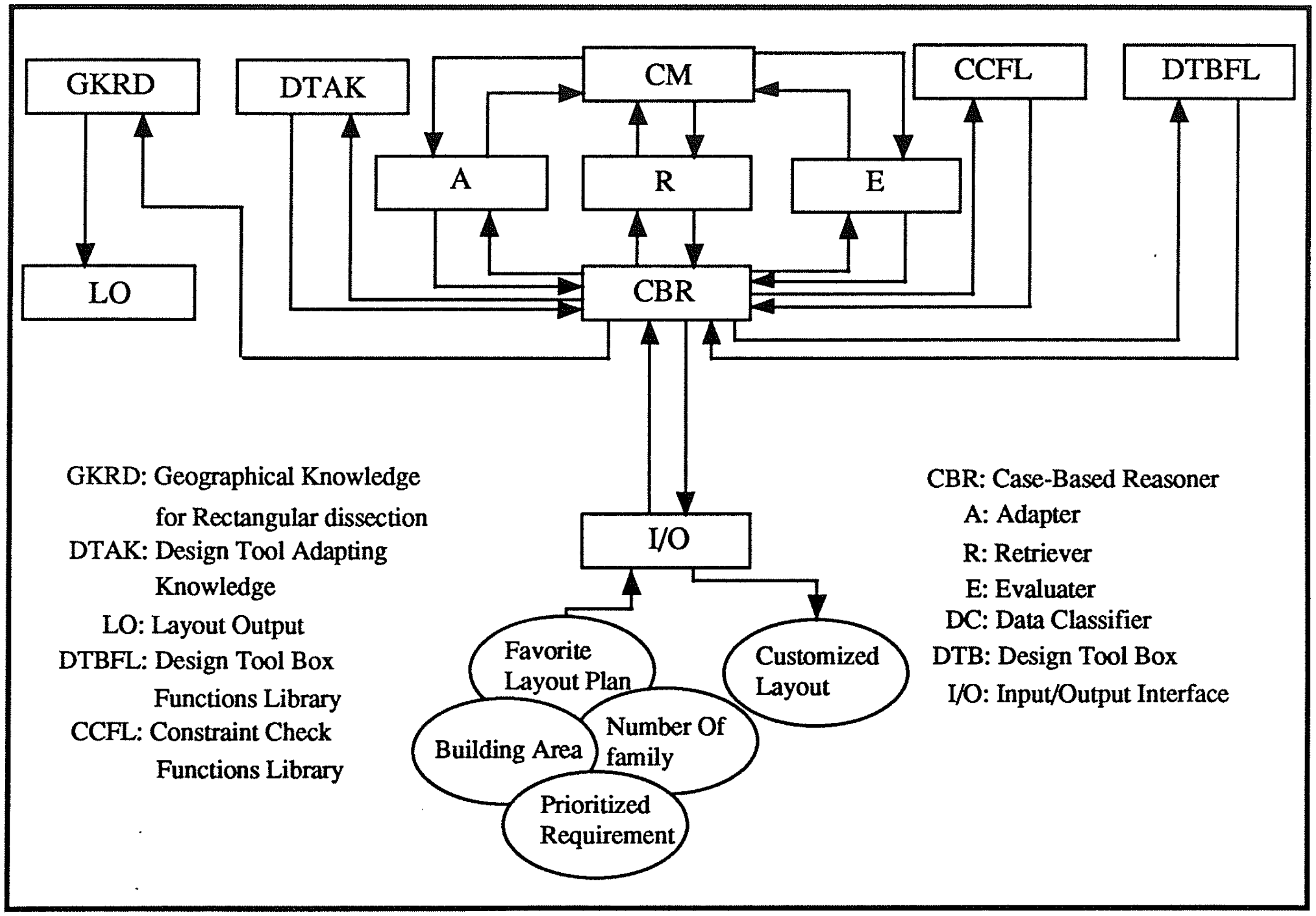


Figure 5.1.4 : Data Flow Stage 2

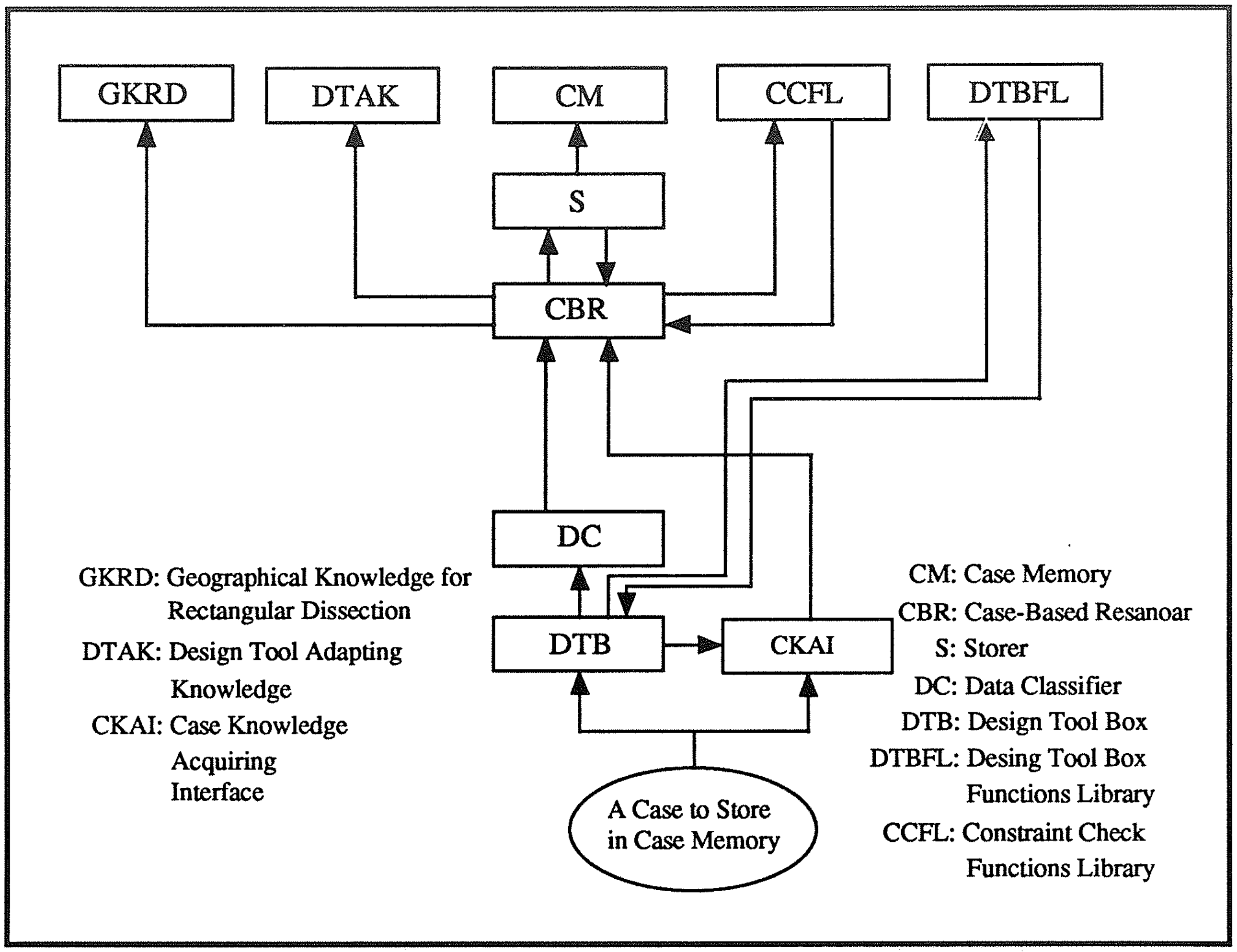

GKRD
DTAK
CM
CCFL
DTBFL
S
CBR
DC
DTB
CKAI
A Case to Store in Case Memory
GKRD: Geographical Knowledge for Rectangular Dissection
DTAK: Design Tool Adapting Knowledge
CKAI: Case Knowledge Acquiring Interface
CM: Case Memory
CBR: Case-Based Resanoar
S: Storer
DC: Data Classifier
DTB: Design Tool Box
DTBFL: Desing Tool Box Functions Library
CCFL: Constraint Check Functions Library

Figure 5.1.5 : Data Flow Stage 3

5.2. A Base Model Design

In this case, a designer's learning process for residential house design is implemented with the case-based reasoning technique. People who want to build their own house often refer to previous examples. In this case, a typical example from "Home Planner's Guide to Residential Design" (Talcott, Hepler and Wallach 86) is selected as a base model case (Figure 5.2.1).

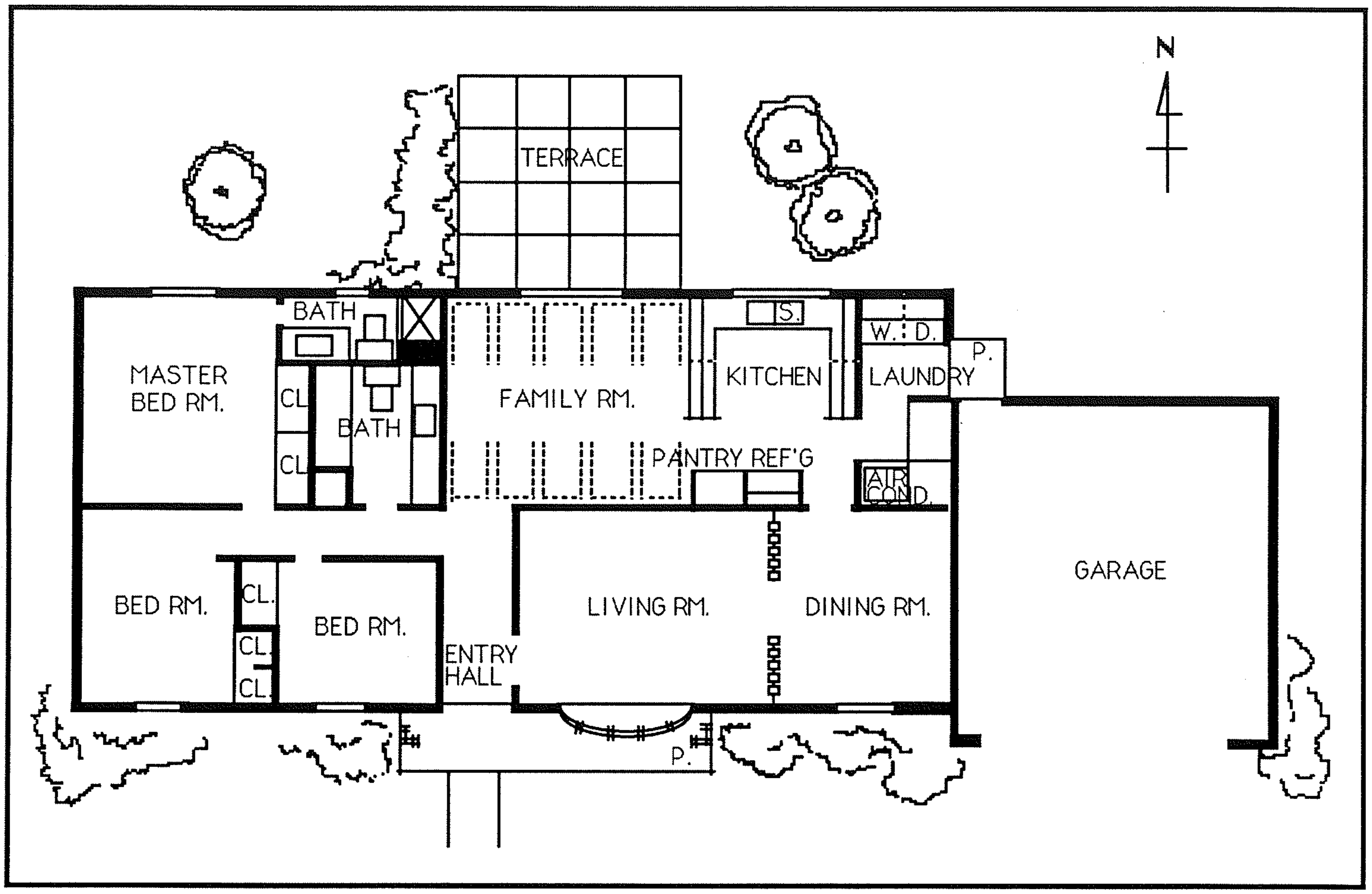


Figure 5.2.1 : A Base Model Residantial House Design

A residence is divided into three major functional areas for planning purposes, i. e. the sleeping area, the living area, and the service area. In this base model case, these three major functional areas are located in the west side zone, the central zone, and the east side zone respectively.

From the designer's point of view, this base model case has several excellent points in terms of residential design. These are :

- The main entrance hall provides direct access to the sleeping area and the living area.

- The main entrance is readily identifiable and provides shelter to anyone awaiting entrance.

- Since the sleeping area is located on the north and west sides, it contains the greatest amount of darkness in the morning and provides the greatest degree of thermal comfort.

- The natural alignment of the living room, dining room, kitchen, family room, and entrance hall provides a circular living-area traffic pattern.

- Since the kitchen is directly adjacent to the family room , the layout provides visual access to children in the family room.

- The family room is not visible from the living area.
  (The family room is cluttered because of the normal accumulation of hobby materials.)

- The sleeping area is located far from the garage. The living area should shut out car noise from the garage.

To simplify the problem, only rectangular shaped rooms are dealt with in this case. Therefore, the sample layout becomes a simple rectangular dissection shown as Figure 5.2.2. Also, to simplify representation, the width and length of the rooms are represented as shown in Figure 5.2.2.

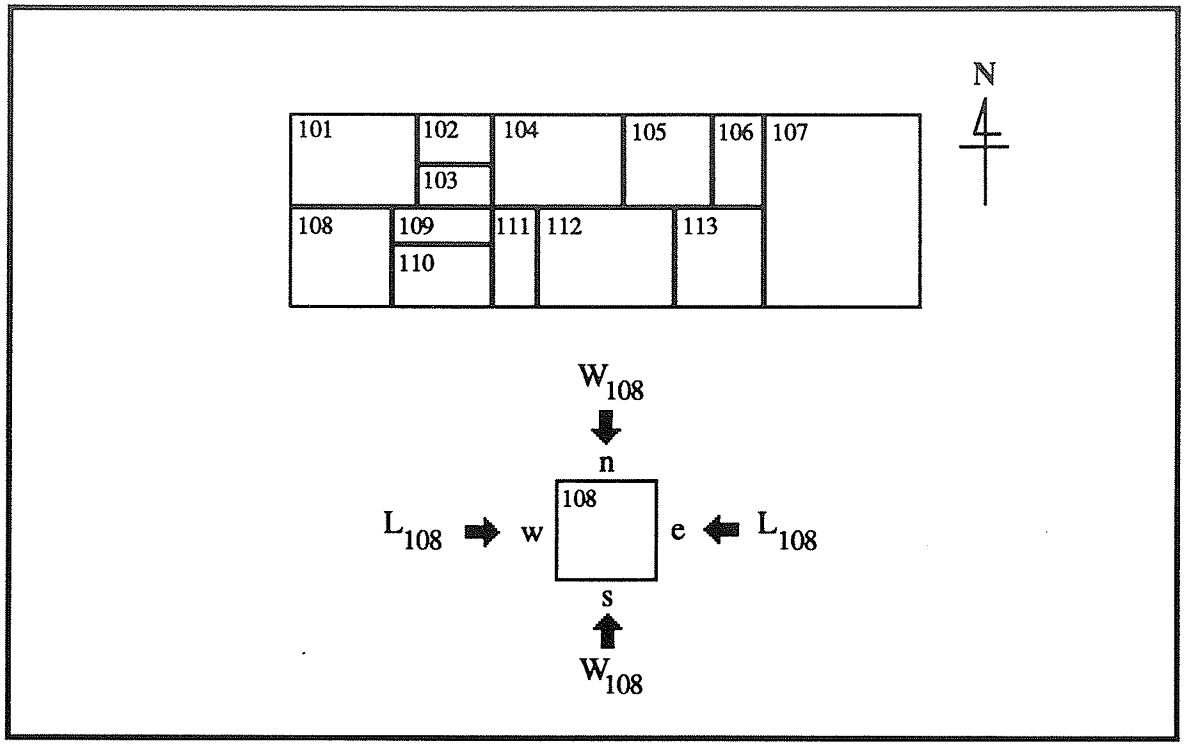


Figure 5.2.2 : Space Number and Room Shape Representation

5.3. The Definitions for The Knowledge in the Design Domain

Figures 5.3.1-1 to 5.3.1-3 show the abstract hierarchy network graph of the base MOP "m-root" in this layout problem. The "M-root" consists of two major types of knowledge, *i. e.* reasoning knowledge and geographical knowledge. Figures 5.3.2-1 to 5.3.2-3 show the instance definition of MOP "i-m-layout001". Instance "i-m-layout001" comprises the attributes and values of each room in the sample residential house shown in Figure 5.2.2. Figure 5.3.3 shows the abstract hierarchy network of MOP "m-space". The MOP "m-space" has four specific MOPs: "m-sleeping-area", "m-living-area", "m-service-area", and "m-traffic-area". Furthermore, each MOP has its own instances. The MOP "m-space" represents the knowledge about the attributes of each room, the relationships between the functional zones and each room. E.g. "space109" is a bedroom with size 9.6' x 11.5' for one person and belongs to the sleeping area.

Figure 5.3.4 shows the abstract hierarchy network of MOP "m-orientation". The MOP "m-orientation" has four specific MOPs: "m-north-orient", "m-east-orient", "m-south-orient", and "m-west-orient". This MOP represents the orientation knowledge of the house. E.g. the north side of the house consists of six rooms: "space101", "space102", "space104", "space105", "space106", and "space107".

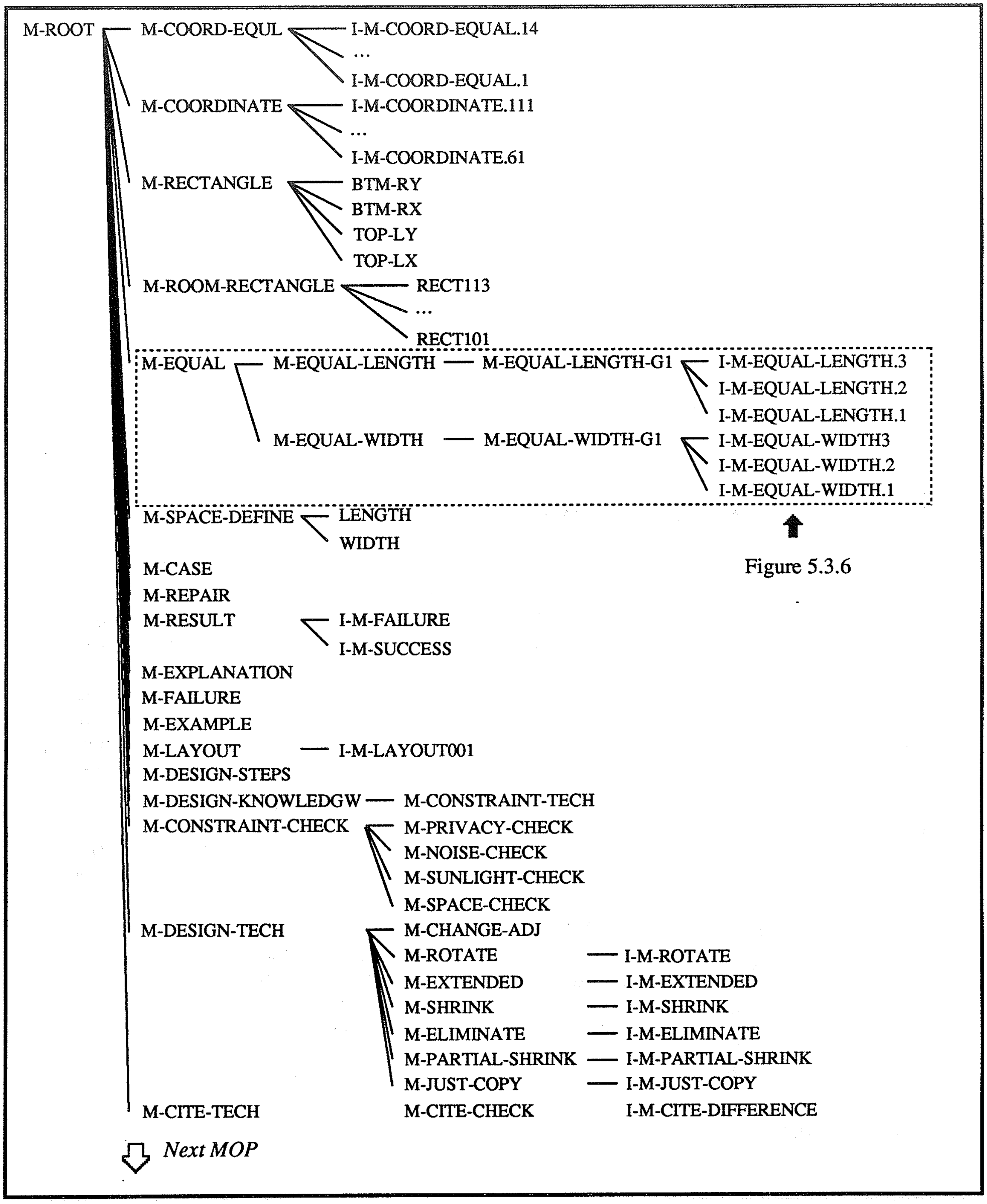


Figure 5.3.1-1 : MOP "m-root"

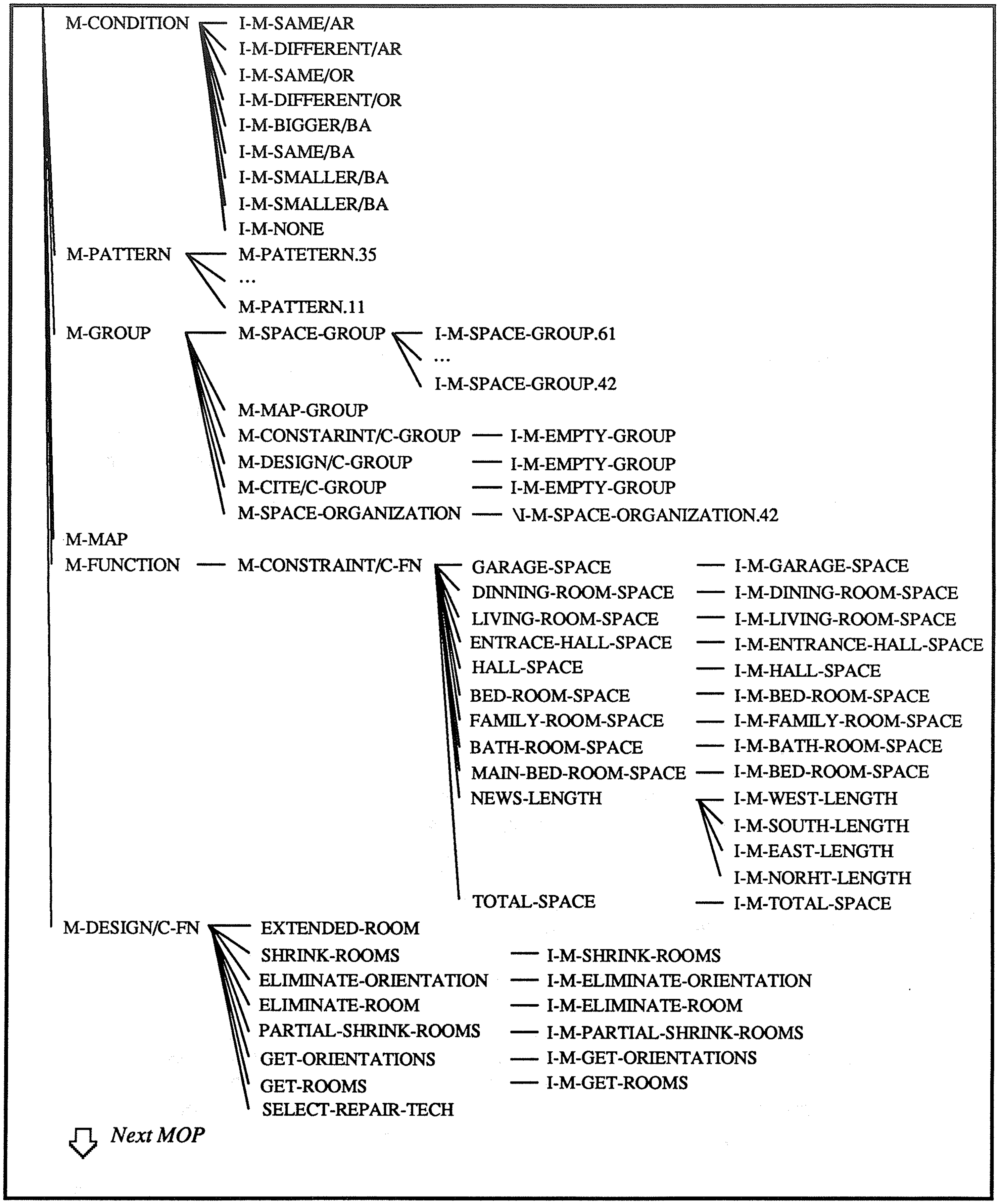


Figure 5.3.1-2 : MOP "m--root" (continued)

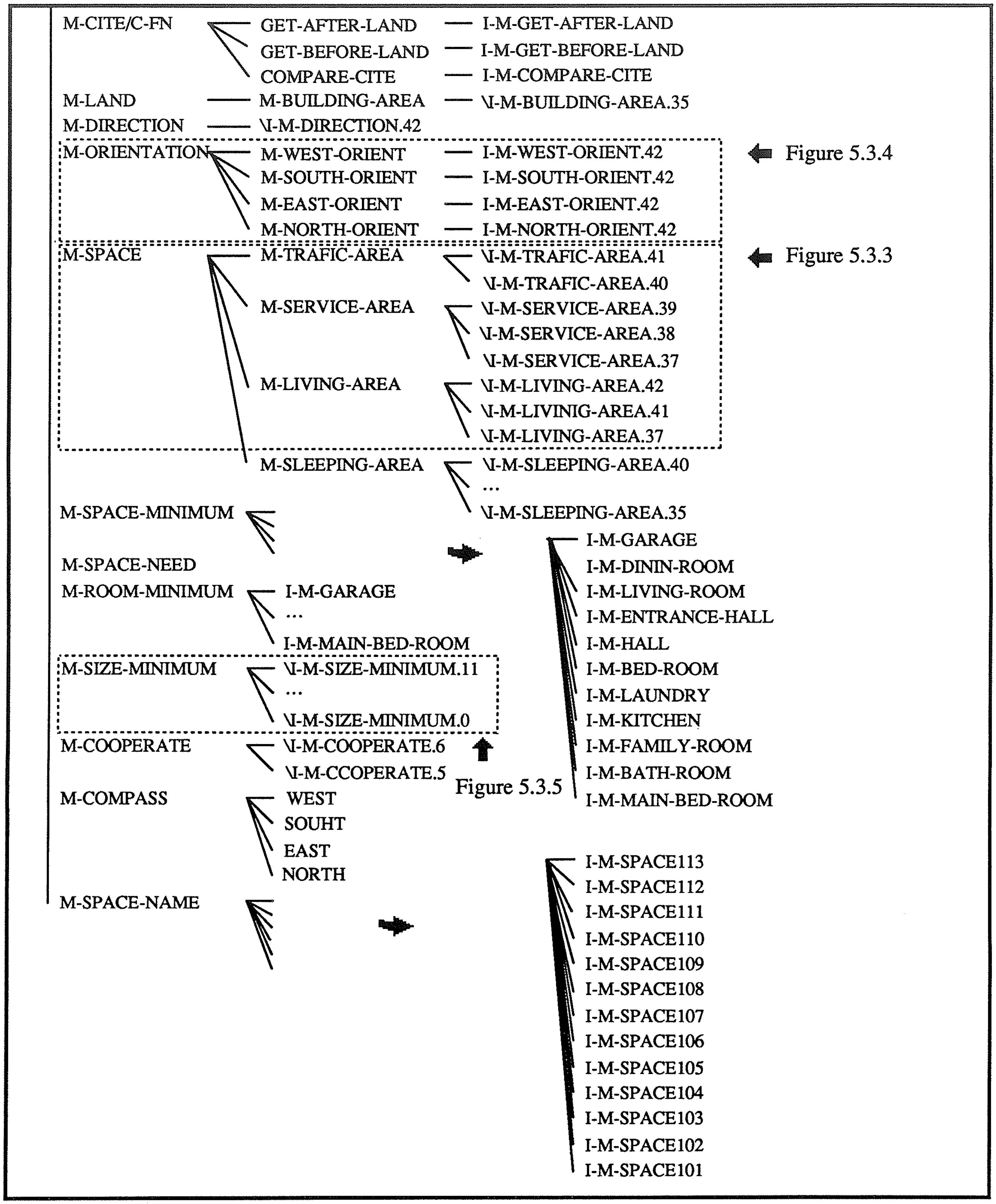


Figure 5.3.1-3 : MOP "m--root" (continued)

```
( defmop    i-m-layout001       ( m-layout )
            ( building-area     m-building-area
                                ( width 76.0 )
                                ( length       25.0 ))
            ( room              m-space-organization
                                ( g1    m-sleeping-area
                                        ( space-name   i-m-space101 )
                                        ( width        15.0 )
                                        ( length       12.0 )
                                        ( no-of-people  2 )
                                        ( use          i-m-main-bed-room ))
                                ( g2    m-sleeping-area
                                        ( space-name   i-m-space102 )
                                        ( width        8.0 )
                                        ( length       6.0 )
                                        ( no-of-people 1 )
                                        ( use          i-m-bath-room ))
                                ( g3    m-sleeping-area
                                        ( space-name   i-m-space103 )
                                        ( width        8.0 )
                                        ( length       6.0 )
                                        ( no-of-people 1 )
                                        ( use          i-m-bath-room ))
                                ( g4    m-living-area
                                        ( space-name   i-m-space104 )
                                        ( width        17.0 )
                                        ( length       12.0 )
                                        ( no-of-people 8 )
                                        ( use          i-m-family-room ))
                                ( g5    m-service-area
                                        ( space-name i-m-space105 )
                                        ( width        9.0 )
                                        ( length       12.0 )
                                        ( use          i-m-kitchen ))
                                ( g6    m-service-area
                                        ( space-name   i-m-space106 )
                                        ( width        7.0 )
                                        ( length       12.0 )
                                        ( use          i-m-laundry ))
                                ( g7    m-service-area
                                        ( space-name   i-m-space107 )
                                        ( width        20.0 )
                                        ( length       25.0 )
                                        ( use          i-m-garage ))
```

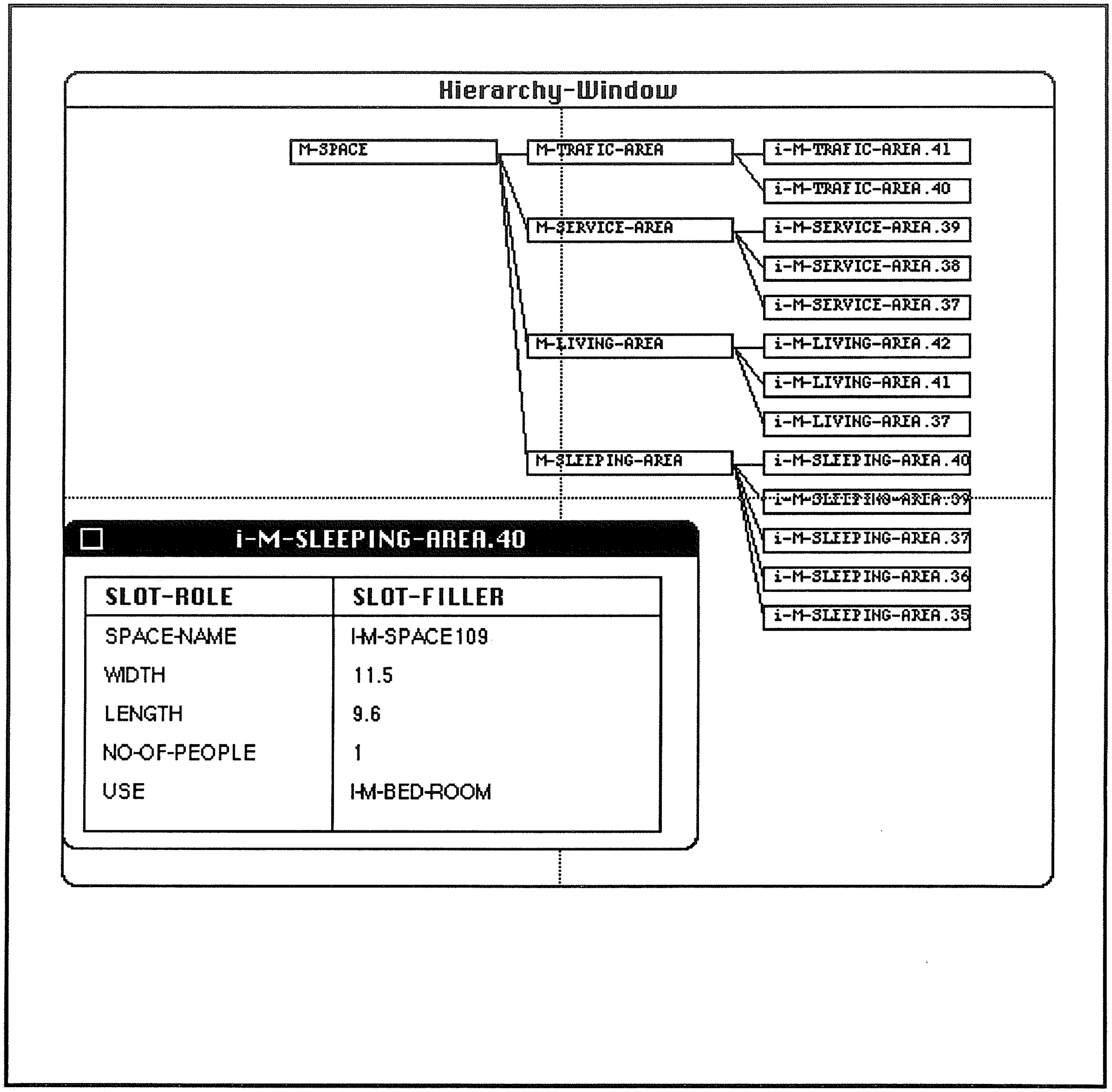
Figure 5.3.2-1 : MOP "i-m-layout001"

```
                    ( g8    m-sleeping-area
                            ( space-name  i-m-space108 )
                            ( width 11.5 )
                            ( length         13.0 )
                            ( no-of-people 1 )
                            (use             i-m-bed-room ))
                    ( g9    m-sleeping-area
                            ( space-name  i-m-space109 )
                            ( width 11.5 )
                            ( length         9.6 )
                            ( no-of-people 1 )
                            ( use            i-m-bed-room ))
                    ( g10   m-trafic-area
                            ( space-name  i-m-space110 )
                            ( width 11.5 )
                            ( length         3.4 )
                            ( use            i-m-hall ) )
                    ( g11   m-trafic-area
                            ( space-name  i-m-space111 )
                            ( width 6.5 )
                            ( length         13.0 )
                            ( use            i-m-entrance-hall ))
                    ( g12   m-living-area
                            ( space-name  i-m-space112 )
                            ( width 16.0 )
                            ( length         13.0 )
                            ( no-of-people 6 )
                            ( use            i-m-living-room ))
                    ( g13   m-living-area
                            ( space-name  i-m-space113 )
                            ( width 10.5 )
                            ( length         13.0 )
                            ( no-of-people 6 )
                            ( use            i-m-dining-room )))
( orientation   m-direction
                    ( north         m-north-orient
                            ( g1    i-m-space101 )
                            ( g2    i-m-space102 )
                            ( g3    i-m-space104 )
                            ( g4    i-m-space105 )
                            ( g5    i-m-space106 )
                            ( g6    i-m-space107 ))
```

Figure 5.3.2-2 : MOP "i-m-layout001" (continued)

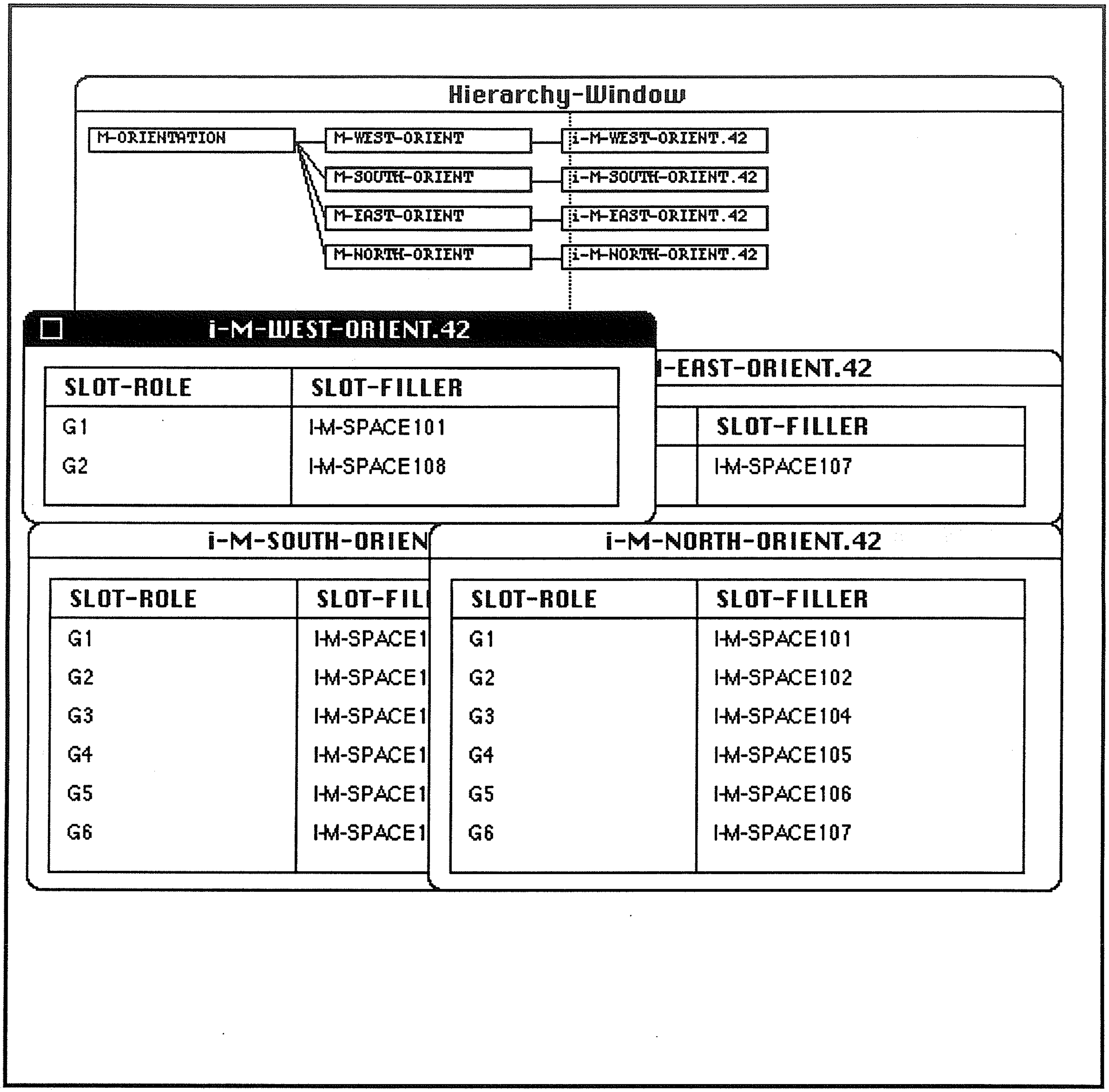
```
                                            ( east          m-east-orient
                                                    ( g1    i-m-space107 ))
                                            ( south         m-south-orient
                                                    ( g1    i-m-space108 )
                                                    ( g2    i-m-space109 )
                                                    ( g3    i-m-space111 )
                                                    ( g4    i-m-space112 )
                                                    ( g5    i-m-space113 )
                                                    ( g6    i-m-space107 ))
                                            ( west          m-west-orient
                                                    ( g1    i-m-space101 )
                                                    ( g2     i-m-space108 ))))
```

Figure 5.3.2-3 : MOP "i-m-layout001" (continued)



SLOT-ROLE	SLOT-FILLER
SPACE-NAME	I-M-SPACE109
WIDTH	11.5
LENGTH	9.6
NO-OF-PEOPLE	1
USE	I-M-BED-ROOM

Figure 5.3.3 : MOP "m-space"



Hierarchy-Window

i-M-WEST-ORIENT.42

SLOT-ROLE	SLOT-FILLER
G1	I-M-SPACE101
G2	I-M-SPACE108

-EAST-ORIENT.42

	SLOT-FILLER
	I-M-SPACE107

i-M-SOUTH-ORIEN

SLOT-ROLE	SLOT-FIL
G1	I-M-SPACE1
G2	I-M-SPACE1
G3	I-M-SPACE1
G4	I-M-SPACE1
G5	I-M-SPACE1
G6	I-M-SPACE1

i-M-NORTH-ORIENT.42

SLOT-ROLE	SLOT-FILLER
G1	I-M-SPACE101
G2	I-M-SPACE102
G3	I-M-SPACE104
G4	I-M-SPACE105
G5	I-M-SPACE106
G6	I-M-SPACE107

Figure 5.3.4 : MOP "m-orientation"

Even where no financial restriction exists, room sizes are limited by the area available for the building. Figure 5.3.5 shows the abstract hierarchy network of MOP "m-room-minimum". The MOP "m-room-minimum" contains knowledge about the minimum size of each room. The minimum size is calculated based on the requirement for the activity performed in each room (Chiara and Callender 90). E.g. the garage has two minimum size dimension, one for single car and the other for two cars.

Figure 5.3.6 shows the abstract hierarchy network of MOP "m-equal". MOP "m-equal" represents topological knowledge for graphical output. To keep traffic access from the entrance hall to the sleeping area and living area on the east walls of "space102", "space103", "space109", and "space110" (or the west wall of "space104" and "space111") are on a straight line. Moreover, because of the rectangular shape restriction for the garage, "space107", the east side of "space106" and "space113" are also on a straight line.

In terms of this model layout, some topological relationships, given as follows, should be taken into account.

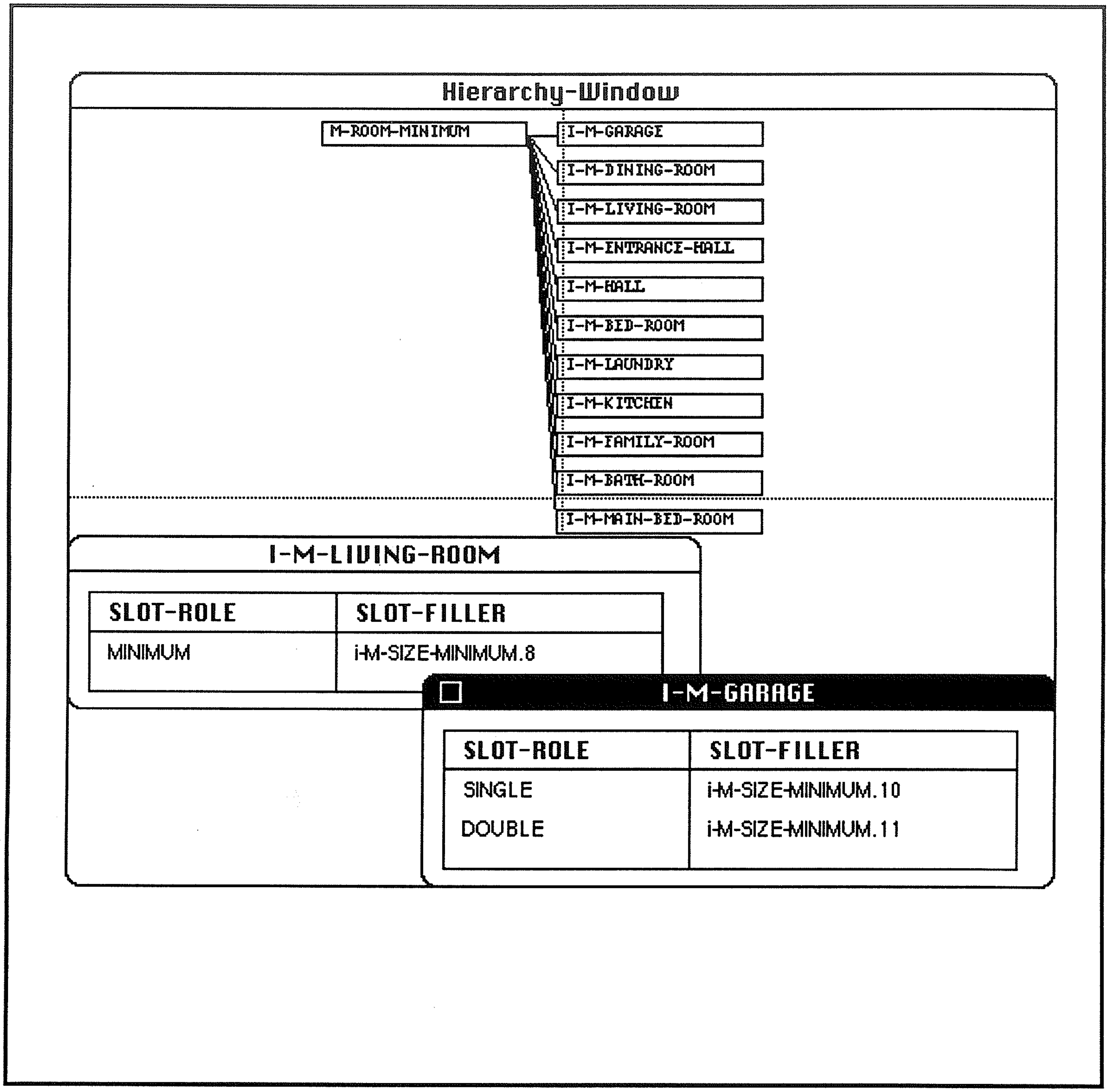
$$\begin{aligned} W_{101} + W_{102} &= W_{101} + W_{103} \\ &= W_{108} + W_{109} \\ &= W_{108} + W_{110} \end{aligned}$$

$$\boxed{W_{111} + W_{112} + W_{113} = W_{104} + W_{105} + W_{105}} \Leftarrow \text{Figure 5.3.6}$$

$$L_{101} = L_{102} + L_{103} = L_{104} = L_{105} = L_{106}$$

$$L_{108} = L_{109} + L_{110} = L_{111} = L_{112} = L_{113}$$

$$L_{101} + L_{108} = L_{107}$$



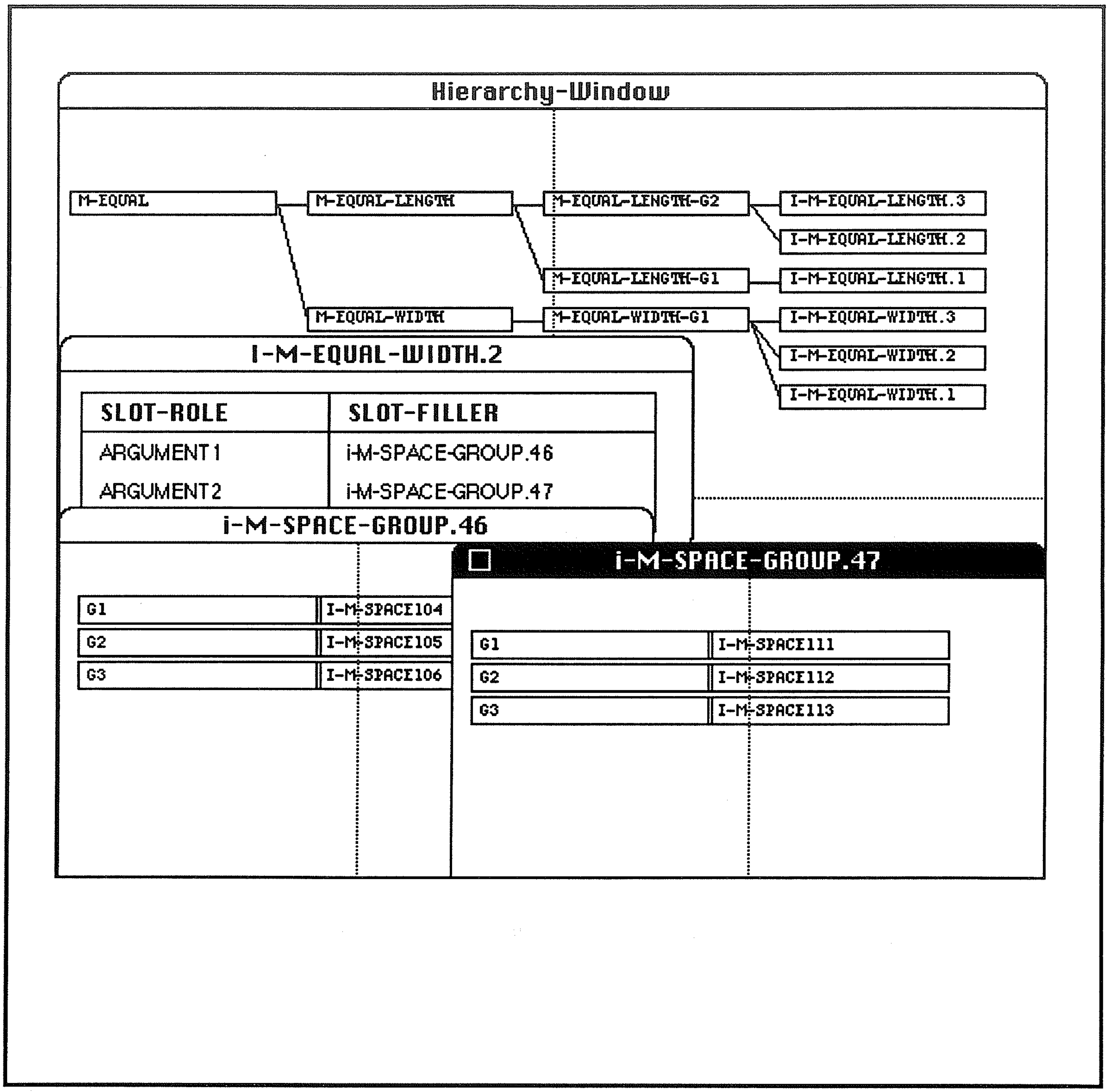
I-M-LIVING-ROOM

SLOT-ROLE	SLOT-FILLER
MINIMUM	i-M-SIZE-MINIMUM.8

I-M-GARAGE

SLOT-ROLE	SLOT-FILLER
SINGLE	i-M-SIZE-MINIMUM.10
DOUBLE	i-M-SIZE-MINIMUM.11

Figure 5.3.5 : MOP "m-room-minimum"



I-M-EQUAL-WIDTH.2

SLOT-ROLE	SLOT-FILLER
ARGUMENT1	i-M-SPACE-GROUP.46
ARGUMENT2	i-M-SPACE-GROUP.47

i-M-SPACE-GROUP.46

G1	I-M-SPACE104
G2	I-M-SPACE105
G3	I-M-SPACE106

i-M-SPACE-GROUP.47

G1	I-M-SPACE111
G2	I-M-SPACE112
G3	I-M-SPACE113

Figure 5.3.6 : MOP "m-equal"

5.4. Execution

Figure 5.4.1 illustrates the execution stages. Each stage has four steps. First, initial values for the building area, i.e. its width and length, are given. Next, the previously successful case is searched. Second, if there is a successful case in case memory, then the design method of that case is applied to the new case. On the other hand, if there isn't a case in memory, the design method "just-copy" ,which is defined as the first instance under MOP "m-design-tech", will be fired. All design methods defined in this program are shown in Figure 5.4.2. Third, all room size constraints are checked for compliance (Figure 5.4.3). Finally, if the constraints are compiled with, the case will be stored in case memory as a successful case. If not, it will be stored as a failed case with the failure reason. At step 4, the layout is displayed on the screen.

Figures 5.4.4-1 and 5.4.4-2 show how the case-based reasoning system works. The graphical output of the base residential house is shown in Figure 5.4.4. At stage 1, since there is not a successful case in case memory, the design method "just-copy" is applied to the new case. Consequently, the case succeeds, because the building area of the new case is the same as that of the base model. Therefore, this result is stored in case memory as a successful case. Figure 5.4.5 is the graphical result.

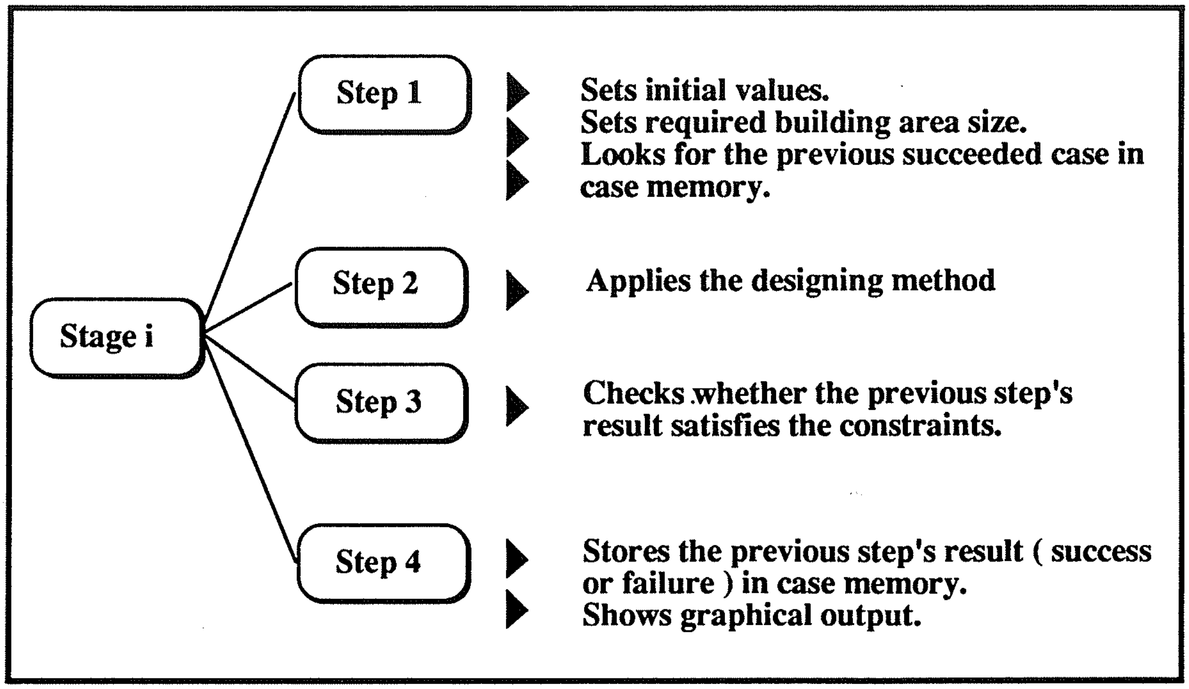

Stage i
Step 1
Sets initial values.
Sets required building area size.
Looks for the previous succeeded case in case memory.
Step 2
Applies the designing method
Step 3
Checks whether the previous step's result satisfies the constraints.
Step 4
Stores the previous step's result (success or failure) in case memory.
Shows graphical output.

Figure 5.4.1 : Tasks of Each Step

At stage 2, the new case, whose building area is 25.0' x 72.2', is given as the required size. At this stage, however, the design method "just-copy" is drawn from memory and applied, because "just-copy" is stored as a successful case in case memory. Otherwise, the case of the design method "just-copy" is stored as a failed case, since a couple of constraints are not satisfied. Following this failed result, other design methods are invoked to repair it. They are "shrink", "eliminate", and "partial shrink". Only the design method "partial shrink" satisfies the constraints in Figure 5.4.3. From Figure 5.4.7 to Figure 5.4.11 the graphical outputs of the above repaired design methods are illustrated.

At stage 3, the new building area, 24.0' x 65.0', is given. This results in the previous procedure not being applicable. At this stage, the design method "partial shrink" is applied immediately, because according to the previous stage's result "partial shrink" is already stored as a successful case in case memory as a case where the new building area size is smaller than that of base model.

Just Copy	Transfer all rooms of the base model house to the new required building area without changing the room size or other attributes.
Partial Shrink	Shrink some rooms of the base model to fit the new required building area.
Eliminate	Eliminate some rooms of the base model house to satisfy the required constraints.
Shrink	To shrink all rooms of base model house to fit new required building area.
Extend	Extend all rooms of the base model to fit new required building area (next version).
Rotate	Rotate some rooms to satisfy the required constraints (next version).
Change Adjacency	Change the adjacency of some rooms to satisfy the required constraints (next version).

Figure 5.4.2 : The Design Methods

Required building gross area size	≥	Total gross area size of each room
The side length of each direction of building area	≥	Total side length of each direction of each room
Main bedroom size	≥	The minimum size of main bedroom
Bathroom size	≥	The minimum size of bathroom
Family room size	≥	The minimum size of family room
Bedroom size	≥	The minimum size of bedroom
Hall size	≥	The minimum size of hall
Entrance hall size	≥	The minimum size of entrance hall
Living room size	≥	The minimum size of living room
Dining room size	≥	The minimum size of dinning room
Garage size	≥	The minimum size of garage

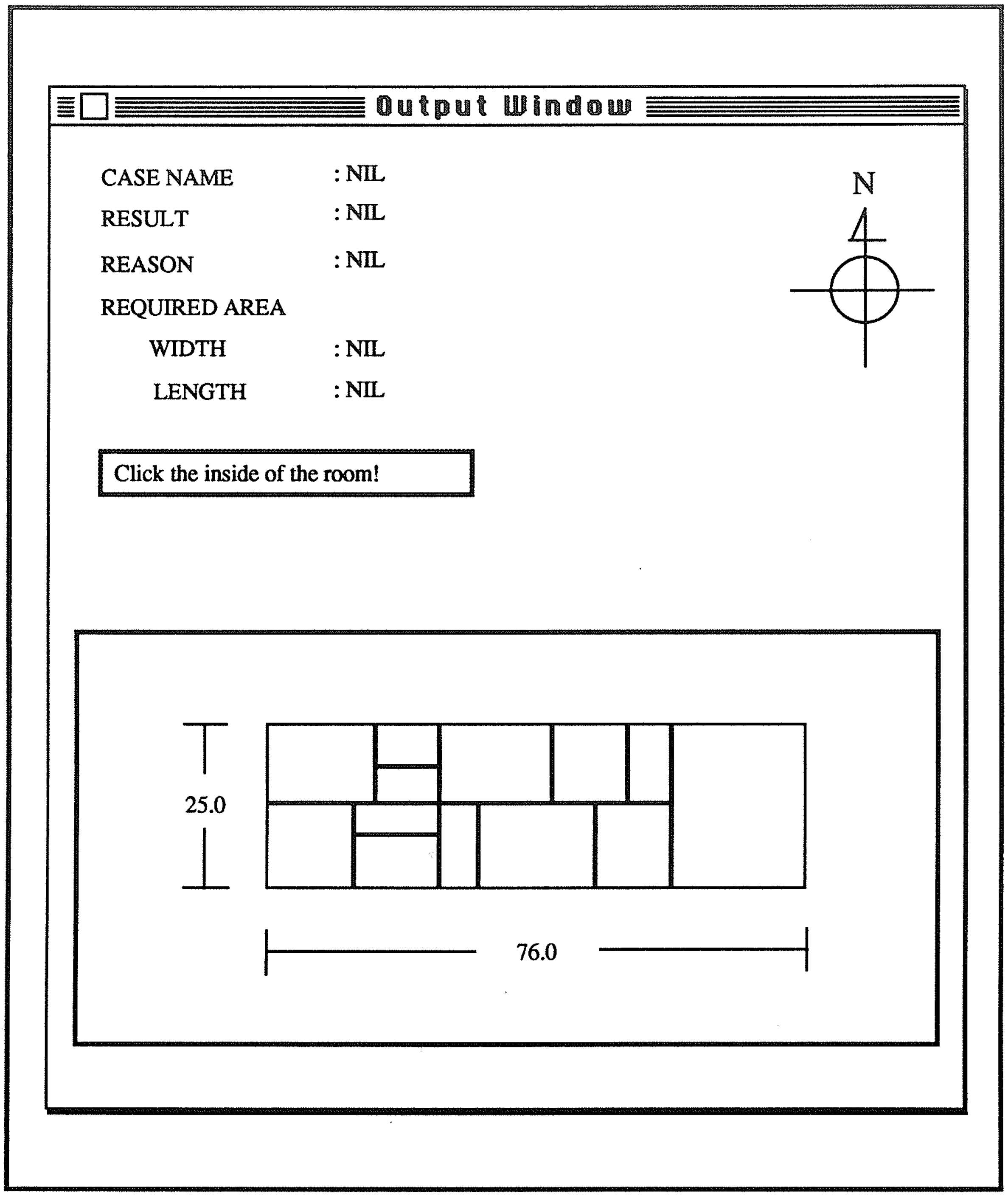
Figure 5.4.3 : The Constraints about Room Size

Base model	Building Area		25.0' x 76.0'
	Output		Figure 5.4.5
Stage 1	Step1	Building Area	25.0' x 76.0'
		Case Memory Searched Result	No Succeeded Case
	Step2	Applied Designing Method	Just Copy
	Step3	Applied Result	Satisfied All Constraints
	Step4	Stored	Succeeded Case
		Output	Figure 5.4.6
Stage2	Step1	Building Area	25.0' x 72.2'
		Case Memory Searched Result	Just Copy
	Step2	Applied Designing Method	Just Copy
	Step3	Applied Result	Not Satisfied All Constraints
	Step4	Stored	Failed Case Failed Explanation
		Output	Figure 5.4.7

Figure 5.4.4-1 : The Result of Each Step

Repaired Method =>			
	Step1	-	-
	Step2	Applied Design Method	Shrink
	Step3	Applied Result	Not Satisfied All Constraints
	Step4	Stored	Failed Case
			Failed Explanation
		Output	Figure 5.4.8
Repaired Method =>			
	Step1	-	-
	Step2	Applied Design Method	Eliminate
	Step3	Applied Result	Not Satisfied All Constraints
	Step4	Stored	Failed Case
			Failed Explanation
		Output	Figure 5.4.9
Repaired Method =>			
	Step1	-	-
	Step2	Applied Design Method	Partial Shrink
	Step3	Applied Result	Satisfied All Constraints
	Step4	Stored	Succeeded Case
		Output	Figure 5.4.10
Stage3	Step1	Building Area	24.0' x 65.0'
		Case Memory Searched Result	Partial Shrink
	Step2	Applied Design Method	Partial Shrink
	Step3	Applied Result	Satisfied All Constraints
	Step4	Stored	Succeeded Case
		Output	Figure 5.4.11

Figure 5.4.4-2 : The Result of Each Step

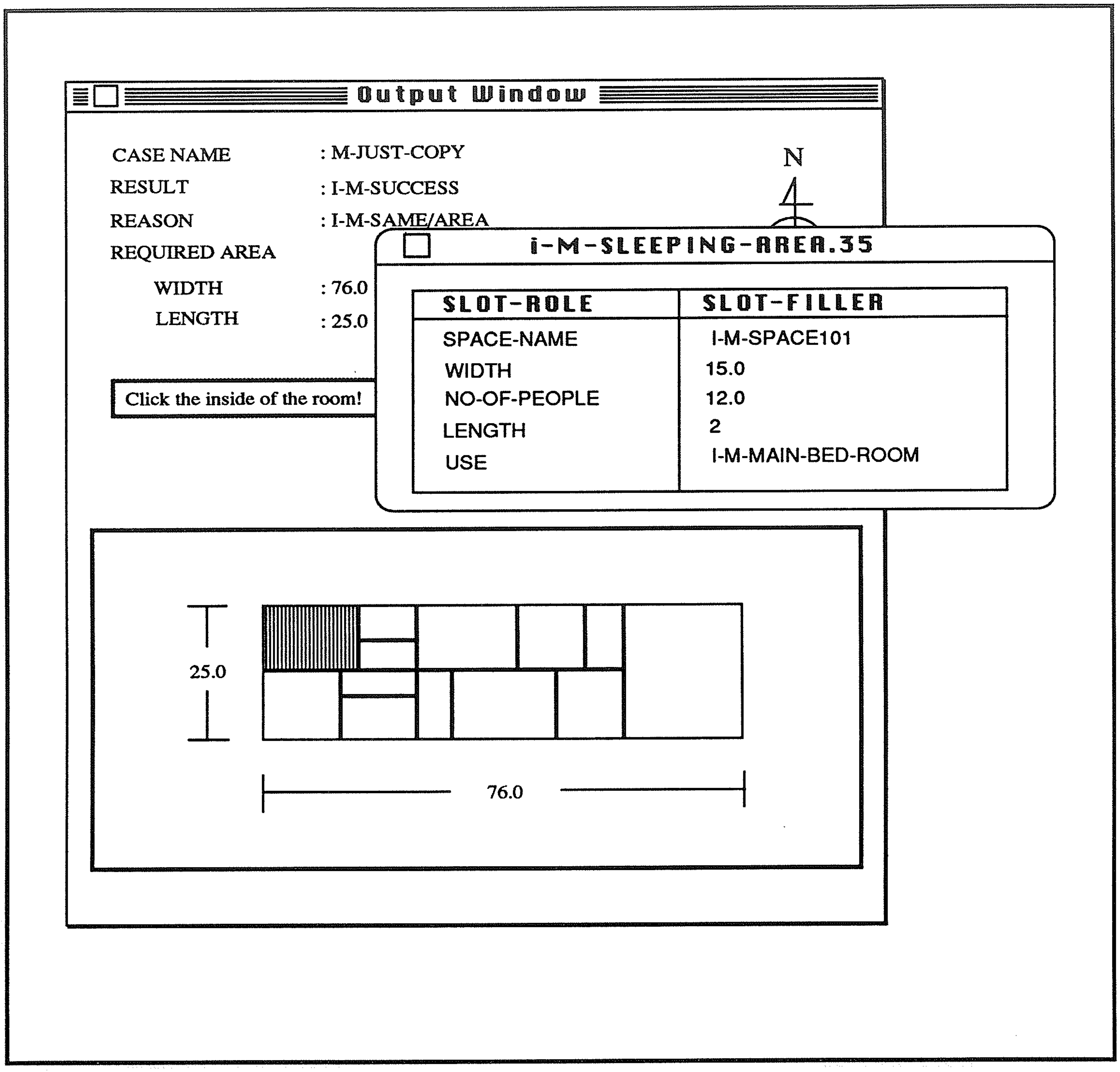


Output Window

CASE NAME : NIL
RESULT : NIL
REASON : NIL
REQUIRED AREA
WIDTH : NIL
LENGTH : NIL

Click the inside of the room!

Figure 5.4.5 : Graphical Output Of A Base Model House



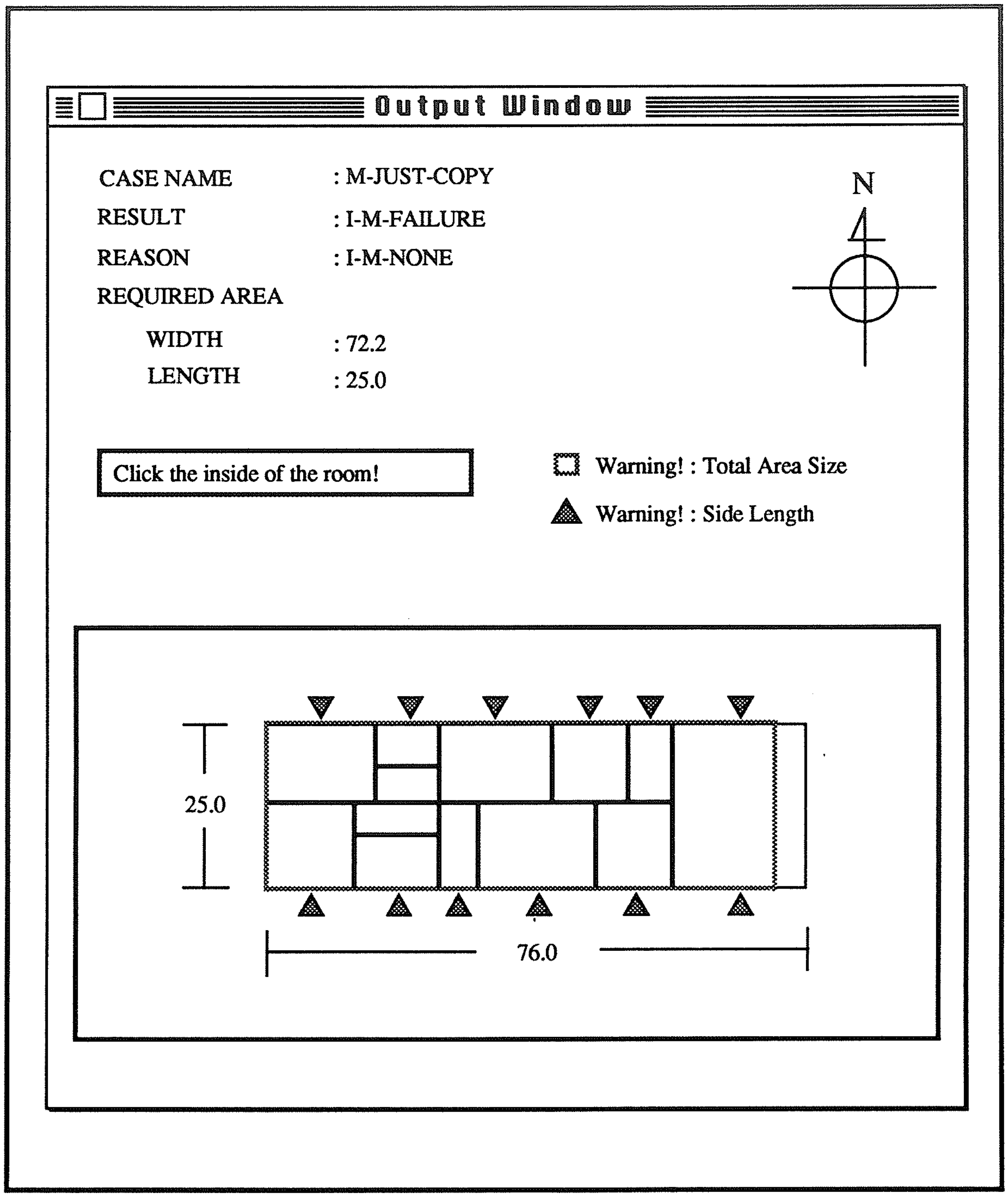
```
CASE NAME        : M-JUST-COPY
RESULT           : I-M-SUCCESS
REASON           : I-M-SAME/AREA
REQUIRED AREA
    WIDTH        : 76.0
    LENGTH       : 25.0
```

Click the inside of the room!

I-M-SLEEPING-AREA.35

SLOT-ROLE	SLOT-FILLER
SPACE-NAME	I-M-SPACE101
WIDTH	15.0
NO-OF-PEOPLE	12.0
LENGTH	2
USE	I-M-MAIN-BED-ROOM

Figure 5.4.6 : Graphical Output of Stage 1

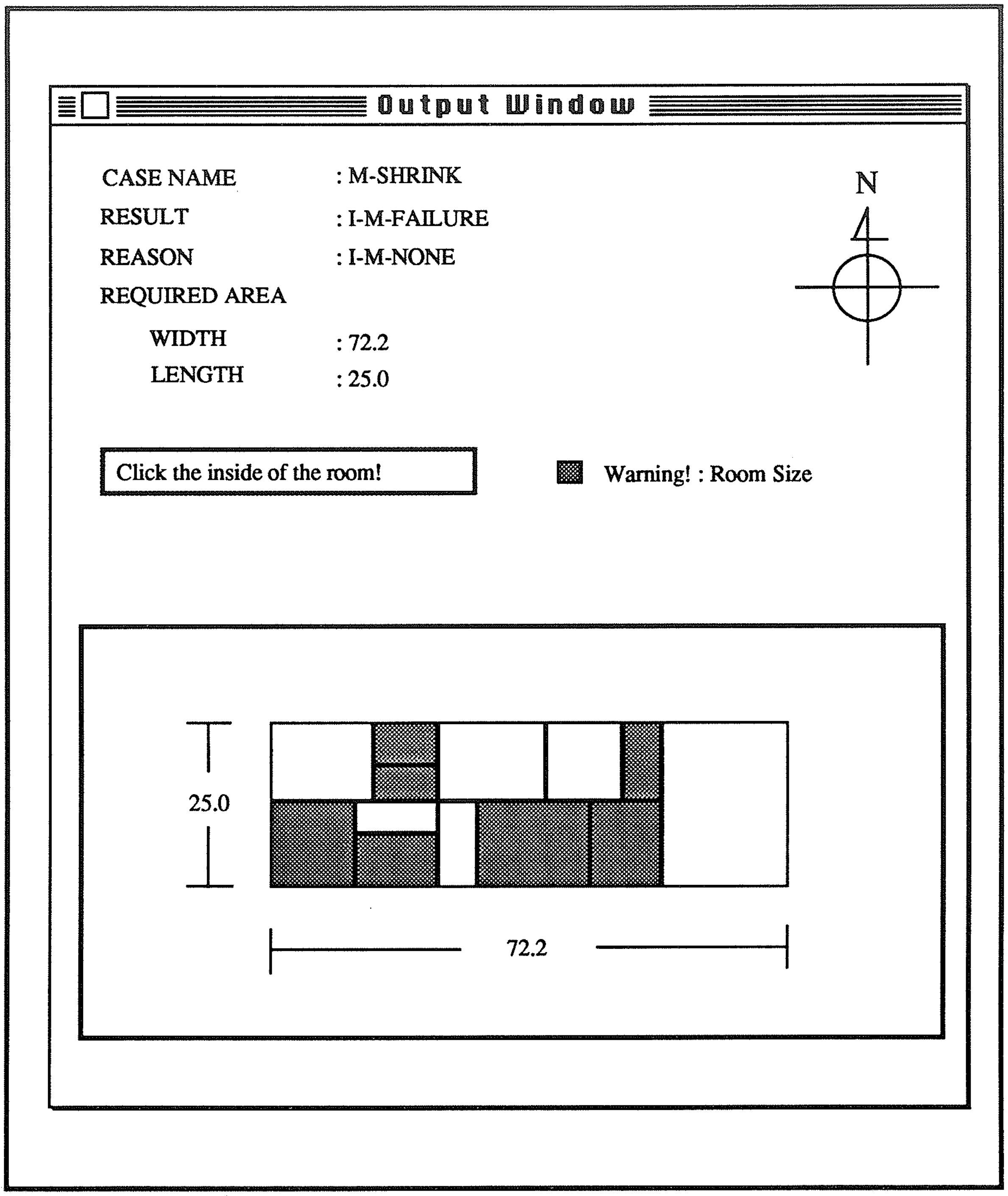

Output Window

CASE NAME : M-JUST-COPY
RESULT : I-M-FAILURE
REASON : I-M-NONE
REQUIRED AREA
WIDTH : 72.2
LENGTH : 25.0

Click the inside of the room!

▢ Warning! : Total Area Size
▲ Warning! : Side Length

Figure 5.4.7 : Graphical Output of Stage 2 no. 1



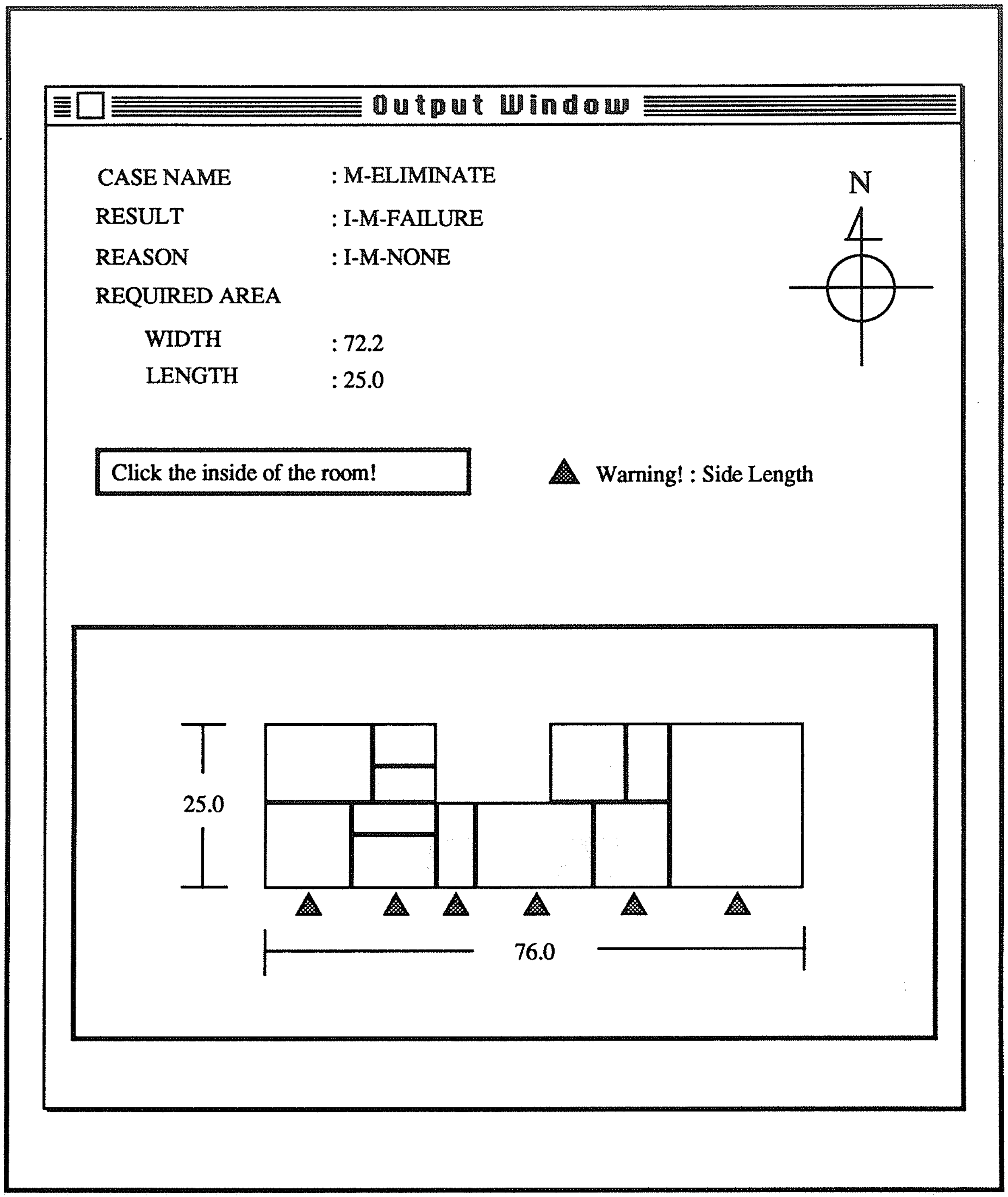
Output Window

CASE NAME	: M-SHRINK
RESULT	: I-M-FAILURE
REASON	: I-M-NONE
REQUIRED AREA	
WIDTH	: 72.2
LENGTH	: 25.0

Click the inside of the room!

▨ Warning! : Room Size

Figure 5.4.8 : Graphical Output of Stage 2 no. 2

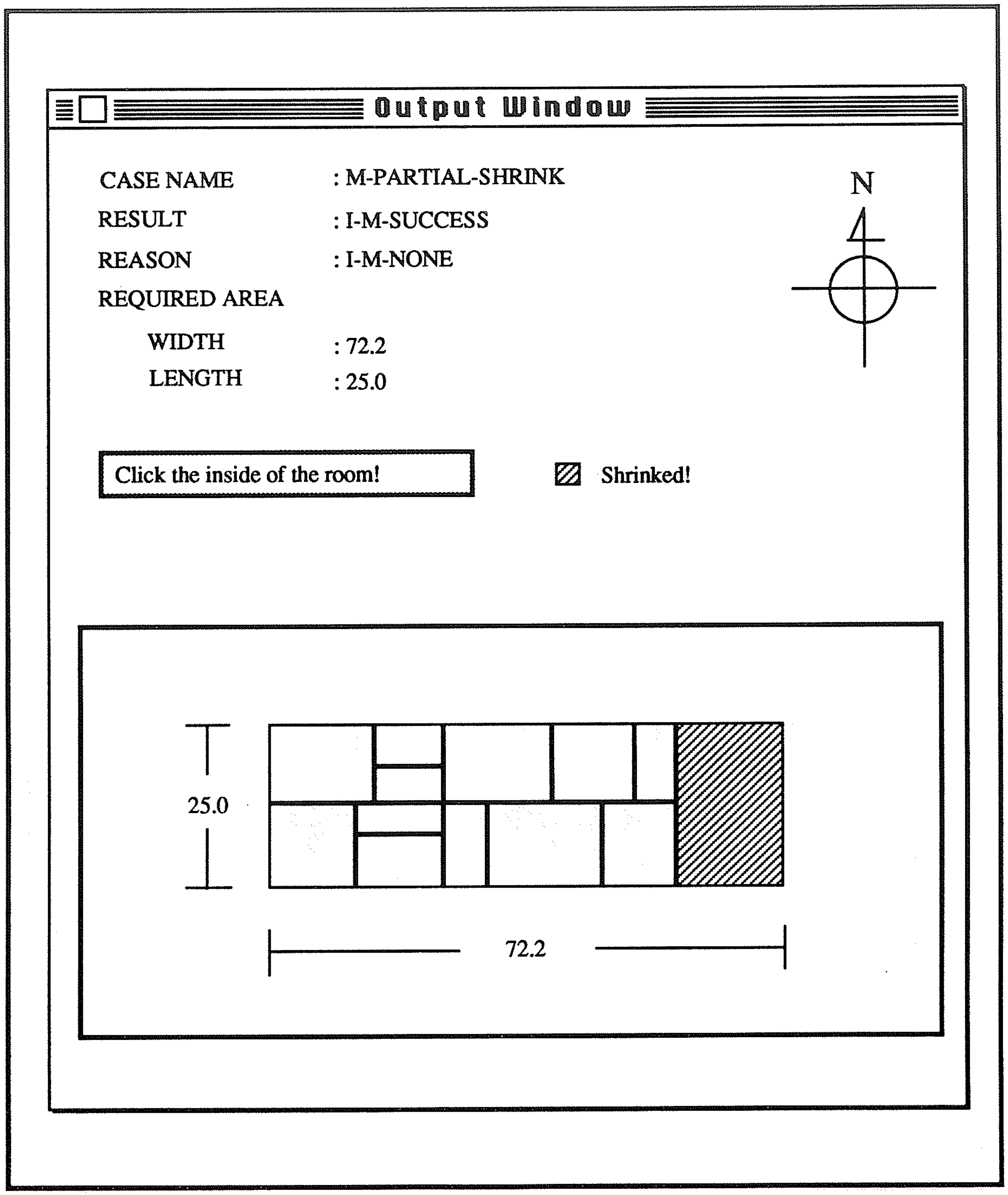


CASE NAME : M-ELIMINATE
RESULT : I-M-FAILURE
REASON : I-M-NONE
REQUIRED AREA
WIDTH : 72.2
LENGTH : 25.0

Click the inside of the room!

▲ Warning! : Side Length

Figure 5.4.9 : Graphical Output of Stage 2 no. 3



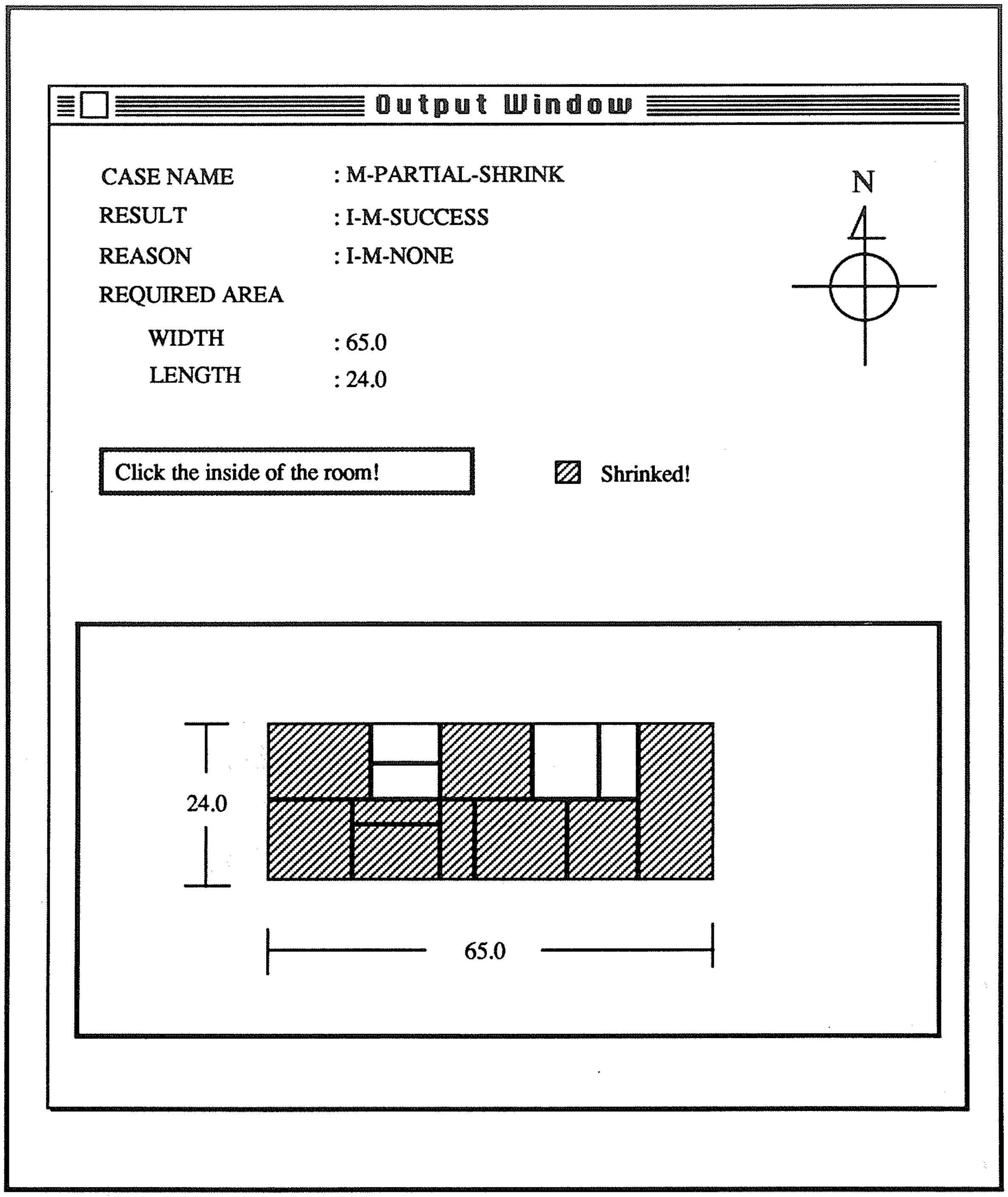
Output Window

CASE NAME	: M-PARTIAL-SHRINK
RESULT	: I-M-SUCCESS
REASON	: I-M-NONE
REQUIRED AREA	
WIDTH	: 72.2
LENGTH	: 25.0

Click the inside of the room!

▨ Shrinked!

Figure 5.4.10 : Graphical Output of Stage 2 no. 4



CASE NAME : M-PARTIAL-SHRINK
RESULT : I-M-SUCCESS
REASON : I-M-NONE
REQUIRED AREA
WIDTH : 65.0
LENGTH : 24.0

Click the inside of the room!

▨ Shrinked!

Figure 5.4.11 : Graphical Output of Stage 3

6. Conclusions

When an expert system development shell, based on rule-based reasoning, was first introduced in public as the epochal problem-solving method several years ago, many people expected that it would provide innovative results in the near future. They, however, recognized that rule-based reasoning could not cover the whole process of a human being after examining it in many domains. Recently, case-based reasoning has begun to attract much attention from AI researchers as a method which can make up for rule-based reasoning's deficiency. In this paper I introduced a case-based reasoning environment, named CBRE, and a simple prototype system for automated floor layout generation. CBRE's functions are based on Schank and Riesbeck's Micro MOPs and developed in Allegro Common Lisp and FrameKit. It provides not only case-based reasoning functions, but graphical windows which show abstract hierarchy networks and slot networks.

Due to time limits, the prototype system contains only adapting and storing functions. Nevertheless, through the development process, I noticed a few points that should be kept in mind to make a case-based reasoning model for automated floor generation system.

- MOPs must be a useful method for simulating a human being's case-based reasoning process. Otherwise, when we use MOPs for our problem domains, we should define all words in advance, except numerical numbers, which are referred to in slot values.
- Using MOPs for the geographical knowledge for graphical output of rectangular dissections is not a good idea from the point of processing speed. Because it takes time to check the abstract hierarchy of geographical knowledge whenever we need the coordinates of vertexes for drawing a rectangular shaped room at new location.

Throughout the whole development process, the following two issues always concerned me.

- In the storing stage, the most important thing is indexing. So far as the prototype developed in my research is concerned, it needs only two indices, success or failure. However, if we attach the function of acquiring knowledge about using design techniques, we should find the indices to store these knowledge in memory for future use. How can I find them from a designer's work and how can I generalize them for reasoning?

- When we use MOPs in a residential house design domain, we should define thecharacteristics of a house with texts. Are there some effective ways to distinguish cases with texts?

Case-based reasoning is one of the efficient methods which can provide reasoning intelligence like a human being. Retrieving and storing functions work like a human's reminding ability, the adapting function works like a human's partial matching and comprehending capabilities, and the evaluating explanation function works like a human's explanation capability. When I started to struggle against my project, a case-based reasoning technique reminded me of computer scientists' eternal theme : "Can machines think like a human?". At this time we can not answer in the affirmative. I, however, believe our honest efforts will bring the day when we can return "yes" as its answer in the future.

7. Acknowledgements

I would like to acknowledge the advice and the encouragement of the following people throughout the course of my studies.

- Professor Paul M. Teicholz :

  Department of Civil Engineering & Director of CIFE Civil Engineering, Stanford University.

  For his encouragement and advice as the principle investigator of my CIFE architectural layout project.

- Mr. Paul Chinowsky :

  Graduate student in Civil Engineering, Stanford University.

  For his help and suggestions as a collaborator of my CIFE architectural layout project.

- Professor Naruo Kano :

  Department of Architecture, Waseda University.

  Visiting Professor for Department of Civil Engineering Stanford University from 1989 to 1990.

  For his encouragement of my research.

- Dr. Yumi Iwasaki :

  Research Associate of Knowledge System Laboratory, Stanford University.

  For her encouragement of my research and advice from her experience as an AI researcher.

- Miss Laura A. Chase :

  Architect at Carrasco & Associates, Palo Alto, CA. and part time Civil Engineering Instructor.

  For her practical advice of my research from her experience as an architectural designer.

- Professor Kincho Law :

  Department of Civil Engineering, Stanford University.

  For his encouragement of my research.

◦ Professor Raymond E. Levitt :

Department of Civil Engineering, Stanford University.

For his encouragement of my research.

I would also like to express my thanks to Ken Ito and Hiroyuki Fuyama for providing technical information and advice for my research. Furthermore my particular thanks go to Chris Madison and Leyla Unerdem who helped with my studies at Stanford.

8. Bibliography

(Amor, Groves, and Donn 90)
Amor R. W., Groves L. J. & Donn M. R. 1990. *Integrating Design Tools for Building Design*, Paper at the Symposium on " Artificial Intelligence in Building Design : Progress and Promise", ASHRAE Conference, 9-13 June, St. Louis, MO.

(Ashley 89)
Ashley K. D. 1989. *Case-Based Reasoning* : The Sixth IEEE Conference on Artificial Intelligence Applications, Tutorial Notes.

(Baybars and Eastman 80)
Baybars I. & Eastman C. M. 1980. *Enumerating Architectural Arrangements by Generating Their Underlying Graphs*, Environment and Planning B, Vol. 7 : pp. 289-310.

(Baybars 82)
Baybars I. 1982. *The Generation of Floor Plans with Circulation Spaces*, Environment and Planning B,

(BDPSA. 76)
Building Development Property Service Agency, Department of the Environment, UK 1976. *A Primer for Users and Designers*, Her Majesty's Stationery Office, London,UK.

(Chase 89)
Chase S. C. 1989. *Shapes and Shape Grammars : From Mathematical Model to ComputerImplementation*, Environment and Planning B, Vol. 16 : pp. 215-242.

(Chiara and Callender 90)
Chiara J. D. & Callender J. H. 1990. *Time-Saver Standards for Building Types : 3rd Edition*, McGraw-Hill, Inc., New York, NY.

(Chinowsky 90, Working Paper)
Chinowsky P. 1990. *A Knowledge-based Layout Generation System*, Center for Integrated Facility Engineering Stanford University Working Paper no. 7.

(Chinowsky 90, Resear Proposal)
Chinowsky P. 1990. *A Knowledge-based System for Architectural Layout Generation,* Research Proposal Stanford University.

(Conye and Gero 85)
Conye R. D. & Gero J. S. 1985. *Design Knowledge and Sequential Plans*, Environment and Planning B, Vol. 12, no. 4 : pp. 401-418.

(Conye and Gero 85)
Conye R. D. & Gero J. S. 1985. *Logic Programming as a Means of Representing Semantics in Design Languages*, Environmental and Planning B, Vol. 12, no. 3 : pp. 351-369.

(Duffy 74)
Duffy F. 1974. *Office and Organizations : 1. Theoretical Basis*, Environments and Planning B, Vol. 1 : pp. 105-118.

(Duffy 74)
Duffy F. 1974. *Office and Organizations : 2. The Testing of a Hypothetical Model*, Environment and Planning B, Vol. 1 : pp. 217-235.

(Duffy, Cave and Worthinton 76)
Duffy F., Cave C. & Worthington J. 1976. *Planning Office Space*, The Architectural Press Ltd., London. & Nichols Publishing Company, New York, NY.

(Flemming 78)
Flemming U. 1978. *Wall Representations of Rectangular Dissections and Their Use in Automated Space Allocation*, Environment and Planning B, Vol. 5 : pp. 215-232.

(Flemming, Coyne, Glavin, Hsi, and Rychener 89)
Flemming U., Coyne R. F., Glavin T., Hsi H., & Rychener M. D.1989. *A Generative Expert System for the Design of Building Layouts (Final Report)*, EDRC- Canegie Mellon University, Pittsburgh, PA.

(Friesz, MIller and Tobin 88)
Friesz T. L., Miller T. & Tobin R. L. 1988. *Algorithms for Spatially Competitive Network Facility- location*, Environment and Planning B, Vol. 15 : pp. 191-203.

(Gilleard 78)
Gilleard J. 1978. *Layout : A Hierarchical Computer Model for The Production of Architectural Floor Plans*, Environment and Planning B, Vol. 5 : pp. 223-241.

(Hammond 89)
Hammond, K. J. 1989. *Case-Based Planning : Viewing Planning as a Memory Task*, Academic Press, Boston, MA.

(Harris, Palmer, Lewis, Munson, Meckler and Gerdes 81)
Harris D. A., Palmer A. E., Munson D. L., Meckler G. and Gerdes R. 1981. *Planning and Designing the Office Environment*, Van Nostrand Reinhold Company, New York, NY.

(Hillier and Hanson 84)
Hillier B. & Hanson J. 1984. *The Social Logic of Space*, Cambridge University Press, Cambridge, CB.

(Ishida 89)
Ishida Y. 1989. *A Framework for Dynamic Representation of Knowledge : A Minimum Principle in Organizing Knowledge Representation*, Paper presented at the 76th Knowledge Engineering System Subcommittee.

(Kolodner and Riesbeck 86)
Kolodner J. L. & Riesbeck, C. K. 1986. *Experience, Memory, and Reasoning*, Lawrence Erlbaum Associates, Inc., Hillsdale, NJ.

(Kolodner and Riesbeck 90)
Kolodner J. L. & Riesbeck C. K. 1990. *Case-Based Reasoning : AAAI-90 Tutorial MP5*.

(Lang 87)
Lang J. 1987. *Creating Architectural Theory : The Role of the Behavioral Sciences in Environmental Design*, Van Nostrand Reinhold, New York, NY.

(Marti 81)
Marti M. 1981. *Space Operational Analysis : A Systematic Approach to Spatial Analysis and Programming*, PDA Publishers Corp., West Lafayette, IN.

(Mitchell, Steadman and Liggett 76)
Mitchell W. J., Steadman J. P. & Liggett R. S. 1976. *Synthesis and Optimization of Small Rectangular Floor Plans*, Environment and Planning B, Vol. 3 : pp. 37-70.

(Mitchell 90)
Mitchell W. J. 1990. *The Logic of Architecture : Design, Computation, and Cognition*, MIT Press, Cambridge, MA.

(Montreuil 90)
Montreuil B. 1990. *Requirements for Representation of Domain knowledge in Intelligent environments for Layout Design*, Computer-Aided Design, Vol. 22, no. 2.

(Oxman and Gero 88)
Oxman R. & Gero J. S. 1988. D*esigning by Prototype Refinement in Architecture*, Artificial Intelligence in Engineering: Design, Computational Mechanics Publications, Southampton, NY.

(Pile 78)
Pile J. 1978. *Open Office Planning : A Handbook for Interior Designers and Architects*, Whitney Library of Design, New York, NY.

(Riesbeck and Shank 89)
Riesbeck, C. K., & Shank, R. C. 1989. *Inside Case-Based Reasoning*, Lawrence Erlbaum Associates, Inc., Hillsdale, NJ.

(Rinsma 87)
Rinsma I. 1987. *Rectangular and Orthogonal Floorplans with Required Room Areas and Tree Adjacency*, Environment and Planning B, Vol. 15, no.1: pp. 111-118.

(Saphier 68)
Saphier M. 1968. *Office Planning and Design*, McGraw-Hill, Inc., New York, NY.

(Schank 82)
Schank, R. 1982. *Dynamic Memory : A Theory of Learning in Computers and People*, Cambridge University Press, New York, NY.

(Stiny 80)
Stiny, G. 1980. *Introduction to Shape and Shape Grammars*, Environment and Planning B,Vol. 7: pp. 343-351.

(Stiny 81)
Stiny, G. 1981. *A Note on The Description of Designs*, Environment and Planning B, Vol. 8: pp. 257-267.

(Talcott, Hepler and Wallach 86)
Talcott C., Hepler D. & Wallach P. 1986. *Home Planner's Guide to Residential Design*, McGraw-Hill, Inc., New York, NY.

(Tompkins and White 84)
Tompkins J. A. & White J. A. 1984. *Facilities Planning*, John Wiley & Sons, Inc., New York, NY.

(Webster 89)
Webster C. J. 1989. *A Theorem-proving Approach to Spacial Problem-Solving*, Environment and Planning B, Vol. 16: pp. 171-186.

(Wilson and Watkins 89)
Wilson R. J. & Watkins J. J. 1989. *Graphs : An Introductory Approach*, John Wiley & Sons, Inc., New York, NY.

(Winston and Horn 89)
Winston P. H. & Horn B. K. P. 1989. *LISP : Third Edition*, Addison-Wesley Publishing Company, Inc.

Appendix : A

Reinstantiation : CHEF (Hammond)

New Problem: Stir-fried chicken and snow peas

Old Case: Beef & Broccoli
< Ingredients >
1/2 lb. beef, 1/2 lb. broccoli, 2 tbs. soy sauce, ...
< Method >
Chop garlic, shred beef, marinate beef in ...,
chop broccoli into chunks, stir fry broccoli for 3 minutes,
remove broccoli from pan, stir fry spices, wine, beef for 3 minutes,
add broccoli, stir fry another 1/2 minute, add salt.

New Dish: Reinstantiate with chicken and snow peas
< Ingredients >
1/2 lb. chicken, 1/2 lb. snow peas, 2 tbs. soy sauce, ...
< Method >
Chop garlic, shred chicken, marinate chicken in ...,
chop snow peas into chunks, stir fry snow peas for 3 minutes,
remove snow peas from pan, stir fry spices, wine, chicken for 3 minutes,
add snow peas, stir fry another 1/2 minute, add salt.

Appendix : B

Parameter Adjustment : JUDGE (Bain)

New Problem: Moe struck Ed several times.
Ed was slightly hurt. Ed struck Moe several times. Moe fell down.
Moe struck Ed several times, breaking Ed's nose.
Ed stabbed Moe with a knife one time, killing Moe. (CRIME4)

Old Case: Hal struck Gary several times.
Gary was slightly hurt. Gary struck Hall several times. Hal fell down.
Hal hit Gary very hard several times, breaking Gary's ribs.
Gary shot at Hal with a gun several times, killing Hal.
Gary's sentence, for murder, was a term of imprisonment of not less than 25 years. (CRIME3)

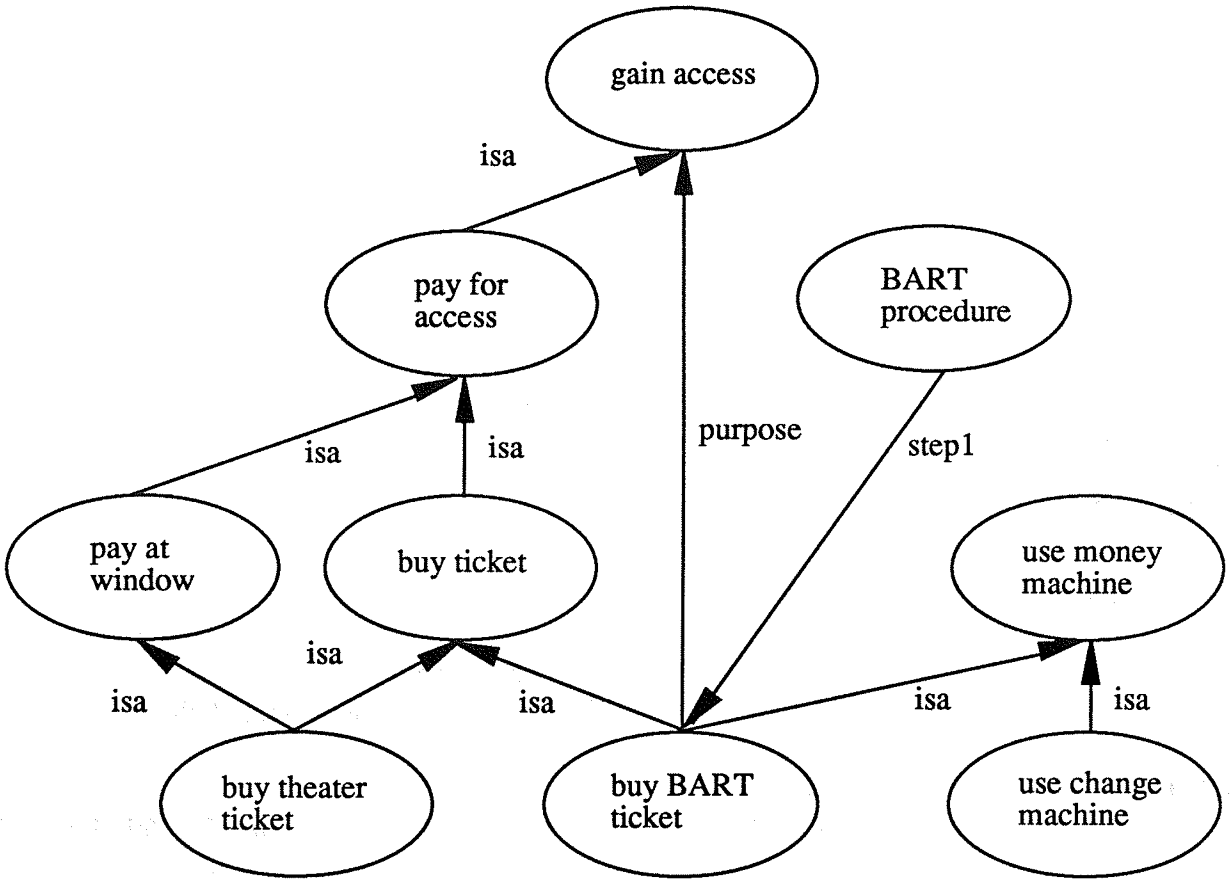
New Solution: The sentence for CRIME4 will be 15 years.

- Compare Problem Descriptions -

In both crimes, the victim was killed. Although results were the same in both situations, the outcome in the newer crime was accidental and involved a less severe intent. Ed's action did not demonstrate as extreme force as Gary's action. Gary's intent, shown by the repeated shooting, was more severe than that of Ed. ...
The old crime, CRIME3, was substantially worse.

Local Search : PLEXUS (Alterman)

New Problem:	First trip on New York City subway.
Old Case:	On BART, I buy a ticket from a machine that is used on entry and exit.
Problem:	No ticket machines on New York City subway.
New Solution:	Walk up to BUY TICKET, then down to buy theater ticket. This new step suggests looking for a person at a ticket window.



Appendix : D

Query Memory : JULIA (Hinrichs)

New Problem:	Plan a meal, method of preparation should be stew.
Old Case:	Main dish was bouillabaisse
New Information from Client:	Several meat and potatoes men will be attending. I need a stew with meat and potatoes.
Proposed Fix:	Substitute a dish, prepared by stewing, with ingredients meat and potatoes.
Local search:	Yields no appropriate dish.
Query Memory:	Find a instance of "A dish, to be used as a main dish, prepared by stewing, with ingredients meat and potatoes". => Irish Stew is found.
Proposed Solution:	Serve Irish Stew as the main dish.

Appendix : E

Specialized Search : SWALE

New problem:	Why did Len Bias, a healthy all-star college basketball player, die of a heart attack.
Old Case:	Jim Fixx, an apparently healthy jogger, died of a heart attack after running, because of a hidden heart defect.
Proposed Solution:	Len Bias had a hidden heart defect.
Problem:	ACTOR-ACTION MISMATCH: STEREOTYPE VIOLATION => Len Bias is not a jogger.
Specialized Search:	Heuristic => SUBSTITUTE ACTION: ACTOR THEME
Result of search:	Bias, a basketball player, normally does practice shots and wind sprints (a kind of running).
Repaired Solution:	Len Bias had a hidden heart defect, and doing wind sprints led to the heart attack.

Appendix : F

Transformation : JULIA (Hinrichs)

New Problem: Planning an Italian meal, want to serve pasta, some guests are vegetarians, attempting to derive main course.

Old Case: Italian meal, main course was lasagne.

Solution Based on Old Case:
Serve lasagne.

Problem: Violates vegetarian constraint.

Transformed Solution: Vegetarian lasagne.
< Solution 1 > delete secondary ingredient
=> delete meat
< Solution 2 > substitute feature, focusing on its function as a texture-adder
=> meat -> spinach
< Solution 3 > substitute feature, focusing on its function as a protein-food
=> meat -> tuna

Appendix : G

Critic Application : CHEF (Hammond) Inserting a Step in a Plan

New Problem: Create a pasta dish that includes duck.

Old Case: Ants-climb-a-tree (ground pork and pasta)

Critic Solution: Recipe included a defatting step before cooking duck.

< Adaptation >
• Reinstantiation => Substitution the roles of the old case.
• Critic Application => When cooking duck, defat it before cooking.

- Note -
• This critic is attached to the concept duck, Whenever the concept duck is encountered during problem solving, its critics are checked for applicability.
• This critic is learned by CHEF after it fails to defat duck in a previous recipe.

Appendix : H

Critic Application : JULIA (Hinrichs)
Divide & Conquer

New Problem: Guests include a vegetarian who is allergic to milk products and someone who only eats "normal" food.

Problem: Over-constraint feature => Main dish.

Critic Solution: Provide two main dishes.
- Vegetarian and others will eat tomato bake.
- Picky eater and others will eat barbecued chicken.

< Adaptation >
- IF a slot cannot be filled because of overconstraint, and overconstraint is due to constraints generated from individual members of a group,

THEN attempt to increase choice by splitting the overconstrained slot and partitioning the constraints.

Appendix : I

Derivational Replay : JULIA (Hinrichs)

New Problem: Plan a school lunch, include chicken season is winter.

Old Case: A nursing home lunch in the summer, where chicken was served. The meal was melon, green salad, broiled chicken, mashed potatoes, milk.

Problem: Melon is unavailable.

Solution: Melon was chosen by a method of choosing a fruit appropriate to the season of the year. Using the same method, apple, orange, grapefruit, and pear are generated as plausible substitutes for melon.

< Derivational Replay >
Look at the reasons melon was chosen in the old meal. Using the same reasoning, choose an item for the new meal.

Appendix : J

Name	Developer	Domain	Case	Input	Output
CLAVIER	Lockheed AI Center 1989	autoclave layout and scheduling	previous good layout	list of parts and their priorities	schedule of loads and layouts for each load
CEILA	Georgia Tech 1989	car mechanics	previous broken cars	list of symptoms	diagnostic and repair plan
Battle Planning	Cognitive Systems,Inc. 1988	battle planning	land warfare database from Data Memory Systems, Inc.	data about intelligence and operational information	similar prior battles with outcomes and analysis
Tactical Assistant	Texas Instruments 1982	scenario projection in battle planning	tactical episodes, pedagogically organized	set of goals for friendly forces, a tactical situation	possible courses of actions and outcomes
JUDGE	Yale 1986	criminal sentencing	previous crimes and sentences	description of crime and statue violated	a sentence
MEDIATOR	Georgia Tech 1985	political dispute mediation	disputes and resolutions	description of goal conflicts, e.g. territorial dispute	possible resolution, e.g. split control of territory
PROTOS	Univ. of Texas 1988	clinical audiology	prior diagnoses	description of patient's symptoms	diagnostic classification
CASEY	MIT 1988	heart failure diagnosis	diagnoses for a variety of cases generated by Heart Failure Program	description of patient's symptoms	causal network connecting symptoms with internal states suggested therapy
PERSUADER	Georgia Tech 1987	labor contract dispute resolution	contracts, negotiation episodes, and management problems	description of business plus management and labor goals	a contract
CYCLOPS	CMU 1988	landscape design	layouts with results and explanations of results	goal, e.g. house on firm ground, or possible layout, e.g. house on a hill	possible layout or evaluation of layout
HYPO	Univ. of Mass 1987	patent law, particularly trade secrets	trade secret cases and their outcomes	description of trade secret violation	a lattice of similar and contrasting precedents
JULIA	Georgia Tech 1988	Catering	menus, planning episodes, menu failures	desired cuisine and constraints	a menu
PLEXUS	Univ. of Texas 1985	plan repair		execution-time failure, e.g. no BART cards in NY	modified plan, e.g. look for ticket seller

Name	Developer	Domain	Case	Input	Output
Runner	Univ. of Chicago 1989	everyday activities	standard activity sequences in memory hierarchy	goals, e.g. have breakfast and object-level visual features	actions, opportunistic optimization of conjunctive goals
TA	Univ. of Mass in progress	automatic program generation	programs, patches, and debugging episodes	program specification in logical form	lisp code
ROENTGEN	Univ. of Chicago 1989	radiation therapy	beam plan for lung cancer therapy	2D rep. of patient cross section, plus constraints and planned dosages	list of beams, strengths, orientations, etc.
CHEF	Yale 1986	cooking	recipes	desired ingredients and tastes	a recipe
BURN Sizer	DEC 1988	determining computer resource needs	organizing prototypes	description of origination (size, type of business, predicted computing user community, desired applications, etc.)	hardware/softw-are recommendatio-ns
TRUCKER	Univ. of Chicago 1988	previous routes	UPS-like delivery	delivery requests, map of the area	schedule of truck deliveries, library of "good" routes
SUPERMOM	Georgia Tech in progress	scheduling house hold tasks	previous experiences achieving tasks individually and in tandem	set of tasks that need to be done, state of the world	schedule that achieves tasks
SEPARATOR	Cognitive Systems, Inc. 1988	bank telex classification	telexes and their correct classifications	free-form unrestricted bank telexes	classification into one of 13 categories

CIFE CENTER FOR INTEGRATED FACILITY ENGINEERING

An Approach to Automated Architectural Floor Layout Generation with Case-based Reasoning Part II

CBRE Lisp Code Lists

by

Yoshihiro Ichioka

TECHNICAL REPORT

Number 44 B

March, 1991

Stanford University

If you would like to contact the authors please write to:

c/o CIFE, Civil Engineering,
Stanford University,
Terman Engineering Center
Mail Code: 4020
Stanford, CA 95305-4020

An Approach to Automated Architectural Floor Layout Generation with Case-based Reasoning

CBRE Lisp Code Lists

by

Yoshihiro Ichioka*

* Systems Engineer, Obayashi Corporation and Visiting Research Fellow, *Center for Integrated Facility Engineering*, Stanford University.

Welcome to CBRE : Case-based Reasoning Environment. The floppy disk, named CBRE, includes following three folders.

1. FRAMEKIT
2. CBRE.Lisp
3. CBR-Library

When you install CBRE in your Macintosh, there are some restrictions. Since CBRE uses some color functions, you should have Allegro Common Lisp Version 1. 3. 2. and a color monitor. CBRE needs more than 5 MB memories. So you should change Allegro Common Lisp memory size before running. You can easily change Allegro Common Lisp memory size with "Get Info" under "File" menu. I will recommend you to set more than 5 MB to reduce Garbage Collection.

FRAMEKIT is a frame-based knowledge representation, written in Common Lisp, that provides the basic mechanisms of frames, inheritance, demons, and views. Folder "FRAMEKIT" contains the functions that can provide above frame-based environment on Allegro Common Lisp. Folder "CBRE.Lisp" contains basic CBRE functions that can provide case-based reasoning environment. You can make your own case-based reasoning applications in Folder "CBR-Library", when you key-in the new file name with "File" under "CBR-Menu". Folder "CBR-Library" is referred by the function which is invoked by clicking "File" submenu item.

When you want to load CBRE into you Macintosh, Please follow next procedures.

1. Click Allegro Common Lisp icon double times to open Lisp environment.
2. Click "Eval" main menu item and draw your mouse below main menu and click "Load" submenu item. Then Macintosh File Allocation window will be popped up.
3. First click "Drive" radio-button and chose "CBRE" floppy disk. Second open Folder "CBRE.Lisp" and click File "CBRE.Load.Lisp" double times, then CBRE functions would be loaded from following twelve files just after FRAMEKIT environment are prepared.

(1) CBRE.Lisp.Framekit

(2) Expand.Lisp

(3) CBRE.Functions1

(4) CBRE.Functions2

(5) CBRE.Functions3

(6) CBRE.Functions4

(7) CBRE.Basic.MOPs

(8) CBRE.Floor.Defmop

(9) CBRE.Floor.Defun

(10) CBRE.Floor.Defun1

(11) CBRE.Floor.Output.Defmop

(12) CBRE.Floor.Output.Defun

If you want to contact the developer, please mail to

Yoshihiro Ichioka

CIFE, Civil Engineering,

Stanford University,

Terman Engineering Center

Mail Code: 4020

Stanford, CA 95305-4020

CBRE.Load.Lisp

```
;;*******************************************************************
;;*                                                                 *
;;*  When loading, you should change *CBRE-Directory* to refer to your *
;;*  current environment.                                           *
;;*                                    Feb. 14, 1991 Y.ICHIOKA      *
;;*******************************************************************
;;*  Requirements                                                   *
;;*  1. Allegro Common Lisp Version 1.3.2 (more than 5.0 MB memory)  *
;;*  2. Framekit Version 2.0                                        *
;;*******************************************************************
;;
(setq *CBRE-Directory* "ichioka:Languages:Allegro:CBRE:")
(setq *CBR-Library*
       (concatenate 'string *CBRE-Directory* "CBR-Library:"))
;
;***** load quickdraw *****
;
(require 'quickdraw)
;
;***** load Framekit *****
;
(load
       (concatenate 'string *CBRE-Directory*
                            "FRAMEKIT:framekit.system"))
;
;***** compile Framekit *****
;
(compile-framekit)
;
;***** load CBRE functions *****
;
(load
       (concatenate 'string *CBRE-Directory*
                            "CBRE.Lisp:CBRE.Framekit"))
(load
       (concatenate 'string *CBRE-Directory*
                            "CBRE.Lisp:Expand.Lisp"))
(load
       (concatenate 'string *CBRE-Directory*
                            "CBRE.Lisp:CBRE.Functions1"))
(load
       (concatenate 'string *CBRE-Directory*
                            "CBRE.Lisp:CBRE.Functions2"))
(load
       (concatenate 'string *CBRE-Directory*
                            "CBRE.Lisp:CBRE.Functions3"))
(load
       (concatenate 'string *CBRE-Directory*
                            "CBRE.Lisp:CBRE.Functions4"))
(load
       (concatenate 'string *CBRE-Directory*
                            "CBRE.Lisp:CBRE.Basic.MOPs"))
;
;***** show CBR main menu *****
;
(m-cbrmenu-fun)
;
;***** load residentail house example MOPs and functions *****
;
(load
       (concatenate 'string *CBRE-Directory*
                            "CBRE.Lisp:CBRE.Floor.Defmop"))
(load
       (concatenate 'string *CBRE-Directory*
                            "CBRE.Lisp:CBRE.Floor.Defun"))
(load
       (concatenate 'string *CBRE-Directory*
                            "CBRE.Lisp:CBRE.Floor.Defun1"))
;
;***** load MOPs and functions for graphical output *****
;
(load
       (concatenate 'string *CBRE-Directory*
                            "CBRE.Lisp:CBRE.Floor.Output.Defmop"))
(load
       (concatenate 'string *CBRE-Directory*
                            "CBRE.Lisp:CBRE.Floor.Output.Defun"))
```

CBRE.Framekit

```
;
;*** Listing 3.1:INSIST
;
(defmacro insist (fnname &rest exps)
 `(and ,@(make-insist-forms fnname exps)))
;
(defun make-insist-forms (fnname exps)
 (and (not (null exps))
      (cons `(or ,(car exps)
            (error "~s failed in ~s"
                   ',(car exps) ',fnname))
            (make-insist-forms fnname (cdr exps)))))
;
(defun sum-squares (x y)
   (insist sum-squares (numberp x) (numberp y))
   (+ (* x x) (* y y)))
;
;*** Listing 3.2:Definition of DEFINE-TABLE
;
(defmacro define-table (fn vars place)
  (let ((key (car vars))
        (set-fn (gentemp "set-fn."))
        (val (gentemp "val.")))
  `(progn (defun ,fn (,key)
              (declare (object-variable ,place))
              (getf ,place ,key))
          (defun ,set-fn (,key ,val)
              (declare (object-variable ,place))
              (setf (getf ,place ,key) ,val))
          (defsetf ,fn ,set-fn)
          ',fn)))
;
(defun delete-key (table key)
 (remf table key) table)
;
(defun table-keys (table)
 (and table
      (cons (car table)
      (table-keys (cdr (cdr table))))))
;
;*** Listing 3.3:Defenition of FOR
;
(setf *for-keys* nil)
(define-table for-key (key) *for-keys*)
;
(defmacro for (&rest for-clauses)
  (let ((when-part (member ':when for-clauses)))
    (for-expander (for-var-forms for-clauses)
                  (and when-part (car (cdr when-part)))
                  (for-body for-clauses))))
;
(defun for-var-forms (l)
 (and l (listp (car l))
      (cons (car l) (for-var-forms (cdr l)))))
;
(defun for-body (l)
  (and l (or (and (for-key (car l)) l)
             (for-body (cdr l)))))
;
(defun for-expander (var-forms when-form body-forms)
 (insist for
         (not (null var-forms))
         (not (null body-forms)))
  (let ((vars (mapcar #'car var-forms))
        (lists (mapcar #'(lambda (var-form)
                           (car (cdr (cdr var-form))))
                       var-forms))
        (mapfn-body (funcall (for-key (car body-forms))
                      when-form
                      `(progn ,@(cdr body-forms)))))
   `(,(car mapfn-body)
    #'(lambda ,vars ,(car (cdr mapfn-body)))
    ,@lists)))
;
(defmacro define-for-key (key vars mapfn body)
 `(progn (setf (for-key ',key)
               #'(lambda ,vars (list ,mapfn ,body)))
         ',key))
;
;*** Listing 3.4:Defenition of FOR Keywords
;
(define-for-key :always (test body)
 'every
 (cond (test `(or (not ,test) ,body)) (t body)))
;
(define-for-key :do (test body)
 'mapc (cond (test `(and ,test ,body)) (t body)))
;
(define-for-key :filter (test body)
 'mapcan
 (let ((fbody `(let ((x ,body)) (and x (list x)))))
   (cond (test `(and ,test ,fbody)) (t fbody))))
;
(define-for-key :first (test body)
 'some (cond (test `(and ,test ,body)) (t body)))
;
(define-for-key :save (test body)
  (cond (test 'mapcan) (t 'mapcar))
  (cond (test `(and ,test (list ,body)))
        (t body)))
;
(define-for-key :splice (test body)
 'mapcan
 `(copy-list
    ,(cond (test `(and ,test ,body)) (t body))))
;
;%%%%%%%%%%% table definition %%%%%%%%%%%
;
(defun mop-all-absts (mop)
       (let ((abst (get 'mop-all-absts mop)))
            (cond ((eql abst nil) (list mop))
                  (t abst))))
;
(defun mop-type (mop)
       (get 'mop-type mop))
;       (cond ((eql (instance-p mop) t) 'instance)
;             ((eql (leaf-instance-p mop 'common) nil) 'mop)))
;
(defun mop-slots (mop)
       (let ((slotnames (skip-this 'is-a (%frame-slot-names mop))))
                  (let ((slotnames (skip-this 'subclasses slotnames)))
                       (let ((slotnames (skip-this 'instances slotnames)))
                            (let ((slots (skip-this 'instance-of slotnames)))
                                 (for (slot :in (reverse slots))
                                :save (list slot
                                      (car (%view-fillers mop slot 'value 'common)))) ↵
↳)))))
;
;%%%%%%%%%%% dah definition %%%%%%%%%%%
;
;* date     * June 14,1990
;* function * print frame inherit structure (subclass & instance)
;* argument * frame name
;* var      * none
;
(defun dah (mop)
       (pprint (tree->list mop #'specs->list nil)))
;
;
(defun tree->list (mop fn visited)
       (cond ((member mop visited) (list mop))
             (t (setf visited (cons mop visited))
               `(,mop ,@(funcall fn mop visited)))))
;
(defun specs->list (mop visited)
       (cond ((member 'subclasses (%frame-slot-names mop))
              (for (spec :in (%view-fillers mop 'subclasses 'value 'common))
                         :save (tree->list spec #'specs->list visited)))
             ((member 'instances (%frame-slot-names mop))
              (for (spec :in (%view-fillers mop 'instances 'value 'common))
                         :save (tree->list spec #'specs->list visited)))))
;
;%%%%%%%%%%% dph definition %%%%%%%%%%%
;
;* date     * June 14,1990
;* function * print frame slots
;* argument * frame name
;* var      * none
;
(defun dph (mop)
       (pprint (tree->list mop #'slots->forms nil)))
;
(defun slots->forms (mop visited)
       (cond ((eql (frame-p mop) nil) nil)
             (t (let ((slotnames (skip-this 'is-a (%frame-slot-names mop))))
                     (let ((slotnames (skip-this 'subclasses slotnames)))
                          (let ((slotnames (skip-this 'instances slotnames)))
                               (let ((slots (skip-this 'instance-of slotnames)))
                                    (for (slot :in (reverse slots))
                                   :save (cons slot
                                               (mop->form (car (%view-fillers mop slot 'value ↲
↳'common))
                                          visited)))))))))
;
(defun mop->form (mop visited)
       (tree->list mop #'slots->forms visited))
;
(defun skip-this (atom l)
       (let ((result ()))
            (dolist (element l result)
                    (cond ((eql atom element))
                          (t (setf result (append result (list element))))))))
;
;*************************************************************************************** ↲
↳*
; (mop-absts x) = (%view-fillers x 'instance-of ...) or (%view-fillers x 'is-a      ...)
; (mop-specs x) = (%view-fillers x 'subclasses  ...) or (%view-fillers x 'instances ...)
;*************************************************************************************** ↲
↳*
```

```
;
(defun mop-absts (mop)
       (cond ((eql (instance-p mop) t) (%view-fillers mop 'instance-of 'value 'common))
             (t    (%view-fillers mop 'is-a 'value 'common))))
;
(defun mop-specs (mop)
       (cond ((eql (leaf-instance-p mop 'common) nil)
              (append (%view-fillers mop 'instances  'value 'common)
                      (%view-fillers mop 'subclasses 'value 'common)))
             (t nil)))
;
;***** Listing 3.6
;
(defun mopp (x)
       (cond ((eql x nil) nil)
             (t (or (numberp x) (and (symbolp x) (mop-type x))))))
;
(defun instance-mopp (x)
       (or (numberp x) (eql (instance-p x) t)))
;
(defun abst-mopp (x)
       (mopp x)
       (and (mopp x)
            (eql (mop-type x) 'mop)))
;
(defun abstp (abst spec)
       (or (eql abst spec)
           (member abst (mop-all-absts spec))))
;
(defun patternp (x) (abstp 'm-pattern x))
;
(defun groupp (x) (abstp 'm-group x))
;
;***** Listing 3.7
;
(defun slot-role (slot) (car slot))
(defun slot-filler (slot) (cadr slot))
;
(defun make-slot (role mop) (list role mop))
;
(defun role-slot (role x)
  (insist role-slot
          (or (mopp x) (listp x)))
  (assoc role
         (cond ((mopp x) (mop-slots x))
               (t x))))
;
(defun role-filler (role x)
      (slot-filler (role-slot role x)))
;
;%%%%%%%%%%% add-role-filler %%%%%%%%%%%
;
;* date     * June 20,1990
;* function * add slots (role filler) to frame
;* argument * role     : slot name
;           * mop      : frame name
;           * filler   : facet value
;* var      * none
;* example  *
;             (add-role-filler 'price 'm-flower '$4.50)
;
;
;            <before>                          <after>
;            (m-flower                         (m-flower
;                   (color  white)                          (color  white)
;                   (flavor sweet))                         (flavor sweet)
;                                                           (price  $4.50))
;
(defun add-role-filler (role mop filler)
      (insist add-role-filler
             (mopp mop) (null (role-filler role mop)))
             (format t "~&~s:~s <= ~s" mop role filler)
             (add-value mop role filler :demons t)
;             (dph mop)
             (setf (get 'mop-slots mop)
                   (cons (make-slot role filler) (mop-slots mop)))
      filler)
;
;***** Listing 3.8
;
(defun link-abst (spec abst)
       (insist link-abst (abst-mopp abst) (mopp spec)
                         (not (abstp spec abst)))
       (cond ((not (abstp abst spec))
              (setf (get 'mop-absts spec)
                   (cons abst (mop-absts spec)))
              (setf (get 'mop-specs abst)
                   (cons spec (mop-specs abst)))
              (let ((type (mop-type spec)))
                  (cond ((eql type 'mop) (add-view-filler abst 'subclasses  'value 'commo ⤸
⤷n spec)
                                         (add-view-filler spec 'is-a        'value 'commo ⤸
⤷n abst))
                        (t               (add-view-filler abst 'instances   'value 'commo ⤸
⤷n spec)
                                         (add-view-filler spec 'instance-of 'value 'commo ⤸
⤷n abst))))
              (redo-all-absts spec)))
       spec)
;
(defun unlink-abst (spec abst)
       (cond ((abstp abst spec)
              (setf (get 'mop-absts spec)
                    (remove abst (mop-absts spec)))
              (setf (get 'mop-specs abst)
                    (remove spec (mop-specs abst)))
              (let ((type (mop-type spec)))
                   (cond ((eql type 'mop) (erase-filler abst 'subclasses  'value  spec)
                                          (erase-filler spec 'is-a        'value  abst))
                         (t               (erase-filler abst 'instances   'value  spec)
                                          (erase-filler spec 'instance-of 'value  abst))) ⤸
⤷)
              (redo-all-absts spec)))
       spec)
;
(defun redo-all-absts (mop)
       (setf (get 'mop-all-absts mop)  (calc-all-absts mop))
       (for (spec :in (mop-specs mop))
            :do (redo-all-absts spec)))
;
(defun calc-all-absts (mop)
       (remove-duplicates
           (cons mop (for (abst :in (mop-absts mop))
                          :splice (mop-all-absts abst)))))
;
;***** Listing 3.9
;
```

```
(setf *new-mop-stack* nil)
(defun new-mop (name absts type slots)
       (declare (special        *new-mop-stack*
                                new-mop)
                (object-variable abst
                                slot))
       (insist new-mop
               (symbolp name)
               (for (abst :in absts) :always (mopp abst)))
       (or type (setf type (calc-type absts slots)))
       (or name (setf name (spec-name absts type)))
       (make-frame name)
       (setf (get 'mop-type name) type)
;;           (and slots (setf (get 'mop-slots name) slots))
       (for (abst :in absts)
                 :do (for (slot :in slots)
                                :do (add-filler name (slot-role slot) 'value (slot-fille
r slot)))
                     (link-abst name abst))
       (setf *new-mop-stack* (append *new-mop-stack* (list name)))
       name)
;
(defun calc-type (absts slots)
       (or (for (abst :in absts)
                :when (patternp abst)
                :first 'mop)
           (and (null slots) 'mop)
           (for (slot :in slots)
                :when (not (instance-mopp (slot-filler slot)))
                :first 'mop)
           'instance))
;
(defun spec-name (absts type)
       (gentemp (format nil (cond ((eql type 'mop) "~s.")
                                  (t "i-~s."))
                            (car absts))))
;
;***** Listing 3.10:Memory Management Functions
;
(defun clear-memory ()
      (new-mop 'm-root nil 'mop nil)
      (setf (get 'mop-all-absts 'm-root)
            (calc-all-absts 'm-root))
      'm-root)
;
;(defun all-mops () (table-keys (mop-table 'type)))
;
(defun remove-mop (name)
      (for (abst :in (mop-absts name))
      :do (unlink-abst name abst)))
;      (for (table-name :in (table-keys *mop-tables*))
;      :do (setf (mop-table table-name)
;             (delete-key (mop-table table-name)
;                         name))))
;
;***** Listing 3.11:Extended Role Access Functions
;
(defun inherit-filler (role mop)
       (for (abst :in (mop-all-absts mop))
                  :first (role-filler role abst)))
;
(defun get-filler (role mop)
       (or (role-filler role mop)
           (let ((filler (inherit-filler role mop)))
                (and filler
                     (or (and (instance-mopp filler) filler)
                         (and (abstp 'm-function filler) filler)
                         (let ((fn (get-filler 'calc-fn filler)))
                              (and fn (let ((new-filler (funcall fn filler mop)))
                                           (and new-filler
                                                (add-role-filler role mop new-filler))))) ⤦
⤥)))))
;
(defun path-filler (path mop)
       (and (for (role :in path)
                 :always (setf mop (get-filler role mop)))
            mop))
;
;***** Listing 3.12: Extended MOP Abstraction Predicates
;
(defun slots-abstp (mop slots)
       (and (abst-mopp mop)
            (not (null (mop-slots mop)))
            (for (slot :in (mop-slots mop))
                 :always (satisfiedp (slot-filler slot)
                                     (get-filler (slot-role slot) slots)
                                     slots))))
;
(defun satisfiedp (constraint filler slots)
       (cond ((null constraint))
             ((patternp constraint)
              (funcall (inherit-filler 'abst-fn constraint) constraint filler slots))
             ((abstp constraint filler))
             ((instance-mopp constraint) (null filler))
             (filler (slots-abstp constraint filler))
             (t nil)))
;
;***** Listing 3.13: MOP Equality Functions
;
(defun mop-includesp (mop1 mop2)
       (and (eql (mop-type mop1) (mop-type mop2))
            (for (slot :in (mop-slots mop2))
                 :always (eql (slot-filler slot)
                    (get-filler (slot-role slot) mop1)))
       mop1))
;
(defun mop-equalp (mop1 mop2)
       (and (mop-includesp mop2 mop1)
            (mop-includesp mop1 mop2)))
;
;%%%%%%%%%%% get-twin %%%%%%%%%%%
;
;* date     * June 26,1990
;* function * returns the first MOP in memory it can find that is equal to mop
;* argument * mop     : abstruct frame name
;* var      * none
;* example  *
;
(defun get-twin (mop)
       (for (abst :in (mop-absts mop))
                  :first (for (spec :in (mop-specs abst))
                                    :when (not (eql spec mop))
                                    :first (mop-equalp spec mop))))
;
;***** Listing 3.14: Refining Functions
;
```

```
(defun refine-instance (instance)
       (for (abst :in (mop-absts instance))
                    :when (mops-abstp (mop-specs abst) instance)
                    :first (unlink-abst instance abst)
                           (refine-instance instance)))
;
(defun mops-abstp (mops instance)
       (not (null (for (mop :in mops)
                             :when (slots-abstp mop instance)
                             :save (link-abst instance mop)))))
;
;***** Listing 3.15: Instance Installation Functions
;
(defun install-instance (instance)
       (refine-instance instance)
       (let ((twin (get-twin instance)))                           ;; checks if any instance ↘
↳in memory include this one
            (cond (twin (remove-mop instance) twin)                ;; when includes, it is re ↘
↳moved and returns old one
                  ((has-legal-absts-p instance) instance)          ;; illegal abstractions ar ↘
↳e unlinked and returns instance
                  (t (remove-mop instance) nil))))                 ;; instance is removed and ↘
↳ returns nil
;
(defun has-legal-absts-p (instance)
      (for (abst :in (mop-absts instance))
                 :when (not (legal-abstp abst instance))           ;; if immedeate abstractio ↘
↳n is illegal
                 :do (unlink-abst instance abst))                  ;; then it is unlinked
      (mop-absts instance))                                        ;; return the list of lleg ↘
↳al abstractions
;
(defun legal-abstp (abst instance)
       (declare (ignore instance)
                (object-variable spec))
       (and (mop-slots abst)                                       ;; has more than one slots
            (for (spec :in (mop-specs abst))                       ;; has a instance below it
                       :always (instance-mopp spec))))
;
;***** Listing 3.16: Abstraction Installation Functions
;
(defun install-abstraction (mop)
       (let ((twin (get-twin mop)))
            (cond (twin (remove-mop mop) twin)                         ;; if identical MOP e ↘
↳xsists, it is returned
                  (t (reindex-siblings mop)))))
;
(defun reindex-siblings (mop)
       (for (abst :in (mop-absts mop))                                 ;; for all immedeate ↘
↳abstractions
                  :do (for (spec :in (mop-specs abst))                 ;; for all immedeate ↘
↳specifications
                                 :when (and (instance-mopp spec)
                                            (slots-abstp mop spec))
                                 :do (unlink-abst spec abst)           ;; instances are unli ↘
↳nked from their abstractions
                                     (link-abst spec mop)))            ;; instances are link ↘
↳ed to mop
        mop)                                                           ;; mop is returned
;
;***** Listing 3.17: Top-Level Memory Update Functions
;
(defun slots->mop (slots absts must-work)
        (insist slots->mop
                (not (null absts))
                (for (abst :in absts)
                     :always (mopp abst)))
                (or (and (null slots) (null (cdr absts)) (car absts))
                    (let ((type (and slots (atom (car slots)) (car slots))))
                         (and type (setf slots (cdr slots)))
                         (let ((mop (new-mop nil absts type slots)))
                              (let ((result (cond ((instance-mopp mop) (install-instance m ⤸
⤷op))
                                                  (t (install-abstraction mop)))))
                              (insist slots->mop
                                      (or result (null must-work)))
                          result)))))
;
;***** Listing 3.18: Form to MOP Functions
;
(defmacro defmop (name absts &rest args)
          (let ((type (and args (atom (car args)) (car args))))
               (let ((slot-forms (cond (type (cdr args))
                                       (t args))))
                    `(new-mop ',name ',absts ',type
                              (forms->slots ',slot-forms)))))
;
(defun forms->slots (slot-forms)
       (for (slot-form :in slot-forms)
                 :save (cond ((atom slot-form) slot-form)
                             (t (make-slot (slot-role slot-form)
                                           (let ((abst (car (cdr slot-form))))
                                                (insist forms->slots (atom abst))
                                                (and abst (slots->mop
                                                           (forms->slots (cddr s ⤸
⤷lot-form))

                                                           (list abst) t))))))))
;
;***** Listing 3.19: Group MOP Functions
;
;%%%%%%%%%%% group-size %%%%%%%%%%%
;
;* date     * July 1,1990
;* function * returns the size of the group
;* argument * x        : abstruct frame name
;* var      * none
;
(defun group-size (x)
       (and (groupp x) (length (mop-slots x))))
;
;%%%%%%%%%%% group->list %%%%%%%%%%%
;
;* date     * July 1,1990
;* function * returns a list of the members of the group
;* argument * group    : name of the group
;* var      * none
;
(defun group->list (group)
       (and group
            (insist group->list (groupp group))
            (for (index :in (make-m-n 1 (group-size group)))
                      :filter (role-filler index group))))
;
;%%%%%%%%%%% list->group %%%%%%%%%%%
;
;* date     * July 1,1990
;* function * returns a group MOP with members from list
;* argument * list    :
;* var      * none
;
(defun list->group (l)
       (cond ((null l) 'i-m-empty-group)
             (t (slots->mop
                (for (x :in l)
                     (i :in (make-m-n 1 (length l)))
                       :save (make-slot i x))
             '(m-group)
             t))))
;
;%%%%%%%%%%% make-m-n %%%%%%%%%%%
;
;* date     * July 1,1990
;* function * generates a list of integers from m to n
;* argument * m       : start integer
;           * n       : end   integer
;* var      * none
;
(defun original-make-m-n (m n)
       (insist original-make-m-n (integerp m) (integerp n))
       (cond ((eql m n) (list n))
             ((< m n) (cons m (original-make-m-n (+ m 1) n)))
             (t (cons m (original-make-m-n (- m 1) n)))))
;
(setf *group-name* nil)
(setf (get *group-name* '1)  'g1)
(setf (get *group-name* '2)  'g2)
(setf (get *group-name* '3)  'g3)
(setf (get *group-name* '4)  'g4)
(setf (get *group-name* '5)  'g5)
(setf (get *group-name* '6)  'g6)
(setf (get *group-name* '7)  'g7)
(setf (get *group-name* '8)  'g8)
(setf (get *group-name* '9)  'g9)
(setf (get *group-name* '10) 'g10)
(setf (get *group-name* '11) 'g11)
(setf (get *group-name* '12) 'g12)
(setf (get *group-name* '13) 'g13)
(setf (get *group-name* '14) 'g14)
(setf (get *group-name* '15) 'g15)
(setf (get *group-name* '16) 'g16)
(setf (get *group-name* '17) 'g17)
(setf (get *group-name* '18) 'g18)
(setf (get *group-name* '19) 'g19)
(setf (get *group-name* '20) 'g20)
(setf (get *group-name* 'g1) '1)
(setf (get *group-name* 'g2) '2)
(setf (get *group-name* 'g3) '3)
(setf (get *group-name* 'g4) '4)
(setf (get *group-name* 'g5) '5)
(setf (get *group-name* 'g6) '6)
(setf (get *group-name* 'g7) '7)
(setf (get *group-name* 'g8) '8)
(setf (get *group-name* 'g9) '9)
(setf (get *group-name* 'g10) '10)
(setf (get *group-name* 'g11) '11)
(setf (get *group-name* 'g12) '12)
(setf (get *group-name* 'g13) '13)
(setf (get *group-name* 'g14) '14)
(setf (get *group-name* 'g15) '15)
```

```
(setf (get *group-name* 'g16) '16)
(setf (get *group-name* 'g17) '17)
(setf (get *group-name* 'g18) '18)
(setf (get *group-name* 'g19) '19)
(setf (get *group-name* 'g20) '20)
;
(defun make-m-n (m n)
       (declare (special         group-name
                                 make-m-n)
                (object-variable numlist))
       (insist make-m-n (integerp m) (integerp n))
       (cond ((> n 20) (format nil "*** error group name is more than 20 ***"))
             (t
                (let ((num-list (original-make-m-n m n)))
                     (for (num :in num-list)
                               :save (get *group-name* num))))))
;
;*** Listing 3.22:Basic Abstraction and Calculation Functions
;
(defun constraint-fn (constraint filler slots)
       (declare (ignore constraint
                        filler
                        slots)) t)
;
(defun not-constraint (constraint filler slots)
       (insist not-constraint (not (null filler)))
       (not (satisfiedp (get-filler 'object constraint)
                    filler slots)))
;
(defun get-sibling (pattern mop)
       (declare (ignore pattern)
                (object-variable abst
                                 spec))
       (for (abst :in (mop-absts mop))
                  :first (for (spec :in (mop-specs abst))
                  :when (and (instance-mopp spec)
                             (not (eql spec mop))
                             (not (abstp 'm-failed-solution spec)))
                  :first spec)))
;
```

Expand.Lisp

```
;;
;;*** Listing A.1:Defenition of BACKQUOTE
;;
(defun backquote (skel)
  (cond ((null skel) nil)
        ((atom skel) (list 'quote skel))
        ((eql (car skel) '$) (car (cdr skel)))
        ((and (listp (car skel))
              (eql (car (car skel)) '$$))
         (list 'append (car (cdr (car skel)))
               (backquote (cdr skel))))
        (t (combine-skels (backquote (car skel))
                          (backquote (cdr skel))))))
;;
(defun combine-skels (left right)
  (cond ((and (constantp left) (constantp right))
         (list 'quote (cons (car (cdr left))
                            (car (cdr right)))))
        ((null right) (list 'list left))
        ((and (listp right)
              (eql (car right) 'list))
         (cons 'list (cons left (cdr right))))
        (t (list 'cons left right))))
;;
;;*** LIsting A.2:Defenition for Backquote (`) Readmacro
;;
(set-macro-character #\`
    #'(lambda (stream char)
        (backquote (read stream t nil t))))
;;
(set-macro-character #\,
    #'(lambda (stream char)
        (list (get-unquoter stream)
              (read stream t nil t))))
;;
(defun get-unquoter (stream)
 (let ((next-char (read-char stream t nil t)))
  (cond ((eql next-char #\@) '$$)
        (t (unread-char next-char stream)
           '$))))
;;
```

CBRE.Functions1

```
;************************************************************************
;                       Define Object
;************************************************************************
;* date     : July 24,1990
;* function : defines *menu-function* subclass under *menu*
;* argument : none
;* var      : none
;************************************************************************
;
(defobject *cbrmenu-function* *menu*)
;
(setq cbrmenu-fun   (oneof  *cbrmenu-function*
                            :menu-title     "CBR-Menu"))
;
;************************************************************************
;                      MOP-File functions
;************************************************************************
;* date     : July 25,1990
;* function : allocs and updates MOP defenition file
;* argument : none
;* var      : none
;************************************************************************
;
(setq mop-file      (oneof  *cbrmenu-function*
                            :menu-title   "MOP-File"))
;
(setq new           (oneof  *menu-item*
                            :menu-item-title   "New"
                            :menu-item-action  #'(lambda ()
                                                    (file-alloc)
                                                    (ask mop-define   (menu-item-enab ↲
↳le))
                                                    (ask mop-utility  (menu-item-enab ↲
↳le))
                                                    (ask mop-edit     (menu-item-enab ↲
↳le)))))
;
(setq update        (oneof. *menu-item*
                            :menu-item-title   "Update"
                            :menu-item-action  #'(lambda ()
                                                    (file-update)
                                                    (ask mop-define   (menu-item-enab ↲
↳le))
                                                    (ask mop-utility  (menu-item-enab ↲
↳le))
                                                    (ask mop-edit     (menu-item-enab ↲
↳le)))))
;
(setq cbrmenu-l1    (oneof  *menu-item*                  ;; Blank line
                            :menu-item-title   "-"))
;
;************************************************************************
;                      MOP defenition
;************************************************************************
;* date     : July 25,1990
;* function : defines MOP using defmop macro and stores them to the file
;* argument : mop name
;* var      : none
;************************************************************************
;
(setq mop-define    (oneof  *menu-item*
                            :menu-item-title   "MOP-Define"
                            :menu-item-action  #'(lambda ()
                                                      (pop-up-input.1)
                                                      (input-dialog.1))
                             :disabled          t))
;
(setq cbrmenu-12     (oneof  *menu-item*                 ;; Blank line
                             :menu-item-title   "-"))
;
;*****************************************************************************
;                        calls display functions
;*****************************************************************************
;* date     : July 25,1990
;* function : shows display function menu
;* argument : none
;* var      : none
;*****************************************************************************
;
;
(setq mop-utility    (oneof   *menu-item*
                              :menu-item-title  "MOP-Utility"
                              :menu-item-action #'(lambda ()
                                                      (m-display-fun))
                              :disabled         t))
;
(setq cbrmenu-13     (oneof   *menu-item*                 ;; Blank line
                                                                              ⤦
⤥
                              :menu-item-title   "-"))
;
;
;*****************************************************************************
;                       edit MOP defenition file
;*****************************************************************************
;* date     : July 25,1990
;* function : shows edit window of MOP defenition file
;* argument : none
;* var      : none
;*****************************************************************************
;
;
(setq mop-edit       (oneof   *menu-item*
                              :menu-item-title  "MOP-Edit"
                              :menu-item-action #'(lambda ()
                                                      (setq cbr-edit (oneof *fred-wind ⤦
⤥ow*
                                                  :filename     *cbr-defmop-file*
                                                  :scratch-p    nil)))
                              :disabled         t))
;
(setq cbrmenu-14     (oneof   *menu-item*                 ;; Blank line
                                                                              ⤦
⤥
                              :menu-item-title   "-"))
;
;*****************************************************************************
;                       quit CBR-Menu
;*****************************************************************************
;* date     : July 25,1990
;* function : removes menu items and returns previous menu
;* argument : none
;* var      : none
;*****************************************************************************
;
(setq cbrmenu-quit   (oneof  *menu-item*
                             :menu-item-title   "Quit-CBR"
                             :menu-item-action  #'(lambda ()
                                                        (ask cbrmenu-fun (remove-menu-items mo ⤵
↳p-file))
                                                        (ask cbrmenu-fun (remove-menu-items cb ⤵
↳rmenu-l1))
                                                        (ask cbrmenu-fun (remove-menu-items mo ⤵
↳p-define))
                                                        (ask cbrmenu-fun (remove-menu-items cb ⤵
↳rmenu-l2))
                                                        (ask cbrmenu-fun (remove-menu-items mo ⤵
↳p-utility))
                                                        (ask cbrmenu-fun (remove-menu-items cb ⤵
↳rmenu-l3))
                                                        (ask cbrmenu-fun (remove-menu-items cb ⤵
↳rmenu-quit))
                                                        (set-menubar *menu-no-cbrmenu*))))
;
;*****************************************************************************
;                      set CBR-Menu functions
;*****************************************************************************
;* date     : July 25,1990
;* function : sets CBR-Menu menu functions
;* argument : none
;* var      : none
;*****************************************************************************
;
(defun m-cbrmenu-fun ()
      (declare (special *menu-no-cbrmenu*
                        cbrmenu-fun
                        mop-file
                        cbrmenu-l1
                        mop-define
                        cbrmenu-l2
                        mop-utility
                        cbrmenu-l3
                        mop-edit
                        cbrmenu-l4
                        cbrmenu-quit
                        new
                        update))
      (setq *menu-no-cbrmenu* (menubar))                                                   ⤵
↳
      (ask cbrmenu-fun   (menu-deinstall))
      (ask cbrmenu-fun   (menu-install))
      (ask cbrmenu-fun   (add-menu-items mop-file
                                         cbrmenu-l1
                                         mop-define
                                         cbrmenu-l2
                                         mop-utility
                                         cbrmenu-l3
                                         mop-edit
                                         cbrmenu-l4
                                         cbrmenu-quit))
      (ask mop-file      (add-menu-items new
                                         update)))
;
;*****************************************************************************
;                      Input dialog defenition no.1
;*****************************************************************************
;* date     * July 22,1990
;* function * define input dialog menu no.1
;* argument * none
;* var      * none
;*****************************************************************************
```

```
 ;
 (defun pop-up-input.1 ()
        (declare (special *mop-or-instance*
                          *input-dialog.1*))
        (setf *mop-or-instance* 'mop)
        (setf *input-dialog.1* (oneof *dialog*
                     :window-type       :document-with-grow
                     :window-title      "MOP Defenition"
                     :window-font       '("courie" 12)
                     :window-position   '(:top 40)
                     :window-size       #@(500 400))))
 ;
 (defun input-dialog.1 ()
        (declare (special *input-dialog.1*))
        (ask *input-dialog.1*
          (add-dialog-items    (oneof *static-text-dialog-item*
                                      :dialog-item-position     #@( 20 30)
                                      :dialog-item-font         '("courie" 12)
                                      :dialog-item-text         "Define abstract hierarchy ⤵
⤷")
                          (have 'save
                                (oneof *button-dialog-item*
                                       :dialog-item-position     #@(250  10)
                                       :dialog-item-text         " Save "
                                       :dialog-item-action
                                        #'(lambda ()
                                                  (make-defmop.1)
                                                  (ask *input-dialog.1* (window-close))
                                                  (pop-up-input.1)
                                                  (input-dialog.1))
                                       :default-button           t))
                          (have 'more
                                (oneof *button-dialog-item*
                                       :dialog-item-position     #@(330  10)
                                       :dialog-item-text         " More "
                                       :dialog-item-action
                                        #'(lambda ()
                                           (setf mop-name  (read-from-string (ask (ask *inpu ⤵
⤷t-dialog.1* mop-name)
                                                           (dialog-item-text))))
                                           (setf abst-name (read-from-string (ask (ask *inpu ⤵
⤷t-dialog.1* abst-name)
                                                           (dialog-item-text))))
                                           (pop-up-input.2)
                                           (input-dialog.2))
                                       :default-button           nil))
                          (have 'cancel
                                (oneof *button-dialog-item*
                                       :dialog-item-position     #@(410  10)
                                       :dialog-item-text         "Cancel"
                                       :dialog-item-action       #'(lambda ()
                                                                     (ask *input-dial ⤵
⤷og.1* (window-close))
                                                                     (pop-up-input.1)
                                                                     (input-dialog.1) ⤵
⤷)
                                       :default-button           nil))
 ;
 ;     MOP name and abstract name
 ;
                                (oneof *static-text-dialog-item*
                                       :dialog-item-position     #@( 30  70)
                                       :dialog-item-font         '("monaco" 12)
                                       :dialog-item-text         "MOP Name      :")
                           (have 'mop-name
                                 (oneof *editable-text-dialog-item*
                                       :dialog-item-position     #@(150  70)
                                       :dialog-item-size         #@(150  20)
                                       :dialog-item-font         '("monaco" 12)))
                                 (oneof *static-text-dialog-item*
                                       :dialog-item-position     #@( 30 110)
                                       :dialog-item-font         '("monaco" 12)
                                       :dialog-item-text         "Abstract Name :")
                           (have 'abst-name
                                 (oneof *editable-text-dialog-item*
                                       :dialog-item-position     #@(150 110)
                                       :dialog-item-size         #@(150  20)
                                       :dialog-item-font         '("monaco" 12)))
;
;     MOP or Instance (default : MOP)
;
                           (have 'mop
                                 (oneof *radio-button-dialog-item*
                                       :dialog-item-position     #@(400  70)
                                       :dialog-item-text         "MOP"
                                       :dialog-item-action       #'(lambda ()
                                                                      (setf *mop-or-instance ↩
↪* 'mop)
                                                                      (radio-button-push))
                                       :radio-button-cluster     :mop-or-instatnce
                                       :radio-button-pushed-p    t))
                           (have 'instance
                                 (oneof *radio-button-dialog-item*
                                       :dialog-item-position     #@(400  90)
                                       :dialog-item-text         "Instance"
                                       :dialog-item-action       #'(lambda ()
                                                                      (setf *mop-or-instance ↩
↪* 'instance)
                                                                      (radio-button-push))
                                       :radio-button-cluster     :mop-or-instatnce
                                       :radio-button-pushed-p    nil))
;
;     Define slots
;
                                 (oneof *static-text-dialog-item*
                                       :dialog-item-position     #@( 20 150)
                                       :dialog-item-font         '("courie" 12)
                                       :dialog-item-text         "Define slots")
                                 (oneof *static-text-dialog-item*
                                       :dialog-item-position     #@( 50 180)
                                       :dialog-item-font         '("monaco" 12)
                                       :dialog-item-text         "Slot-Role")
                                 (oneof *static-text-dialog-item*
                                       :dialog-item-position     #@(250 180)
                                       :dialog-item-font         '("monaco" 12)
                                       :dialog-item-text         "Slot-Filler")
;
                                 (oneof *static-text-dialog-item*
                                       :dialog-item-position     #@( 30 200)
                                       :dialog-item-font         '("monaco" 12)
                                       :dialog-item-text         "1.")
                           (have 'role1
                                 (oneof *editable-text-dialog-item*
                                       :dialog-item-position     #@( 50 200)
                                       :dialog-item-size         #@(150  20)
                                       :dialog-item-text         "nil"
                                 :dialog-item-font          '("monaco" 12)))
                          (oneof *static-text-dialog-item*
                                 :dialog-item-position      #@(220 200)
                                 :dialog-item-font          '("monaco" 12)
                                 :dialog-item-text          ":")
                    (have 'filler1
                          (oneof *editable-text-dialog-item*
                                 :dialog-item-position      #@(250 200)
                                 :dialog-item-size          #@(220  20)
                                 :dialog-item-text          "nil"
                                 :dialog-item-font          '("monaco" 12)))
;
                    (oneof *static-text-dialog-item*
                                 :dialog-item-position      #@( 30 240)
                                 :dialog-item-font          '("monaco" 12)
                                 :dialog-item-text          "2.")
                    (have 'role2
                          (oneof *editable-text-dialog-item*
                                 :dialog-item-position      #@( 50 240)
                                 :dialog-item-size          #@(150  20)
                                 :dialog-item-text          "nil"
                                 :dialog-item-font          '("monaco" 12)))
                          (oneof *static-text-dialog-item*
                                 :dialog-item-position      #@(220 240)
                                 :dialog-item-font          '("monaco" 12)
                                 :dialog-item-text          ":")
                    (have 'filler2
                          (oneof *editable-text-dialog-item*
                                 :dialog-item-position      #@(250 240)
                                 :dialog-item-size          #@(220  20)
                                 :dialog-item-text          "nil"
                                 :dialog-item-font          '("monaco" 12)))
;
                    (oneof *static-text-dialog-item*
                                 :dialog-item-position      #@( 30 280)
                                 :dialog-item-font          '("monaco" 12)
                                 :dialog-item-text          "3.")
                    (have 'role3
                          (oneof *editable-text-dialog-item*
                                 :dialog-item-position      #@( 50 280)
                                 :dialog-item-size          #@(150  20)
                                 :dialog-item-text          "nil"
                                 :dialog-item-font          '("monaco" 12)))
                          (oneof *static-text-dialog-item*
                                 :dialog-item-position      #@(220 280)
                                 :dialog-item-font          '("monaco" 12)
                                 :dialog-item-text          ":")
                    (have 'filler3
                          (oneof *editable-text-dialog-item*
                                 :dialog-item-position      #@(250 280)
                                 :dialog-item-size          #@(220  20)
                                 :dialog-item-text          "nil"
                                 :dialog-item-font          '("monaco" 12)))
;
                    (oneof *static-text-dialog-item*
                                 :dialog-item-position      #@( 30 320)
                                 :dialog-item-font          '("monaco" 12)
                                 :dialog-item-text          "4.")
                    (have 'role4
                          (oneof *editable-text-dialog-item*
                                 :dialog-item-position      #@( 50 320)
                                 :dialog-item-size          #@(150  20)
                                 :dialog-item-text          "nil"
```

```
                                :dialog-item-font          '("monaco" 12)))
                          (oneof *static-text-dialog-item*
                                :dialog-item-position      #@(220 320)
                                :dialog-item-font          '("monaco" 12)
                                :dialog-item-text          ":")
                    (have 'filler4
                          (oneof *editable-text-dialog-item*
                                :dialog-item-position      #@(250 320)
                                :dialog-item-size          #@(220  20)
                                :dialog-item-text          "nil"
                                :dialog-item-font          '("monaco" 12)))
;
                    (oneof *static-text-dialog-item*
                                :dialog-item-position      #@( 30 360)
                                :dialog-item-font          '("monaco" 12)
                                :dialog-item-text          "5.")
                    (have 'role5
                          (oneof *editable-text-dialog-item*
                                :dialog-item-position      #@( 50 360)
                                :dialog-item-size          #@(150  20)
                                :dialog-item-text          "nil"
                                :dialog-item-font          '("monaco" 12)))
                          (oneof *static-text-dialog-item*
                                :dialog-item-position      #@(220 360)
                                :dialog-item-font          '("monaco" 12)
                                :dialog-item-text          ":")
                    (have 'filler5
                          (oneof *editable-text-dialog-item*
                                :dialog-item-position      #@(250 360)
                                :dialog-item-size          #@(220  20)
                                :dialog-item-text          "nil"
                                :dialog-item-font          '("monaco" 12)))

                    )))
;
;***************************************************************************
;                    Input dialog defenition no.2
;***************************************************************************
;* date     * July 22,1990
;* function * define input dialog menu no.2
;* argument * none
;* var      * none
;***************************************************************************
;
(defun pop-up-input.2 ()
       (declare (special *input-dialog.2*))
       (setf *input-dialog.2* (oneof *dialog*
                    :window-type      :document-with-grow
                    :window-title     "MOP Defenition"
                    :window-font      '("courie" 12)
                    :window-position  '(:top 40)
                    :window-size      #@(500 400))))
;
(defun input-dialog.2 ()
       (declare (special *input-dialog.2*
                          mop-name
                          abst-name))
       (ask *input-dialog.2*
          (add-dialog-items   (oneof *static-text-dialog-item*
                                :dialog-item-position      #@( 20 30)
                                :dialog-item-font          '("courie" 12)
                                :dialog-item-text          "Define abstract hierarchy ⤸
⤷")
```

```
                              (have 'save
                                    (oneof *button-dialog-item*
                                           :dialog-item-position     #@(250  10)
                                           :dialog-item-text         " Save "
                                           :dialog-item-action
                                            #'(lambda ()
                                                       (make-defmop.2)
                                                       (ask *input-dialog.1* (window-close))
                                                       (ask *input-dialog.2* (window-close))
                                                       (pop-up-input.1)
                                                       (input-dialog.1))
                                           :default-button           t))
                              (have 'more
                                    (oneof *button-dialog-item*
                                           :dialog-item-position     #@(330  10)
                                           :dialog-item-text         " More "
                                           :dialog-item-enabled-p    nil
                                           :default-button           t))
                              (have 'cancel
                                    (oneof *button-dialog-item*
                                           :dialog-item-position     #@(410  10)
                                           :dialog-item-text         "Cancel"
                                           :dialog-item-action       #'(lambda ()
                                                                              (ask *input-dial ↘
↳og.2* (window-close))
                                                                              (pop-up-input.2)
                                                                              (input-dialog.2) ↘
↳)
                                           :default-button           nil))
;
;     MOP name and abstract name
;
                                    (oneof *static-text-dialog-item*
                                           :dialog-item-position     #@( 30  70)
                                           :dialog-item-font         '("monaco" 12)
                                           :dialog-item-text         "MOP Name      :")
                                    (oneof *editable-text-dialog-item*
                                           :dialog-item-position     #@(150  70)
                                           :dialog-item-size         #@(150  20)
                                           :dialog-item-font         '("monaco" 12)
                                           :dialog-item-text         (string mop-name))
                                    (oneof *static-text-dialog-item*
                                           :dialog-item-position     #@( 30 110)
                                           :dialog-item-font         '("monaco" 12)
                                           :dialog-item-text         "Abstract Name :")
                                    (oneof *editable-text-dialog-item*
                                           :dialog-item-position     #@(150 110)
                                           :dialog-item-size         #@(150  20)
                                           :dialog-item-font         '("monaco" 12)
                                           :dialog-item-text         (string abst-name))
;
;     MOP or Instance (default : MOP)
;
                              (have 'mop
                                    (oneof *radio-button-dialog-item*
                                           :dialog-item-position     #@(400  70)
                                           :dialog-item-text         "MOP"
                                           :dialog-item-enabled-p    nil
                                           :radio-button-cluster     :mop-or-instatnce
                                           :radio-button-pushed-p    nil))
                              (have 'instance
                                    (oneof *radio-button-dialog-item*
                                           :dialog-item-position     #@(400  90)
```

```
                                  :dialog-item-text         "Instance"
                                  :dialog-item-enabled-p    nil
                                  :radio-button-cluster     :mop-or-instatnce
                                  :radio-button-pushed-p    nil))
;
;     Define slots
;
                            (oneof *static-text-dialog-item*
                                  :dialog-item-position     #@( 20 150)
                                  :dialog-item-font         '("courie" 12)
                                  :dialog-item-text         "Define slots")
                            (oneof *static-text-dialog-item*
                                  :dialog-item-position     #@( 50 180)
                                  :dialog-item-font         '("monaco" 12)
                                  :dialog-item-text         "Slot-Role")
                            (oneof *static-text-dialog-item*
                                  :dialog-item-position     #@(250 180)
                                  :dialog-item-font         '("monaco" 12)
                                  :dialog-item-text         "Slot-Filler")
;
                            (oneof *static-text-dialog-item*
                                  :dialog-item-position     #@( 30 200)
                                  :dialog-item-font         '("monaco" 12)
                                  :dialog-item-text         "6.")
                       (have 'role6
                            (oneof *editable-text-dialog-item*
                                  :dialog-item-position     #@( 50 200)
                                  :dialog-item-size         #@(150  20)
                                  :dialog-item-text         "nil"
                                  :dialog-item-font         '("monaco" 12)))
                            (oneof *static-text-dialog-item*
                                  :dialog-item-position     #@(220 200)
                                  :dialog-item-font         '("monaco" 12)
                                  :dialog-item-text         ":")
                       (have 'filler6
                            (oneof *editable-text-dialog-item*
                                  :dialog-item-position     #@(250 200)
                                  :dialog-item-size         #@(220  20)
                                  :dialog-item-text         "nil"
                                  :dialog-item-font         '("monaco" 12)))
;
                       (oneof *static-text-dialog-item*
                                  :dialog-item-position     #@( 30 240)
                                  :dialog-item-font         '("monaco" 12)
                                  :dialog-item-text         "7.")
                       (have 'role7
                            (oneof *editable-text-dialog-item*
                                  :dialog-item-position     #@( 50 240)
                                  :dialog-item-size         #@(150  20)
                                  :dialog-item-text         "nil"
                                  :dialog-item-font         '("monaco" 12)))
                            (oneof *static-text-dialog-item*
                                  :dialog-item-position     #@(220 240)
                                  :dialog-item-font         '("monaco" 12)
                                  :dialog-item-text         ":")
                       (have 'filler7
                            (oneof *editable-text-dialog-item*
                                  :dialog-item-position     #@(250 240)
                                  :dialog-item-size         #@(220  20)
                                  :dialog-item-text         "nil"
                                  :dialog-item-font         '("monaco" 12)))
;
                       (oneof *static-text-dialog-item*
                              :dialog-item-position      #@( 30 280)
                              :dialog-item-font          '("monaco" 12)
                              :dialog-item-text          "8.")
              (have 'role8
                    (oneof *editable-text-dialog-item*
                              :dialog-item-position      #@( 50 280)
                              :dialog-item-size          #@(150  20)
                              :dialog-item-text          "nil"
                              :dialog-item-font          '("monaco" 12)))
                    (oneof *static-text-dialog-item*
                              :dialog-item-position      #@(220 280)
                              :dialog-item-font          '("monaco" 12)
                              :dialog-item-text          ":")
              (have 'filler8
                    (oneof *editable-text-dialog-item*
                              :dialog-item-position      #@(250 280)
                              :dialog-item-size          #@(220  20)
                              :dialog-item-text          "nil"
                              :dialog-item-font          '("monaco" 12)))
;
              (oneof *static-text-dialog-item*
                              :dialog-item-position      #@( 30 320)
                              :dialog-item-font          '("monaco" 12)
                              :dialog-item-text          "9.")
              (have 'role9
                    (oneof *editable-text-dialog-item*
                              :dialog-item-position      #@( 50 320)
                              :dialog-item-size          #@(150  20)
                              :dialog-item-text          "nil"
                              :dialog-item-font          '("monaco" 12)))
                    (oneof *static-text-dialog-item*
                              :dialog-item-position      #@(220 320)
                              :dialog-item-font          '("monaco" 12)
                              :dialog-item-text          ":")
              (have 'filler9
                    (oneof *editable-text-dialog-item*
                              :dialog-item-position      #@(250 320)
                              :dialog-item-size          #@(220  20)
                              :dialog-item-text          "nil"
                              :dialog-item-font          '("monaco" 12)))
;
              (oneof *static-text-dialog-item*
                              :dialog-item-position      #@( 22 360)
                              :dialog-item-font          '("monaco" 12)
                              :dialog-item-text          "10.")
              (have 'role10
                    (oneof *editable-text-dialog-item*
                              :dialog-item-position      #@( 50 360)
                              :dialog-item-size          #@(150  20)
                              :dialog-item-text          "nil"
                              :dialog-item-font          '("monaco" 12)))
                    (oneof *static-text-dialog-item*
                              :dialog-item-position      #@(220 360)
                              :dialog-item-font          '("monaco" 12)
                              :dialog-item-text          ":")
              (have 'filler10
                    (oneof *editable-text-dialog-item*
                              :dialog-item-position      #@(250 360)
                              :dialog-item-size          #@(220  20)
                              :dialog-item-text          "nil"
                              :dialog-item-font          '("monaco" 12))))))

;
```

```
;     Make defmop lisp expression with 1 input-dialog menu
;
(defun make-defmop.1 ()
       (declare (special         *slot-set*
                                 *input-dialog.1*
                                 *cbr-defmop-file*
                                 *mop-or-instance*)
                (object-variable mop-name
                                 abst-name
                                 role1
                                 role2
                                 role3
                                 role4
                                 role5
                                 filler1
                                 filler2
                                 filler3
                                 filler4
                                 filler5))
       (setf *print-pretty* t)
       (setf *slot-set* nil)
       (let ((mop-name  (read-from-string (ask (ask *input-dialog.1* mop-name)  (dialog-i ↩
↪tem-text))))
             (abst-name (read-from-string (ask (ask *input-dialog.1* abst-name) (dialog-i ↩
↪tem-text))))
             (role1     (read-from-string (ask (ask *input-dialog.1* role1)     (dialog-i ↩
↪tem-text))))
             (role2     (read-from-string (ask (ask *input-dialog.1* role2)     (dialog-i ↩
↪tem-text))))
             (role3     (read-from-string (ask (ask *input-dialog.1* role3)     (dialog-i ↩
↪tem-text))))
             (role4     (read-from-string (ask (ask *input-dialog.1* role4)     (dialog-i ↩
↪tem-text))))
             (role5     (read-from-string (ask (ask *input-dialog.1* role5)     (dialog-i ↩
↪tem-text))))
             (filler1   (read-from-string (ask (ask *input-dialog.1* filler1)   (dialog-i ↩
↪tem-text))))
             (filler2   (read-from-string (ask (ask *input-dialog.1* filler2)   (dialog-i ↩
↪tem-text))))
             (filler3   (read-from-string (ask (ask *input-dialog.1* filler3)   (dialog-i ↩
↪tem-text))))
             (filler4   (read-from-string (ask (ask *input-dialog.1* filler4)   (dialog-i ↩
↪tem-text))))
             (filler5   (read-from-string (ask (ask *input-dialog.1* filler5)   (dialog-i ↩
↪tem-text)))))
             (setf (get *slot-set* 'role1) filler1)
             (setf (get *slot-set* 'role2) filler2)
             (setf (get *slot-set* 'role3) filler3)
             (setf (get *slot-set* 'role4) filler4)
             (setf (get *slot-set* 'role5) filler5)
             (let ((role-list (list role1 role2 role3 role4 role5))
                   (filler-list (list filler1 filler2 filler3 filler4 filler5))
                   (slot-list nil))
                  (let ((n (length role-list)) (result nil))
                       (dotimes (count n result)
                                (cond ((eql (car role-list) nil))
                                      ( t
                                        (cond ((eql (string-equal (char (string (car fill ↩
↪er-list)) 0) "m") t)
                                               (format t "~%Does MOP : ~s have a slot? <y ↩
↪/n> ==> " (car filler-list))
                                               (let ((answer (read)))
                                                  (cond ((eql answer 'n)          ;; M ↩
¬OP wothout a slot
                                                        (setf slot-list (append slot-list ¬
¬
                                                                                    (list
                                                                                    (list (car ro ¬
¬le-list)
                                                                                    (car filler-l ¬
¬ist))))))
                                                        ((eql answer 'y)             ;; M ¬
¬OP with a slot
                                                         (format t "~%please input its sl ¬
¬ot ==> ")
                                                         (let ((its-slot (read)))
                                                            (setf slot-list (append slot- ¬
¬list
                                                                                    (list
                                                                                    (list (car ro ¬
¬le-list)
                                                                                    (car filler-l ¬
¬ist)
                                                                                    its-slot))))) ¬
¬))))
                                                (t (setf slot-list (append slot-list
                                                                                    (list
                                                                                    (list (car ro ¬
¬le-list)
                                                                                    (car filler-l ¬
¬ist)))))))
                                                   (setf role-list (cdr role-list))
                                                   (setf filler-list (cdr filler-list)))) ¬
¬))
                    (with-open-file (defmop-stream *cbr-defmop-file*
                                                  :direction :output
                                                  :if-exists :append)
                                    (print `(defmop ,mop-name ,(list abst-name) ,*mop-or-i ¬
¬nstance* ,@slot-list)
                                           defmop-stream))
                   )))
 ;
 ;     Make defmop lisp expression with 2 input-dialog menus
 ;
 (defun make-defmop.2 ()
       (declare (special        *slot-set*
                                *input-dialog.1*
                                *input-dialog.2*
                                *cbr-defmop-file*
                                *mop-or-instance*)
                (object-variable mop-name
                                abst-name
                                role1
                                role2
                                role3
                                role4
                                role5
                                role6
                                role7
                                role8
                                role9
                                role10
                                filler1
                                filler2
                                filler3
                                filler4
```

```
                                          filler5
                                          filler6
                                          filler7
                                          filler8
                                          filler9
                                          filler10
                                          role-list
                                          filler-list
                                          slot-list
                                          n
                                          result
                                          answer
                                          its-slot))
        (setf *print-pretty* t)
        (setf *slot-set* nil)
        (let ((role1     (read-from-string (ask (ask *input-dialog.1* role1)      (dialog-i ↩
↪tem-text))))
              (role2     (read-from-string (ask (ask *input-dialog.1* role2)      (dialog-i ↩
↪tem-text))))
              (role3     (read-from-string (ask (ask *input-dialog.1* role3)      (dialog-i ↩
↪tem-text))))
              (role4     (read-from-string (ask (ask *input-dialog.1* role4)      (dialog-i ↩
↪tem-text))))
              (role5     (read-from-string (ask (ask *input-dialog.1* role5)      (dialog-i ↩
↪tem-text))))
              (role6     (read-from-string (ask (ask *input-dialog.2* role6)      (dialog-i ↩
↪tem-text))))
              (role7     (read-from-string (ask (ask *input-dialog.2* role7)      (dialog-i ↩
↪tem-text))))
              (role8     (read-from-string (ask (ask *input-dialog.2* role8)      (dialog-i ↩
↪tem-text))))
              (role9     (read-from-string (ask (ask *input-dialog.2* role9)      (dialog-i ↩
↪tem-text))))
              (role10    (read-from-string (ask (ask *input-dialog.2* role10)     (dialog-i ↩
↪tem-text))))
              (filler1   (read-from-string (ask (ask *input-dialog.1* filler1)    (dialog-i ↩
↪tem-text))))
              (filler2   (read-from-string (ask (ask *input-dialog.1* filler2)    (dialog-i ↩
↪tem-text))))
              (filler3   (read-from-string (ask (ask *input-dialog.1* filler3)    (dialog-i ↩
↪tem-text))))
              (filler4   (read-from-string (ask (ask *input-dialog.1* filler4)    (dialog-i ↩
↪tem-text))))
              (filler5   (read-from-string (ask (ask *input-dialog.1* filler5)    (dialog-i ↩
↪tem-text))))
              (filler6   (read-from-string (ask (ask *input-dialog.2* filler6)    (dialog-i ↩
↪tem-text))))
              (filler7   (read-from-string (ask (ask *input-dialog.2* filler7)    (dialog-i ↩
↪tem-text))))
              (filler8   (read-from-string (ask (ask *input-dialog.2* filler8)    (dialog-i ↩
↪tem-text))))
              (filler9   (read-from-string (ask (ask *input-dialog.2* filler9)    (dialog-i ↩
↪tem-text))))
              (filler10  (read-from-string (ask (ask *input-dialog.2* filler10)   (dialog-i ↩
↪tem-text)))))
              (setf (get *slot-set* 'role1)  filler1)
              (setf (get *slot-set* 'role2)  filler2)
              (setf (get *slot-set* 'role3)  filler3)
              (setf (get *slot-set* 'role4)  filler4)
              (setf (get *slot-set* 'role5)  filler5)
              (setf (get *slot-set* 'role6)  filler6)
              (setf (get *slot-set* 'role7)  filler7)
              (setf (get *slot-set* 'role8)  filler8)
```

```
               (setf (get *slot-set* 'role9)  filler9)
               (setf (get *slot-set* 'role10) filler10)
               (let ((role-list (list role1 role2 role3 role4 role5
                                      role6 role7 role8 role9 role10))
                     (filler-list (list filler1 filler2 filler3 filler4 filler5
                                        filler6 filler7 filler8 filler9 filler10))
                     (slot-list nil))
                    (let ((n (length role-list)) (result nil))
                         (dotimes (count n result)
                                  (cond ((eql (car role-list) nil))
                                        ( t
                                          (cond ((eql (string-equal (char (string (car fill ↩
↪er-list)) 0) "m") t)
                                                 (format t "~%Does MOP : ~s have a slot? <y ↩
↪/n> ==> " (car filler-list))
                                                 (let ((answer (read))
                                                    (cond ((eql answer 'n)              ;; M ↩
↪OP wothout a slot
                                                          (setf slot-list (append slot-list ↩
↪
                                                                                (list
                                                                                (list (car ro ↩
↪le-list)
                                                                                (car filler-l ↩
↪ist))))))
                                                          ((eql answer 'y)             ;; M ↩
↪OP with a slot
                                                           (format t "~%please input its sl ↩
↪ot ==> ")
                                                           (let ((its-slot (read)))
                                                              (setf slot-list (append slot- ↩
↪list
                                                                                (list
                                                                                (list (car ro ↩
↪le-list)
                                                                                (car filler-l ↩
↪ist)
                                                                               its-slot))))) ↩
↪))))
                                                  (t (setf slot-list (append slot-list
                                                                                (list
                                                                                (list (car ro ↩
↪le-list)
                                                                                (car filler-l ↩
↪ist)))))))
                                                     (setf role-list (cdr role-list))
                                                     (setf filler-list (cdr filler-list)))) ↩
↪))
                   (with-open-file (defmop-stream *cbr-defmop-file*
                                                 :direction :output
                                                 :if-exists :append)
                                   (print  `(defmop ,mop-name ,(list abst-name) ,*mop-or-i ↩
↪nstance* ,@slot-list)
                                           defmop-stream))
                   )))
 ;
 ;     Get date & time
 ;
 (defun get-datetime ()
       (declare (object-variable dtlist))
       (let ((dtlist (cdddr (reverse (multiple-value-list (get-decoded-time))))))
            (format nil "< ~d.~d.~d ~d:~d:~d >"
                    (first  dtlist)
```

```
                        (second dtlist)
                        (third  dtlist)
                        (fourth dtlist)
                        (fifth  dtlist)
                        (sixth  dtlist))))
 ;
 (defun file-alloc ()
       (declare (special *cbr-defmop-file*
                         *CBR-Library*))
       (setf *cbr-defmop-file* (choose-new-file-dialog :directory *CBR-Library*
                                                       :prompt    "As CBR-name.Defmop.Lisp ↩
↪"))
       (with-open-file (defmop-stream *cbr-defmop-file*
                                                   :direction :output
                                                   :if-exists :supersede)
                             (princ ";******************************************************* ↩
↪*" defmop-stream)
                             (princ #\newline defmop-stream)
                             (princ ";*                 CBR MOP Defenition                    ↩
↪*" defmop-stream)
                             (princ #\newline defmop-stream)
                             (princ ";*            Created " defmop-stream)
                             (princ (get-datetime) defmop-stream)
                             (princ "             *" defmop-stream)
                             (princ #\newline defmop-stream)
                             (princ ";******************************************************* ↩
↪*" defmop-stream)))
 ;
 (defun file-update ()
       (declare (special *cbr-defmop-file*
                         *CBR-Library*))
       (setf *cbr-defmop-file* (choose-file-dialog :directory  *CBR-Library*))
       (with-open-file (defmop-stream *cbr-defmop-file*
                                                   :direction :output
                                                   :if-exists :append)
                             (princ #\newline defmop-stream)
                             (princ ";;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; ↩
↪;" defmop-stream)
                             (princ #\newline defmop-stream)
                             (princ ";             Updated " defmop-stream)
                             (princ (get-datetime) defmop-stream)
                             (princ "             ;" defmop-stream)
                             (princ #\newline defmop-stream)
                             (princ ";;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; ↩
↪;" defmop-stream)))
 ;
```

CBRE.Functions2

```
;**************************************************************************
;                          Set Initial Values
;**************************************************************************
;
(defun set-initial ()
       (declare (special *initial-position*
                         *central-position*
                         *l-l-coord*
                         *pre-value*
                         *rect-position*
                         *mop-order*))
       (setf *initial-position* '(10 50))
       (setf *central-position* '(250 200))
       (setf *l-l-coord* nil)                                 ;; list of start and end c ⤸
⤷oordinates of line
       (setf *pre-value* nil)                                 ;; previous value of carri ⤸
⤷ge return variable
       (setf *rect-position* nil)                             ;; list of x-y coordinates ⤸
⤷ of MOP-rectangle's point
       (setf *mop-order* nil))                                ;; list of the order of MO ⤸
⤷Ps shown on the screen
;
;**************************************************************************
;                        define display-function menu items
;**************************************************************************
;* date     : July 7,1990
;* function : defines *display-function* subclass under *menu*
;* argument : none
;* var      : none
;**************************************************************************
;
(defobject *display-function* *menu*)
;
(setq display-fun   (oneof  *display-function*
                           :menu-title     "Display"))
;
;**************************************************************************
;                        display MOP-hierarchy
;**************************************************************************
;* date     : July 7,1990
;* function : shows MOP hierarchy structure below the pointed mop
;* argument : mop name
;* var      : none
;**************************************************************************
;
(setq display-mops  (oneof  *menu-item*
                           :menu-item-title   "MOPs-List"
                           :menu-item-action  '(eval-enqueue
                                                 '(progn
                                                    (format t "enter MOP name ==> ")
                                                    (let ((mop (read))
                                                         (dah mop))))))
;
;**************************************************************************
;                        display slot-hierarchy
;**************************************************************************
;* date     : July 7,1990
;* function : shows slots below the pointed mop
;* argument : mop name
;* var      : none
;**************************************************************************
;
(setq display-slots (oneof  *menu-item*
                              :menu-item-title   "SLOTs-List"
                              :menu-item-action  '(eval-enqueue
                                                    '(progn
                                                       (format t "enter MOP name ==> ")
                                                       (let ((mop (read)))
                                                            (dph mop))))))
;
(setq display-11    (oneof  *menu-item*                 ;; Blank line
                              :menu-item-title   "-"))
;
;**********************************************************************
;                       MOP-hierarchy graph (dah & dph)
;**********************************************************************
;* date     : July 16,1990
;* function : shows hierarchy-graph below the pointed and calls hierarchy
;             functions menu
;* argument : none
;* var      : none
;**********************************************************************
;
;
(setq dah-hierarchy-graph (oneof *menu-item*
                                  :menu-item-title  "MOP-Hierarchy"
                                  :menu-item-action '(eval-enqueue
                                                      '(progn
                                                       (open-hierarchy-window)
                                                       (set-initial)
                                                       (define-redraw-window *hierarchy-gra ↩
↪ph*)
                                                       (p-zoom-out)
                                                       (format t "enter MOP name ==> ")
                                                       (let ((mop (read)))
                                                            (print-grobal-hierarchy *initia ↩
↪l-position* mop))
                                                       (draw-link-line *l-l-coord*)
                                                       (draw-xy-axis)
                                                       (store-hierarchy-graph)
                                                       (ask dah-hierarchy-graph (menu-item- ↩
↪disable))
                                                       (setq *menu-no-hierarchy* (menubar))
                                                       (m-hierarchy-fun)))))
;
(setq dph-hierarchy-graph (oneof *menu-item*
                                  :menu-item-title  "SLOT-Hierarchy"
                                  :menu-item-action '(eval-enqueue
                                                      '(progn
                                                       (dph-set-initial)
                                                       (format t "enter MOP name ==> ")
                                                       (let ((mop (read)))
                                                           (open-dph-window mop)
                                                           (define-redraw-window *dph-graph ↩
↪*)
                                                           (dph-zoom-out)
                                                           (draw-dph mop)
                                                           (setf *shown-dph* mop))
                                                       (store-dph-hierarchy-graph)
                                                       (setq *menu-no-dph* (menubar))
                                                       (m-dph-fun)))))
;
(setq display-12    (oneof  *menu-item*                 ;; Blank line               ↩
↪
                              :menu-item-title   "-"))
;
```

```
;****************************************************************************
;                    Display cases in memory
;****************************************************************************
;* date     : Aug 15,1990
;* function : shows cases in memory
;* argument : none
;* var      : none
;****************************************************************************
(setq cases-in-memory (oneof *menu-item*
                        :menu-item-title  "Cases"
                        :menu-item-action #'(lambda ()
                                              (cond ((eql (show-instances 'm-c ⤸
⤷ase) nil)
                                                     (no-case-message))
                                                    (t
                                                     (show-instances 'm-case) ⤸
⤷
                                                     (setf *cases-dialog-item- ⤸
⤷position* nil)
                                                     (calc-dialog-position)
                                                     (pop-up-cases)
                                                     (make-radio-button)))))))
;
;****************************************************************************
;                    Display new-MOPs in memory
;****************************************************************************
;* date     : Sep 18,1990
;* function : shows new-MOPs in memory
;* argument : none
;* var      : none
;****************************************************************************
;
(setq newmops-in-memory (oneof *menu-item*
                          :menu-item-title  "New MOPs"
                          :menu-item-action #'(lambda ()
                                                (show-new-mop-stack))))
;
(setq display-13    (oneof   *menu-item*                  ;; Blank line          ⤸
⤷
                       :menu-item-title   "-"))
;
;****************************************************************************
;                    quit Display menu
;****************************************************************************
;* date     : July 20,1990
;* function : remove Display menu items and returns previous menu
;* argument : none
;* var      : none
;****************************************************************************
;
(setq display-quit  (oneof  *menu-item*
                       :menu-item-title   "Quit-Display"
                       :menu-item-action  #'(lambda ()
                                              (ask display-fun (remove-menu-items di ⤸
⤷splay-mops))
                                              (ask display-fun (remove-menu-items di ⤸
⤷splay-slots))
                                              (ask display-fun (remove-menu-items di ⤸
⤷splay-11))
                                              (ask display-fun (remove-menu-items da ⤸
⤷h-hierarchy-graph))
                                              (ask display-fun (remove-menu-items dp ⤸
⤷h-hierarchy-graph))
```

```
                                                  (ask display-fun (remove-menu-items di ↩
↪splay-l2))
                                                  (ask display-fun (remove-menu-items ca ↩
↪ses-in-memory))
                                                  (ask display-fun (remove-menu-items ne ↩
↪wmops-in-memory))
                                                  (ask display-fun (remove-menu-items di ↩
↪splay-l3))
                                                  (ask display-fun (remove-menu-items di ↩
↪splay-quit))
                                                  (set-menubar *menu-no-display*))))
;
;*****************************************************************************
;                       set display functions
;*****************************************************************************
;* date     : July 20,1990
;* function : sets Display menu functions
;* argument : none
;* var      : none
;*****************************************************************************
;
(defun m-display-fun ()
      (declare (special *menu-no-display*
                        display-fun
                        display-mops
                        display-slots
                        display-l1
                        dah-hierarchy-graph
                        dph-hierarchy-graph
                        display-l2
                        cases-in-memory
                        newmops-in-memory
                        display-l3
                        display-quit))
      (setq *menu-no-display* (menubar))                                         ↩
↪
      (ask display-fun   (menu-deinstall))
      (ask display-fun   (menu-install))
      (ask display-fun   (add-menu-items display-mops
                                         display-slots
                                         display-l1
                                         dah-hierarchy-graph
                                         dph-hierarchy-graph
                                         display-l2
                                         cases-in-memory
                                         newmops-in-memory
                                         display-l3
                                         display-quit)))
;
;*****************************************************************************
;                       define hierarchy-function menu items
;*****************************************************************************
;* date     : July 20,1990
;* function : defines *hierachy-functions* subclass below *menu*
;* argument : none
;* var      : none
;*****************************************************************************
;
(defobject *hierarchy-function* *menu*)
;
(setq hierarchy-fun   (oneof *hierarchy-function*
                             :menu-title     "MOPs"))
;
```

```
;************************************************************************
;                     scroll hierarchy-grpah
;************************************************************************
;* date     : July 20,1990
;* function : scrolls the screen of hierarchy-graph
;* argument : none
;* var      : none
;************************************************************************
;
(setq scroll-window   (oneof *menu-item*
                             :menu-item-title  "Scroll"
                             :menu-item-action #'(lambda ()
                                                          (scroll-window))
                             :disabled         nil))
;
(setq hierarchy-l1    (oneof *menu-item*                   ;; Blank line
                             :menu-item-title  "-"))
;
;************************************************************************
;                     show slots with its roles and fillers
;************************************************************************
;* date     : July 20,1990
;* function : shows slots below the pointed mop with its roles and fillers
;* argument : none
;* var      : none
;************************************************************************
;
(setq show-slots      (oneof *menu-item*
                             :menu-item-title  "Show-Slots"
                             :menu-item-action #'(lambda ()
                                                          (get-mop-slots))
                             :disabled         nil))
(setq hierarchy-l2    (oneof *menu-item*                   ;; Blank line
                             :menu-item-title  "-"))
;
;************************************************************************
;                     zoom-in
;************************************************************************
;* date     : July 7,1990
;* function : zooms in hierarchy-graph
;* argument : none
;* var      : none
;************************************************************************
;
(setq zoom-in         (oneof *menu-item*
                            :menu-item-title   "Zoom-in"
                            :menu-item-action  #'(lambda ()
                                                          (erase-screen *rect-position* *l ⤸
⤷-l-coord*)
                                                          (p-zoom-in)
                                                          (draw-xy-axis)
                                                          (print-grobal-hierarchy *xy-star ⤸
⤷t* *mop-shown*)
                                                          (draw-link-line *l-l-coord*)
                                                          (store-hierarchy-graph)
                                                          (ask zoom-in  (menu-item-disable ⤸
⤷))
                                                          (ask zoom-out (menu-item-enable) ⤸
⤷))
                            :disabled          nil))
;
;************************************************************************
;                     zoom-out
```

```
;*************************************************************************
;* date     : July 7,1990
;* function : zooms out hierarchy-graph
;* argument : none
;* var      : none
;*************************************************************************
;
(setq zoom-out       (oneof *menu-item*
                        :menu-item-title    "Zoom-out"
                        :menu-item-action   #'(lambda ()
                                                   (erase-screen *rect-position* *l ⤸
⤷-l-coord*)
                                                   (p-zoom-out)
                                                   (draw-xy-axis)
                                                   (print-grobal-hierarchy *xy-star ⤸
⤷t* *mop-shown*)
                                                   (draw-link-line *l-l-coord*)
                                                   (store-hierarchy-graph)
                                                   (ask zoom-in  (menu-item-enable) ⤸
⤷)
                                                   (ask zoom-out (menu-item-disable ⤸
⤷)))
                        :disabled           t))
(setq hierarchy-13   (oneof *menu-item*                  ;; Blank line
                         :menu-item-title   "-"))
;
;*************************************************************************
;                        quit Hierarchy menu
;*************************************************************************
;* date     : July 20,1990
;* function : quits Hierarchy menu and returns the previous menu
;* argument : none
;* var      : none
;*************************************************************************
;
(setq hierarchy-quit  (oneof *menu-item*
                        :menu-item-title    "Quit"
                        :menu-item-action   #'(lambda ()
                                                   (ask hierarchy-fun (remove-menu- ⤸
⤷items scroll-window))
                                                   (ask hierarchy-fun (remove-menu- ⤸
⤷items hierarchy-l1))
                                                   (ask hierarchy-fun (remove-menu- ⤸
⤷items show-slots))
                                                   (ask hierarchy-fun (remove-menu- ⤸
⤷items hierarchy-l2))
                                                   (ask hierarchy-fun (remove-menu- ⤸
⤷items zoom-in))
                                                   (ask hierarchy-fun (remove-menu- ⤸
⤷items zoom-out))
                                                   (ask hierarchy-fun (remove-menu- ⤸
⤷items hierarchy-13))
                                                   (ask hierarchy-fun (remove-menu- ⤸
⤷items hierarchy-quit))
                                                   (ask dah-hierarchy-graph (menu-i ⤸
⤷tem-enable))
                                                   (set-menubar *menu-no-hierarchy* ⤸
⤷))))
;
;*************************************************************************
;                        set hierarchy functions
;*************************************************************************
;* date     : July 20,1990
;* function : sets hierarchy menu functions
;* argument : none
;* var      : none
;*************************************************************************
;
(defun m-hierarchy-fun ()
      (declare (special hierarchy-fun
                        scroll-window
                        hierarchy-l1
                        show-slots
                        hierarchy-l2
                        zoom-in
                        zoom-out
                        hierarchy-l3
                        hierarchy-quit))
      (ask hierarchy-fun (menu-deinstall))
      (ask hierarchy-fun (menu-install))
      (ask hierarchy-fun (add-menu-items scroll-window
                                         hierarchy-l1
                                         show-slots
                                         hierarchy-l2
                                         zoom-in
                                         zoom-out
                                         hierarchy-l3
                                         hierarchy-quit)))
;
;*************************************************************************
;                      define *hierarchy-graph* graphic window
;*************************************************************************
;* date     : July 20,1990
;* function : defines *hierarchy-graph* window below *graphics-window*
;* argument : none
;* var      : none
;*************************************************************************
;
(defobject *graphics-window* *window*)
;
(defun open-hierarchy-window ()
  (declare (special *hierarchy-graph*
                    *graphics-window*))
  (setf *hierarchy-graph* (oneof *graphics-window*
              :window-position #@(100  40)
              :window-size     #@(500 400)
              :window-font     '("courier" 10)
              :window-type     :tool
              :window-title    "Hierarchy-Window")))
;
;*************************************************************************
;                      define-redraw-window
;*************************************************************************
;* date     : July 20,1990
;* function : redraws the stored picture when it exists
;* argument : window-name
;* var      : none
;*************************************************************************
(defun define-redraw-window (window-name)
      (defobfun (view-draw-contents window-name) ()
                (declare (object-variable saved-pict))
                (when (ask (self) (ownp 'saved-pict))
                      (ask (self) (draw-picture (ask (self) saved-pict))))
                (usual-view-draw-contents)))
;
;*************************************************************************
```

```
;                         scroll-window
;*****************************************************************************
;* date     : July 20,1990
;* function : scroll the screen with the distance from the central point of screen
;* argument : none
;* var      : x       x-coordinate of clicked point
;           : y       y-coordinate of clicked point
;           : x-dist  the distance along x axis from the central point of screen
;           : y-dist  the distance along y axis from the central point of screen
;*****************************************************************************
;
(defun scroll-window ()
       (declare (special *hierarchy-graph*))
       (defobfun (window-click-event-handler *hierarchy-graph*) (where)
                 (declare (special        *no-of-cr*
                                          *xy-start*
                                          *mop-shown*
                                          *pre-value*
                                          *x-inc*
                                          *y-inc*
                                          *rect-position*
                                          *l-l-coord*
                                          *central-position*)
                          (object-variable x
                                          y
                                          x-dist
                                          y-dist))
                 (erase-screen *rect-position* *l-l-coord*)
                 (draw-xy-axis)
                 (let ((x (point-h where)) (y (point-v where)))
                      (let ((x-dist (- x (car *central-position*))) (y-dist (- y (cadr *c ↲
↳entral-position*))))
                           (print-grobal-hierarchy (list (- (car *xy-start*) x-dist) (- ( ↲
↳cadr *xy-start*) y-dist))
                                                  *mop-shown*)
                           (draw-link-line *l-l-coord*)))
                 (store-hierarchy-graph)))
;
;*****************************************************************************
;                         store-hierarchy-graph
;*****************************************************************************
;* date     : July 20,1990
;* function : stores the drawing procedure of hierarchy-graph as a picture
;* argument : none
;* var      : none
;*****************************************************************************
;
(defun store-hierarchy-graph ()
       (declare (special *mop-order*
                         *rect-position*
                         *hierarchy-graph*
                         *xy-start*
                         *mop-shown*
                         *l-l-coord*))
       (setf *mop-order* nil)
       (setf *rect-position* nil)
       (ask *hierarchy-graph* (start-picture))
       (print-grobal-hierarchy *xy-start* *mop-shown*)
       (draw-link-line *l-l-coord*)
       (draw-xy-axis)
       (ask *hierarchy-graph* (have 'saved-pict (ask *hierarchy-graph* (get-picture))))) ↲
↳
;
```

```
;*************************************************************************
;                      p-zoom-in
;*************************************************************************
;* date     : July 20,1990
;* function : zooms in hierarchy-graph
;* argument : none
;* var      : none
;*************************************************************************
;
(defun p-zoom-in ()
       (declare (special *hierarchy-graph*
                         *initial-position*
                         *previous-position*
                         *pen-size*
                         *rect-x1*
                         *rect-y1*
                         *rect-x2*
                         *rect-y2*
                         *x-inc*
                         *y-inc*))
       (ask *hierarchy-graph* (window-select))
       (ask *hierarchy-graph* (set-window-font '("helvetica" 12)))
       (setf *previous-position* *initial-position*)
       (setf *pen-size* '#@(2 2))
       (setf *rect-x1* 10)
       (setf *rect-y1* 15)
       (setf *rect-x2* 150)
       (setf *rect-y2* 10)
       (setf *x-inc*   200)
       (setf *y-inc*   40))
;
;*************************************************************************
;                      p-zoom-out
;*************************************************************************
;* date     : July 20,1990
;* function : zooms out hierarchy-graph
;* argument : none
;* var      : none
;*************************************************************************
;
(defun p-zoom-out ()
(declare (special *hierarchy-graph*
                  *initial-position*
                  *previous-position*
                  *pen-size*
                  *rect-x1*
                  *rect-y1*
                  *rect-x2*
                  *rect-y2*
                  *x-inc*
                  *y-inc*))
       (ask *hierarchy-graph* (window-select))
       (ask *hierarchy-graph* (set-window-font '("courier" 9)))
       (setf *previous-position* *initial-position*)
       (setf *pen-size* '#@(1 1))
       (setf *rect-x1* 5)
       (setf *rect-y1* 8)
       (setf *rect-x2* 100)
       (setf *rect-y2* 5)
       (setf *x-inc*   120)
       (setf *y-inc*   20))
;
;*************************************************************************
```

```
;                          draw-xy-axis
;***************************************************************************
;* date     : July 10,1990
;* function : draws x-y axis on the hierachy-graph
;* argument : none
;* var      : none
;***************************************************************************
;
(defun draw-xy-axis ()
       (declare (special *hierarchy-graph*))
       (ask *hierarchy-graph* (window-select))
       (ask *hierarchy-graph* (set-pen-pattern *gray-pattern*))
       (ask *hierarchy-graph* (set-pen-size #@(1 1)))
       (ask *hierarchy-graph* (move-to 250   0))
       (ask *hierarchy-graph* (line-to 250 400))
       (ask *hierarchy-graph* (pen-hide))
       (ask *hierarchy-graph* (move-to   0 200))
       (ask *hierarchy-graph* (pen-show))
       (ask *hierarchy-graph* (line-to 500 200)))
;
;***************************************************************************
;                          draw-mop-unit
;***************************************************************************
;* date     : July 10,1990
;* function : draws the rectangle to represent MOP
;* argument : xy-list    x-y coordinates of the basic point of the rectangle
;*          : mop-name
;* var      : none
;***************************************************************************
;
(defun draw-mop-unit (xy-list mop-name)
       (declare (special        *rect-position*
                                *mop-order*
                                *hierarchy-graph*
                                *pen-size*
                                *rect-x1*
                                *rect-y1*
                                *rect-x2*
                                *rect-y2*)
                (object-variable x
                                y))
       (let ((x (car xy-list)) (y (cadr xy-list)))
            (setf *rect-position* (append (list xy-list) *rect-position*))
            (setf *mop-order* (append (list mop-name) *mop-order*))
            (ask *hierarchy-graph* (window-select))
            (ask *hierarchy-graph* (set-pen-pattern *black-pattern*))
            (ask *hierarchy-graph* (set-pen-size *pen-size*))
            (ask *hierarchy-graph* (move-to x   y))
            (ask *hierarchy-graph* (princ mop-name *hierarchy-graph*))
            (ask *hierarchy-graph* (frame-rect (- x *rect-x1*) (- y *rect-y1*)
                                               (+ x *rect-x2*) (+ y *rect-y2*)))))
;
;***************************************************************************
;                          erase-mop-unit
;***************************************************************************
;* date     : July 20,1990
;* function : erases the rectangle of MOP
;* argument : xy-list    x-y coordinates of the basic point of the rectangle
;* var      : none
;***************************************************************************
;
(defun erase-mop-unit (xy-list)
       (declare (special        *hierarchy-graph*
                                  *pen-size*
                                  *rect-x1*
                                  *rect-y1*
                                  *rect-x2*
                                  *rect-y2*)
                 (object-variable x
                                  y))
       (let ((x (car xy-list)) (y (cadr xy-list)))
           (ask *hierarchy-graph* (window-select))
           (ask *hierarchy-graph* (set-pen-pattern *white-pattern*))
           (ask *hierarchy-graph* (set-pen-size *pen-size*))
           (ask *hierarchy-graph* (move-to x   y))
           (ask *hierarchy-graph* (paint-rect (- x *rect-x1*) (- y *rect-y1*)
                                              (+ x (+ 50 *rect-x2*)) (+ y *rect-y2*)))))
;
;*************************************************************************
;                       erase-screen
;*************************************************************************
;* date     : July 20,1990
;* function : erases MOP restangles and thier liked lines
;* argument : position-list  list of the basic point of MOP rectangles
;*          : l-l-list       list of start and end point of MOP rectangle linked line
;* var      : none
;*************************************************************************
;
(defun erase-screen (position-list l-l-list)
      (declare (special          *rect-position*
                                 *l-l-coord*
                                 *mop-order*)
               (object-variable xy-list
                                se-list))
      (for (xy-list :in position-list)
                    :do (erase-mop-unit xy-list))
      (setf *rect-position* nil)
      (for (se-list :in l-l-list)
                    :do (erase-link-line se-list))
      (setf *l-l-coord*     nil)
      (setf *mop-order*     nil))
;
;*************************************************************************
;                       calc-line-pair.1
;*************************************************************************
;* date     : July 20,1990
;* function : calculates start and end point of MOP linked line
;*          : as a case that MOP has some instances
;* argument : xy-list    x-y coordinates of the basic point of MOP rectangles
;*          : var-list   list of instances below the pointed MOP
;* var      : x-position x coordinate of the basic point of MOP rectangle
;*          : y-position y coordinate of the basic point of MOP rectangle
;*          : x1         x coordinate of upper left corner of MOP rectangle
;*          : y1         y coordinate of upper left corner of MOP rectangle
;*************************************************************************
;
(defun calc-line-pair.1 (xy-list var-list)
      (declare (special          *rect-x2*
                                 *rect-y2*
                                 *rect-x1*
                                 *rect-y1*
                                 *l-l-coord*
                                 *x-inc*
                                 *y-inc*)
               (object-variable x-position
                                y-position
                                n
                                result
                                x1
                                y1))
        (let ((x-position (car xy-list)) (y-position (cadr xy-list))
             (n (length var-list)) (result 0))
            (let ((x1 (+ x-position *rect-x2*))
                  (y1 (- (+ y-position *rect-y2*) (round (* 0.5 (+ *rect-y1* *rect-y2*))) ⤸
⤹)))
                 (dotimes (count n result)
                         (setf *l-l-coord* (append *l-l-coord* (list (list (list x1 y1)
                                                                  (list (- (+  x ⤸
⤹1 (- *x-inc* *rect-x2*)) *rect-x1*)
                                                                       (+  y1 ( ⤸
⤹* *y-inc* result)))))))
                         (setf result (+ 1 result))))))
;
;***************************************************************************
;                       calc-line-pair.2
;***************************************************************************
;* date     : July 20,1990
;* function : calculates start and end point of MOP linked line
;*          : as a case that MOP has vertical hierarchy structure
;* argument : xy-list   x-y coordinates of the basic point of MOP rectangles
;* var      : x-position x coordinate of the basic point of MOP rectangle
;*          : y-position y coordinate of the basic point of MOP rectangle
;*          : x1         x coordinate of upper left corner of MOP rectangle
;*          : y1         y coordinate of upper left corner of MOP rectangle
;*          : x2         x coordinate of lower right corner of MOP rectangle
;*          : y2         y coordinate of lower right corner of MOP rectangle
;***************************************************************************
;
(defun calc-line-pair.2 (xy-list)
      (declare (special        *rect-x2*
                               *rect-y2*
                               *rect-x1*
                               *rect-y1*
                               *l-l-coord*
                               *x-inc*)
               (object-variable x-position
                               y-position
                               x1
                               y1
                               x2
                               y2))
      (let ((x-position (car xy-list)) (y-position (cadr xy-list)))
           (let ((x1 (+ x-position *rect-x2*))
                 (y1 (- (+ y-position *rect-y2*) (round (* 0.5 (+ *rect-y1* *rect-y2*))) ⤸
⤹)))
                (let ((x2 (- (+ x1 (- *x-inc* *rect-x2*)) *rect-x1*)) (y2 y1))
                    (setf *l-l-coord* (append *l-l-coord* (list (list (list x1 y1) (lis ⤸
⤹t x2 y2)))))))))
;
;***************************************************************************
;                       calc-line-pair.3
;***************************************************************************
;* date     : July 20,1990
;* function : calculates start and end point of MOP linked line
;*          : as a case that MOP has paralel hierarchy structure
;* argument : xy-list    x-y coordinates of the basic point of MOP rectangles
;*          : n          the number of paralel MOP hierarchies
;* var      : x-position x coordinate of the basic point of MOP rectangle
;*          : y-position y coordinate of the basic point of MOP rectangle
```

```
 ;*           : x1         x coordinate of upper left corner of MOP rectangle
 ;*           : y1         y coordinate of upper left corner of MOP rectangle
 ;*           : x2         x coordinate of lower right corner of MOP rectangle
 ;*           : y2         y coordinate of lower right corner of MOP rectangle
 ;****************************************************************************
 ;
 (defun calc-line-pair.3 (xy-list n)
       (declare (special *rect-x2*
                         *rect-y2*
                         *rect-x1*
                         *rect-y1*
                         *l-l-coord*
                         *x-inc*
                         *y-inc*
                         *xy-start*)
                (object-variable x-position
                                 y-position
                                 x1
                                 y1
                                 x2
                                 y2))
       (let ((x-position (car xy-list)) (y-position (cadr xy-list)))
            (let ((x1 (+ x-position *rect-x2*))
                  (y1 (- (+ y-position *rect-y2*) (round (* 0.5 (+ *rect-y1* *rect-y2*))) ⤸
⤷)))
                        (let ((x2 (- (+ x1 (- *x-inc* *rect-x2*)) *rect-x1*))
                              (y2 (- (+ (+ (cadr *xy-start*) (* *y-inc* n)) *rect-y2*) ⤸
⤷(round (* 0.5 (+ *rect-y1* *rect-y2*))))))
                             (setf *l-l-coord* (append *l-l-coord* (list (list (list x1 ⤸
⤷ y1) (list x2 y2)))))))))
 ;
 ;****************************************************************************
 ;                       get-mop-slots
 ;****************************************************************************
 ;* date      : July 20,1990
 ;* function  : shows slots below the clicked MOP with its roles and fillers
 ;* argument  : none
 ;* var       : xx        x coordinate of the clicked point
 ;*           : yy        y coordinate of the clicked point
 ;*           : x         x coordinate of the basic point of MOP rectangle
 ;*           : y         y coordinate of the basic point of MOP rectangle
 ;*           : x-min     x coordinate of upper left corner of MOP rectangle
 ;*           : y-min     y coordinate of upper left corner of MOP rectangle
 ;*           : x-max     x coordinate of lower right corner of MOP rectangle
 ;*           : y-max     y coordinate of lower right corner of MOP rectangle
 ;*           : mop-order drawing order of mop
 ;*           : mop-name  mop name
 ;****************************************************************************
 ;
 (defun get-mop-slots ()
       (declare (special *hierarchy-graph*))
       (defobfun (window-click-event-handler *hierarchy-graph*) (where)
                (declare (special         *rect-position*
                                          *beep-sound*
                                          *rect-x1*
                                          *rect-x2*
                                          *rect-y1*
                                          *rect-y2*
                                          *mop-order*
                                          *pop-up-slots*)
                         (object-variable xx
                                          yy
                                          result
```

```
                                            mop-position
                                            x
                                            y
                                            x-min
                                            x-max
                                            y-min
                                            y-max
                                            mop-order
                                            mopname))
                  (let ((xx (point-h where)) (yy (point-v where)))
                       (let ((result 0) (mop-position 0))
                            (setf *beep-sound* nil)
                            (dolist (element *rect-position* result)
                                    (let ((x (car element)) (y (cadr element)))
                                         (let ((x-min (- x *rect-x1*)) (x-max (+ x *rect-x ↩
↪2*))
                                               (y-min (- y *rect-y1*)) (y-max (+ y *rect-y ↩
↪2*)))
                                              (cond ((and (and (> xx x-min) (< xx x-max))
                                                          (and (> yy y-min) (< yy y-max)))
                                                     (setf *beep-sound* (or *beep-sound*   ↩
↪t))
                                                     (let ((result1 0) (mop-order *mop-ord ↩
↪er*))
                                                          (dotimes (count mop-position res ↩
↪ult1)
                                                                   (setf mop-order (cdr mo ↩
↪p-order)))
                                                          (let ((mopname (car mop-order)))
                                                               (pop-up-slots mopname)
                                                               (define-redraw-window *pop- ↩
↪up-slots*)
                                                               (print-pop-up-slots mopname ↩
↪))))
                                                    (t (setf result (+ 1 result))
                                                       (setf *beep-sound* (or *beep-sound* ↩
↪  nil))
                                                       (setf mop-position result)))))))
                  (when (eql *beep-sound* nil) (ed-beep))))
;
;*****************************************************************************
;                      last-out
;*****************************************************************************
;* date     : July 15,1990
;* function : throws out the last atom of the list
;* argument : l          list
;* var      : l-reverse  reversed list
;*****************************************************************************
;
(defun last-out (l)
       (declare (object-variable l-reverse))
       (let ((l-reverse (reverse l)))
            (pop l-reverse)))
;
;*****************************************************************************
;                      last-out-list
;*****************************************************************************
;* date     : July 15,1990
;* function : the rest list after throwing out the last atom of the list
;* argument : l          list
;* var      : l-reverse  reversed list
;*****************************************************************************
;
```

```
(defun last-out-list (l)
       (declare (object-variable l-reverse))
       (let ((l-reverse (reverse l)))
            (pop l-reverse)
            (reverse l-reverse)))
;
;***************************************************************************
;                       draw-link-line
;***************************************************************************
;* date     : July 15,1990
;* function : draws MOP rectangles linked lines
;* argument : l-l-list    the list of start and end points of MOP rectangles liked line
;* var      : x-front     x coordinate of start point of line
;*          : y-front     y coordinate of start point of line
;*          : x-rear      x coordinate of end point of line
;*          : y-rear      y coordinate of end point of line
;***************************************************************************
;
(defun draw-link-line (l-l-list)
       (declare (special         *hierarchy-graph*)
                (object-variable result
                                 x-front
                                 y-front
                                 x-rear
                                 y-rear))
       (let ((result 0))
            (dolist (element l-l-list result)
                    (let ((x-front (car  (car  element)))
                          (y-front (cadr (car  element)))
                          (x-rear  (car  (cadr element)))
                          (y-rear  (cadr (cadr element))))
                          (ask *hierarchy-graph* (window-select))
                          (ask *hierarchy-graph* (set-pen-pattern *black-pattern*))
                          (ask *hierarchy-graph* (move-to x-front y-front))
                          (ask *hierarchy-graph* (line-to x-rear  y-rear))))))
;
;***************************************************************************
;                       erase-link-line
;***************************************************************************
;* date     : July 15,1990
;* function : erases MOP rectangles linked lines
;* argument : l-l-list    the list of start and end points of MOP rectangles liked line
;* var      : x-front     x coordinate of start point of line
;*          : y-front     y coordinate of start point of line
;*          : x-rear      x coordinate of end point of line
;*          : y-rear      y coordinate of end point of line
;***************************************************************************
;
(defun erase-link-line (l-l-list)
       (declare (special *hierarchy-graph*)
                (object-variable x-front
                                 y-front
                                 x-rear
                                 y-rear))
             (let ((x-front (car  (car  l-l-list)))
                   (y-front (cadr (car  l-l-list)))
                   (x-rear  (car  (cadr l-l-list)))
                   (y-rear  (cadr (cadr l-l-list))))
                   (ask *hierarchy-graph* (window-select))
                   (ask *hierarchy-graph* (set-pen-pattern *white-pattern*))
                   (ask *hierarchy-graph* (move-to x-front y-front))
                   (ask *hierarchy-graph* (line-to x-rear  y-rear))))
;
```

```
;**************************************************************************
;                      print-vertical
;**************************************************************************
;* date     : July 10,1990
;* function : draws the vertical hierarchy structure of MOPs from the pointed point
;* argument : xy-list        x-y coordinates of the start point of MOP hierarchy
;*          : list-form      vertical hierarchy structure of MOPs
;* var      : x-position     x coordinate of start point of vertical hierarchy structure
;*          : y-position     y coordinate of start point of vertical hierarchy structure
;*          : inc            increment of u coordinate
;**************************************************************************
;
(defun print-vertical (xy-list list-form)
       (declare (special        *y-inc*)
                (object-variable x-position
                                 y-position
                                 result
                                 inc))
       (let ((x-position (car xy-list)) (y-position (cadr xy-list)) (result 0) (inc *y-in ↩
↪c*))
            (dolist (element list-form result)
                    (draw-mop-unit  (list x-position (+ y-position (* inc result))) elem ↩
↪ent)
                    (setf result (+ result 1)))))
;
;**************************************************************************
;                      print-partical-hierarchy
;**************************************************************************
;* date     : July 15,1990
;* function : draws the hierarchy structure of MOPs from the pointed point
;* argument : xy-list        x-y coordinates of the start point of MOP hierarchy
;*          : list-form      vertical hierarchy structure of MOPs
;* var      : x-position     x coordinate of start point of vertical hierarchy structure
;*          : y-position     y coordinate of start point of vertical hierarchy structure
;*          : var-list       list of MOPs
;*          : item           the first MOP of MOP list
;**************************************************************************
;
(defun print-partial-hierarchy (xy-list list-form)
       (declare (special         *pre-value*
                                 *no-of-cr*
                                 *x-inc*
                                 *y-inc*
                                 *xy-start*)
                (object-variable x-position
                                 y-position))
       (let ((x-position (car xy-list)) (y-position (cadr xy-list))
             (var-list list-form) (item (car list-form)))
            (cond ((eql (cdr list-form) nil)
                   (draw-mop-unit (list x-position y-position) item)
                   (setf *pre-value* (append *pre-value* (list *no-of-cr*)))
                   (setf *no-of-cr* (+ 1 *no-of-cr*)))
                  (t
                   (cond ((eql (%view-fillers item 'instances 'value 'common) nil)
                          (cond ((eql (cddr var-list) nil)
                                 (print-vertical (list x-position y-position) (list item) ↩
↪)
                                 (calc-line-pair.2 (list x-position y-position))
                                 (setf var-list (cadr var-list))
                                 (print-partial-hierarchy (list (+ *x-inc* x-position) y- ↩
↪position) var-list))
                                (t
                                 (print-vertical (list x-position y-position) (list item) ↩
↳)
                                   (setf var-list (cdr var-list))
                                   (setf *pre-value* (append *pre-value* (list *no-of-cr*)) ↵
↳)
                                   (let ((result 0))
                                        (dolist (i var-list result)
                                                (print-partial-hierarchy
                                                (list (+ *x-inc* x-position)
                                                      (+ (cadr *xy-start*) (* *y-inc* *no-o ↵
↳f-cr*))) i)
                                                (calc-line-pair.3 (list x-position y-positi ↵
↳on) (last-out *pre-value*))
                                                (setf *pre-value* (last-out-list *pre-value ↵
↳*)))))))
                             (t
                              (print-vertical (list x-position y-position) (list item))
                              (setf var-list   (for (item :in (cdr var-list))
                                                    :save (car item)))
                              (print-vertical (list (+ *x-inc* x-position) y-position) var-li ↵
↳st)
                              (calc-line-pair.1 (list x-position y-position) var-list)
                              (setf *pre-value* (append *pre-value* (list *no-of-cr*)))
                              (setf *no-of-cr* (+ *no-of-cr* (length var-list)))))))))
;
;*************************************************************************
;                      print-grobal-hierarchy
;*************************************************************************
;* date     : July 15,1990
;* function : draws the whole hierarchy structure of MOPs from the pointed point
;* argument : xy-list        x-y coordinates of the start point of MOP hierarchy
;*          : mop            the parent MOP name of MOP hierarchy structure
;* var      : x-position     x coordinate of start point of vertical hierarchy structure
;*          : y-position     y coordinate of start point of vertical hierarchy structure
;*          : mop-list       list of MOPs below the parent MOP
;*************************************************************************
;
(defun print-grobal-hierarchy (xy-list mop)
       (declare (special        *no-of-cr*
                                *xy-start*
                                *mop-shown*
                                *pre-value*
                                *x-inc*
                                *y-inc*)
                (object-variable x-position
                                 y-position))
       (setf *no-of-cr* 0)
       (setf *xy-start* xy-list)
       (setf *mop-shown* mop)
       (cond ((eql (%view-fillers mop 'instances 'value 'common) nil)
              (print-vertical xy-list (list mop))
              (setf *pre-value* (append *pre-value* (list *no-of-cr*)))
              (let ((x-position (car xy-list)) (y-position (cadr xy-list))
                    (result 0) (mop-list (specs->list mop nil)))
                   (dolist (element mop-list result)
                   (print-partial-hierarchy (list (+ *x-inc* x-position) (+ (* *y-inc* *n ↵
↳o-of-cr*) y-position)) element)
                   (setf result (+ 1 result))
                   (calc-line-pair.3 (list x-position y-position) (last-out *pre-value*))
                   (setf *pre-value* (last-out-list *pre-value*)))))
             (t
              (print-partial-hierarchy xy-list (append (list mop) (specs->list mop nil))) ↵
↳)))
;
```

```
;*************************************************************************
;                         pop-up-slots
;*************************************************************************
;* date     : July 20,1990
;* function : defines *show-slots* window under *window*
;* argument : mopname    MOP name
;* var      : none
;*************************************************************************
;
(defun pop-up-slots (mopname)
       (declare (special *pop-up-slots*
                         *graphics-window*))
       (setf *pop-up-slots* (oneof *graphics-window*
                    :window-type      :tool
                    :window-title     (string mopname)
                    :window-font      '("helvetica" 12)
                    :window-position  #@(10 40)
                    :close-box-p      t)))
;
;*************************************************************************
;                         print-no-slot
;*************************************************************************
;* date     : July 20,1990
;* function : draws no-slot message
;* argument : none
;* var      : none
;*************************************************************************
;
(defun print-no-slot ()
       (declare (special *pop-up-slots*))
       (ask *pop-up-slots* (window-select))
       (ask *pop-up-slots* (set-window-font '("chicago" 12)))
       (ask *pop-up-slots* (set-window-size #@(250 100)))
       (ask *pop-up-slots* (move-to 20 50))
       (ask *pop-up-slots* (princ "There is no slot below this MOP!" *pop-up-slots*))
       (ask *pop-up-slots* (move-to 10 30))
       (ask *pop-up-slots* (frame-rect 10 30 240 60)))
;
;*************************************************************************
;                         print-slots
;*************************************************************************
;* date     : July 20,1990
;* function : draws a table of slots with thier roles and fillers
;* argument : print-buffer    list of slots
;* var      : title           title of slots table
;*          : column          the column of drawing
;*************************************************************************
;
(defun print-slots (print-buffer)
       (declare (special *pop-up-slots*)
                (object-variable title
                                 column
                                 x-start
                                 y-start
                                 x-inc
                                 y-inc
                                 sr
                                 sf))
       (let ((title (list 'slot-role 'slot-filler)) (column (length print-buffer)))
            (let ((window-size (make-point 320 (+ 50 (* column 20)))))
                 (ask *pop-up-slots* (window-select))
                 (ask *pop-up-slots* (set-window-font '("chicago" 12)))
                 (ask *pop-up-slots* (set-window-size window-size)))
            (ask *pop-up-slots* (move-to 20 25))
            (ask *pop-up-slots* (princ (car title) *pop-up-slots*))
            (ask *pop-up-slots* (move-to (+ 10 135) 25))
            (ask *pop-up-slots* (princ (cadr title) *pop-up-slots*))
            (ask *pop-up-slots* (set-window-font '("helvetica" 10)))
            (let ((x-start 20) (y-start 45) (x-inc 125) (y-inc 20) (result 0))
                    (dolist (element print-buffer result)
                       (let ((sr (slot-role element)) (sf (slot-filler element)))
                           (ask *pop-up-slots* (move-to x-start (+ y-start (* y-inc ⤦
⤥ result))))

                           (ask *pop-up-slots* (princ sr *pop-up-slots*))
                           (ask *pop-up-slots* (move-to (+ x-start x-inc) (+ y-star ⤦
⤥t (* y-inc result))))

                           (ask *pop-up-slots* (princ sf *pop-up-slots*))
                           (setf result (+ 1 result)))))
            (ask *pop-up-slots* (move-to 10 10))
            (ask *pop-up-slots* (frame-rect 10 10 302 (round (+ 42 (* column 20)))))
            (ask *pop-up-slots* (move-to 135  10))
            (ask *pop-up-slots* (line-to 135 (round (+ 40 (* column 20)))))
            (ask *pop-up-slots* (move-to  10 30))
            (ask *pop-up-slots* (line-to 300 30))))
;
;**************************************************************************
;                         print-pop-up-slots
;**************************************************************************
;* date     : July 20,1990
;* function : draws slots table
;* argument : mopname       MOP name
;* var      : print-buffer  list of slots below pointed MOP
;**************************************************************************
;
(defun print-pop-up-slots (mopname)
      (declare (special         *pop-up-slots*)
               (object-variable print-buffer))
     (let ((print-buffer (mop-slots mopname)))
         (cond ((eql print-buffer nil)
                (print-no-slot)
                (without-interrupts
                (ask *pop-up-slots* (start-picture))
                (print-no-slot)
                (ask *pop-up-slots* (have 'saved-pict (ask *pop-up-slots* (get-picture) ⤦
⤥)))))
               (t
                (print-slots print-buffer)
                (without-interrupts
                (ask *pop-up-slots* (start-picture))
                (print-slots print-buffer)
                (ask *pop-up-slots* (have 'saved-pict (ask *pop-up-slots* (get-picture) ⤦
⤥))))))))
;
;**************************************************************************
;                        show-instances
;**************************************************************************
;* date     : Aug 13,1990
;* function : returns instances of pointed mop
;* argument : mop          mop name
;* var      : none
;**************************************************************************
(defun show-instances (mop)
      (declare (special *work-instances*))
      (setf *work-instances* nil)
      (get-instances mop))
;
```

```
;***************************************************************************
;                        get-instances
;***************************************************************************
;* date     : Aug 13,1990
;* function : get insancecs of mop
;* argument : mop         mop name
;* var      : speclist    list of mop specs
;***************************************************************************
(defun get-instances (mop)
       (declare (special        *work-instances*)
                (object-variable speclist
                                 element
                                 inst))
       (cond ((eql (get-fillers mop 'instances 'value) nil)
              (let ((speclist (mop-specs mop)))
                   (for (element :in speclist)
                                 :do (let ((inst (get-fillers element 'instances 'value)) ⤸
⤷)
                                          (cond ((eql inst nil)
                                                 (get-instances element))
                                                (t (setf *work-instances* (append *work-i ⤸
⤷nstances* inst))))))))
             (t (setf *work-instances* (append *work-instances* (get-fillers mop 'instan ⤸
⤷ces 'value)))))
       (setf *work-instances* (remove-duplicates *work-instances*)))
;
;***************************************************************************
;                        Define pop-up-cases dialog window
;***************************************************************************
;* date     * Aug 14,1990
;* function * defines *cases-dialog* window under *dialog*
;* argument * none
;* var      * none
;***************************************************************************
;
(defun pop-up-cases ()
       (declare (special *work-instances*
                         *cases-dialog*))
       (setf *cases-dialog* (oneof *dialog*
                    :window-type      :tool
                    :window-title     "Cases in Memory"
                    :window-font      '("helveta" 12)
                    :window-size      (make-point 250 (* 25 (length *work-instances*)))
                    :window-position  #@(380 40)
                    :default-button   t)))
;
;***************************************************************************
;                        calc-dialog-item-position
;***************************************************************************
;* date     * Aug 14,1990
;* function * calculates *radio-button-dialog-item* position from *work-instance*
;* argument * none
;* var      * none
;***************************************************************************
(defobject *cases-radio-button-dialog-item* *radio-button-dialog-item*)
;
(defun calc-dialog-position ()
       (declare (special        *work-instances*
                                *cases-dialog-item-position*)
                (object-variable x-init
                                 y-init
                                 result))
       (let ((x-init 20) (y-init 10) (result 0))
              (dolist (mop *work-instances* result)
                      (setf (get *cases-dialog-item-position* mop) (make-point x-init (+ y- ↲
↳init result)))
                      (setf result (+ result 20)))))
 ;
 ;***************************************************************************
 ;                         make-radio-button
 ;***************************************************************************
 ;* date     * Aug 14,1990
 ;* function * defines dialog-items to *cases-dialog*
 ;* argument * none
 ;* var      * none
 ;***************************************************************************
 ;
 (defun make-radio-button ()
        (declare (special        *work-instances*
                                 *cases-dialog*
                                 *cases-radio-button-dialog-item*
                                 *cases-dialog-item-position*)
                 (object-variable i
                                 mop))
        (let ((i 0))
             (declare (ignore i))
             (for (mop :in *work-instances*)
                      :do
                      (ask *cases-dialog*
                      (add-dialog-items (oneof *cases-radio-button-dialog-item*
                               :dialog-item-position  (get *cases-dialog-item-position* m ↲
↳op)
                               :dialog-item-text      (string mop)
                               :dialog-item-action    '(progn (radio-button-push)
                                                              (princ "Hello"))
                               :radio-button-cluster  0                              ↲
↳
                               :radio-button-pushed-p nil))))))
 ;
 (defun no-case-message ()
        (declare (special *no-case-dialog*))
        (setf *no-case-dialog* (oneof *dialog*
                      :window-type      :tool
                      :window-title     "Cases in Memory"
                      :window-font      '("helveta" 12)
                      :window-size      #@(250 100)
                      :window-position  #@(380  40)
                      :default-button   t))
         (ask *no-case-dialog*
              (add-dialog-items (oneof *static-text-dialog-item*
                               :dialog-item-position #@(30 40)
                               :dialog-item-text     "There is no case in memory!"))))
 ;
 (defobfun (dialog-item-click-event-handler *cases-radio-button-dialog-item*) (where)
           (declare (ignore          where)
                    (special         *work-instances*
                                     *cases-dialog-item-position*
                                     *pop-up-slots*)
                    (object-variable click
                                     result
                                     pos))
           (radio-button-push)
           (let ((click (dialog-item-position)) (result 0))
                (dolist (mop *work-instances* result)
                        (let ((pos (get *cases-dialog-item-position* mop)))
                             (cond ((eql click pos)
                                        (pop-up-slots mop)
                                        (define-redraw-window *pop-up-slots*)
                                        (print-pop-up-slots mop))
                                       (t))))))
;
```

CBRE.Functions3

```
;************************************************************************
;                        Set dph initial values
;************************************************************************
(defun dph-set-initial ()
       (declare (special *dph-central-position*
                         *dph-element-order*
                         *dph-element-position*))
       (setf *dph-central-position* '(150 200))
       (setf *dph-element-order* nil)
       (setf *dph-element-position* nil))
;
;
;************************************************************************
;                        define dph-function menu items
;************************************************************************
;* date     : Aug 20,1990
;* function : defines *dph-hierachy-functions* subclass below *menu*
;* argument : none
;* var      : none
;************************************************************************
;
(defobject *dph-hierarchy-function* *menu*)
;
(setq dph-hierarchy-fun   (oneof *dph-hierarchy-function*
                                 :menu-title     "SLOTs"))
;
;************************************************************************
;                        scroll dph-hierarchy-grpah
;************************************************************************
;* date     : Aug 20,1990
;* function : scrolls the screen of dph-hierarchy-graph
;* argument : none
;* var      : none
;************************************************************************
;
(setq dph-scroll-window   (oneof *menu-item*
                                 :menu-item-title  "Scroll"
                                 :menu-item-action #'(lambda ()
                                                       (dph-scroll-window))
                                 :disabled         nil))
;
(setq dph-hierarchy-l1    (oneof *menu-item*                ;; Blank line
                                 :menu-item-title  "-"))
;
;************************************************************************
;                       dph-zoom-in
;************************************************************************
;* date     : Aug 20,1990
;* function : zooms in dph-hierarchy-graph
;* argument : none
;* var      : none
;************************************************************************
;
(setq dph-zoom-in    (oneof *menu-item*
                            :menu-item-title   "Zoom-in"
                            :menu-item-action  #'(lambda ()
                                                   (dph-erase-screen)
                                                   (dph-zoom-in)
                                                   (draw-dph-xy-axis)
                                                   (draw-dph *shown-dph*)
                                                   (store-dph-hierarchy-graph)
                                                   (ask dph-zoom-in  (menu-item-dis ↩
↪able))
```

```
                                                              (ask dph-zoom-out (menu-item-ena ↘
↳ble)))
                             :disabled          nil))
 ;
 ;*****************************************************************************
 ;                       dph-zoom-out
 ;*****************************************************************************
 ;* date     : Aug 20,1990
 ;* function : zooms out dph-hierarchy-graph
 ;* argument : none
 ;* var      : none
 ;*****************************************************************************
 ;
 (setq dph-zoom-out   (oneof *menu-item*
                             :menu-item-title    "Zoom-out"
                             :menu-item-action   #'(lambda ()
                                                     (dph-erase-screen)
                                                     (dph-zoom-out)
                                                     (draw-dph-xy-axis)
                                                     (draw-dph *shown-dph*)
                                                     (store-dph-hierarchy-graph)
                                                     (ask dph-zoom-in  (menu-item-ena ↘
↳ble))
                                                     (ask dph-zoom-out (menu-item-dis ↘
↳able)))
                             :disabled          t))
 (setq dph-hierarchy-12   (oneof *menu-item*                 ;; Blank line
                                 :menu-item-title  "-"))
 ;
 ;*****************************************************************************
 ;                       quit Hierarchy menu
 ;*****************************************************************************
 ;* date     : Aug 20,1990
 ;* function : quits dph-hierarchy menu and returns the previous menu
 ;* argument : none
 ;* var      : none
 ;*****************************************************************************
 ;
 (setq dph-hierarchy-quit  (oneof *menu-item*
                             :menu-item-title   "Quit"
                             :menu-item-action  #'(lambda ()
                                                (ask dph-hierarchy-fun (remove-menu-it ↘
↳ems dph-scroll-window))
                                                (ask dph-hierarchy-fun (remove-menu-it ↘
↳ems dph-hierarchy-11))
                                                (ask dph-hierarchy-fun (remove-menu-it ↘
↳ems dph-zoom-in))
                                                (ask dph-hierarchy-fun (remove-menu-it ↘
↳ems dph-zoom-out))
                                                (ask dph-hierarchy-fun (remove-menu-it ↘
↳ems dph-hierarchy-12))
                                                (ask dph-hierarchy-fun (remove-menu-it ↘
↳ems dph-hierarchy-quit))
                                                (ask dph-hierarchy-graph (menu-item-en ↘
↳able))
                                                (set-menubar *menu-no-dph*))))
 ;
 ;*****************************************************************************
 ;                       set dph hierarchy functions
 ;*****************************************************************************
 ;* date     : Aug 20,1990
 ;* function : sets dph-hierarchy menu functions
 ;* argument : none
;* var      : none
;*************************************************************************
;
(defun m-dph-fun ()
      (declare (special dph-hierarchy-fun
                        dph-scroll-window
                        dph-hierarchy-l1
                        dph-zoom-in
                        dph-zoom-out
                        dph-hierarchy-l2
                        dph-hierarchy-quit))
      (ask dph-hierarchy-fun (menu-deinstall))
      (ask dph-hierarchy-fun (menu-install))
      (ask dph-hierarchy-fun (add-menu-items dph-scroll-window
                                             dph-hierarchy-l1
                                             dph-zoom-in
                                             dph-zoom-out
                                             dph-hierarchy-l2
                                             dph-hierarchy-quit)))
;
;*************************************************************************
;                         open-dph-graph
;*************************************************************************
;* date     : Aug 19,1990
;* function : defines *dph-graph* window under *graphics-window*
;* argument : mopname    MOP name
;* var      : none
;*************************************************************************
;
(defun open-dph-window (mop)
       (declare (special *dph-graph*
                         *graphics-window*))
       (setf *dph-graph* (oneof *graphics-window*
                     :window-type      :tool
                     :window-title     (string mop)
                     :window-size      #@(300 400)
                     :window-font      '("helvetica" 12)
                     :window-position  #@(10 40)
                     :close-box-p      t)))
;
;*************************************************************************
;                         draw-dph
;*************************************************************************
;* date     : Aug 19,1990
;* function : draws dph hierarchy graph
;* argument : mop      mopname
;* var      : none
;*************************************************************************
(defun draw-dph (mop)
       (declare (special *dph-initial-position*
                         *dph-first-position*))
       (setf *dph-first-position* *dph-initial-position*)
       (calc-element-position (cdr (tree->list mop #'slots->forms nil)))
       (draw-element-procedure))
;
;*************************************************************************
;                         dph-zoom-in
;*************************************************************************
;* date     : Aug 20,1990
;* function : zooms in dph-graph
;* argument : none
;* var      : none
;*************************************************************************
```

```
;
(defun dph-zoom-in ()
       (declare (special *dph-initial-position*
                         *dph-graph*
                         *dph-pen-size*
                         *dph-rect-x1*
                         *dph-rect-y1*
                         *dph-rect-x2*
                         *dph-rect-y2*
                         *dph-x-inc*
                         *dph-y-inc*))
       (setf *dph-initial-position* '(-280 20))
       (ask  *dph-graph* (window-select))
       (ask  *dph-graph* (set-window-font '("helvetica" 12)))
       (setf *dph-pen-size* '#@(2 2))
       (setf *dph-rect-x1*  10)
       (setf *dph-rect-y1*  15)
       (setf *dph-rect-x2* 145)
       (setf *dph-rect-y2*  10)
       (setf *dph-x-inc*   150)
       (setf *dph-y-inc*    30))
;
;*******************************************************************************
;                       dph-zoom-out
;*******************************************************************************
;* date     : Aug 20,1990
;* function : zooms out dph-graph
;* argument : none
;* var      : none
;*******************************************************************************
;
(defun dph-zoom-out ()
       (declare (special *dph-initial-position*
                         *dph-graph*
                         *dph-pen-size*
                         *dph-rect-x1*
                         *dph-rect-y1*
                         *dph-rect-x2*
                         *dph-rect-y2*
                         *dph-x-inc*
                         *dph-y-inc*))
       (setf *dph-initial-position* '(-226 20))
       (ask  *dph-graph* (window-select))
       (ask  *dph-graph* (set-window-font '("courier" 9)))
       (setf *dph-pen-size* '#@(1 1))
       (setf *dph-rect-x1*   5)
       (setf *dph-rect-y1*  10)
       (setf *dph-rect-x2* 118)
       (setf *dph-rect-y2*   5)
       (setf *dph-x-inc*   120)
       (setf *dph-y-inc*    17))
;
;*******************************************************************************
;                       draw-dph-xy-axis
;*******************************************************************************
;* date     : Aug 20,1990
;* function : draws x-y axis on dph-graph
;* argument : none
;* var      : none
;*******************************************************************************
;
(defun draw-dph-xy-axis ()
       (declare (special *dph-graph*))
```

```
        (ask *dph-graph* (window-select))
        (ask *dph-graph* (set-pen-pattern *gray-pattern*))
        (ask *dph-graph* (set-pen-size #@(1 1)))
        (ask *dph-graph* (move-to 150   0))
        (ask *dph-graph* (line-to 150 400))
        (ask *dph-graph* (pen-hide))
        (ask *dph-graph* (move-to   0 200))
        (ask *dph-graph* (pen-show))
        (ask *dph-graph* (line-to 300 200)))
;
;*****************************************************************************
;                        calc-element-position
;*****************************************************************************
;* date     : Aug 20,1990
;* function : calculates every element's start position
;* argument : element-list    slotname lists
;* var      : none
;*****************************************************************************
;
(defun calc-element-position (element-list)
       (declare (special *dph-first-position*
                         *dph-start-position*
                         *dph-x-inc*
                         *dph-y-inc*
                         *dph-element-order*
                         *dph-element-position*)
                (object-variable x-start
                                 x-position
                                 y-position
                                 result
                                 y-newline
                                 x-inc
                                 y-inc))
       (setf *dph-start-position* *dph-first-position*)
       (let ((x-position (car *dph-start-position*)) (y-position (cadr *dph-start-positio ➘
➘n*))
             (result 0) (x-inc *dph-x-inc*) (y-inc *dph-y-inc*))
           (dolist (element element-list result)
                   (let ((x-start (+ x-inc x-position)))
                   (cond ((eql (listp element) nil)
                          (setf *dph-element-order*    (append *dph-element-order* (list ➘
➘element)))
                          (setf *dph-element-position* (append *dph-element-position*
                                (list (list (+ x-start (* x-inc result)) y-position))))
                          (setf *dph-start-position* (list (+ x-start (* result x-inc)) y ➘
➘-position))
                          (setf result (+ 1 result)))
                         (t
                          (let ((y-newline (+ y-inc (cadr *dph-start-position*))))
                               (setf *dph-first-position* (list x-start y-newline)))
                          (calc-element-position element)))))))
;
;*****************************************************************************
;                        draw-element-procedure
;*****************************************************************************
;* date     : Aug 20,1990
;* function : draws slot's rolls and lts roll-fillers
;* argument : none
;* var      : none
;*****************************************************************************
;
(defun draw-element-procedure ()
       (declare (special        *dph-graph*
                                         *dph-pen-size*
                                         *dph-element-order*
                                         *dph-element-position*
                                         *dph-rect-x1*
                                         *dph-rect-y1*
                                         *dph-rect-x2*
                                         *dph-rect-y2*)
                      (object-variable result
                                         n
                                         element-order
                                         element-position
                                         element
                                         x-pos
                                         y-pos))
         (ask *dph-graph* (window-select))
         (ask *dph-graph* (set-pen-pattern *black-pattern*))
         (ask *dph-graph* (set-pen-size *dph-pen-size*))
         (let ((result 0)
               (n (length *dph-element-order*))
               (element-order *dph-element-order*)
               (element-position *dph-element-position*))
              (dotimes (count n result)
                      (let ((element (car element-order))
                            (x-pos (car  (car element-position)))
                            (y-pos (cadr (car element-position))))
                            (ask *dph-graph* (move-to x-pos y-pos))
                            (ask *dph-graph* (princ element *dph-graph*))
                            (ask *dph-graph* (frame-rect (- x-pos *dph-rect-x1*) (-  y-pos ↳
↳*dph-rect-y1*)
                                                              (+ x-pos *dph-rect-x2*) (+  y-pos ↳
↳ *dph-rect-y2*)))
                            (setf element-order (cdr element-order))
                            (setf element-position (cdr element-position))))
              (draw-dph-xy-axis)))
;
;*****************************************************************************
;                       dph-erase-screen
;*****************************************************************************
;* date     : Aug 20,1990
;* function : erases screen
;* argument : none
;* var      : none
;*****************************************************************************
;
(defun dph-erase-screen ()
       (declare (special *dph-graph*
                         *dph-pen-size*
                         *dph-element-order*
                         *dph-element-position*
                         *dph-rect-x1*
                         *dph-rect-y1*
                         *dph-rect-x2*
                         *dph-rect-y2*)
                (object-variable result
                                 n
                                 element-position
                                 x-pos
                                 y-pos))
       (ask *dph-graph* (set-pen-pattern *white-pattern*))
       (ask *dph-graph* (set-pen-size *dph-pen-size*))
       (let ((result 0)
             (n (length *dph-element-order*))
             (element-position *dph-element-position*))
```

```
            (dotimes (count n result)
                  (let ((x-pos (car  (car element-position)))
                        (y-pos (cadr (car element-position))))
                        (ask *dph-graph* (move-to x-pos y-pos))
                        (ask *dph-graph* (paint-rect (- x-pos *dph-rect-x1*) (-  y-pos ↩
↪*dph-rect-y1*)
                                                               (+ x-pos (+ 25 *dph-rect-x2*)) (+ ↩
↪ y-pos *dph-rect-y2*)))
                        (setf element-position (cdr element-position)))))
        (setf *dph-element-order*    nil)
        (setf *dph-element-position* nil))
;
;
;***************************************************************************
;                        dph-scroll-window
;***************************************************************************
;* date     : Aug 20,1990
;* function : scroll the screen with the distance from the central point of screen
;* argument : none
;* var      : x       x-coordinate of clicked point
;           : y       y-coordinate of clicked point
;           : x-dist  the distance along x axis from the central point of screen
;           : y-dist  the distance along y axis from the central point of screen
;***************************************************************************
;
(defun dph-scroll-window ()
       (declare (special *dph-graph*))
       (defobfun (window-click-event-handler *dph-graph*) (where)
                 (declare (special         *dph-initial-position*
                                           *shown-dph*
                                           *dph-central-position*)
                          (object-variable x
                                           y
                                           x-dist
                                           y-dist))
                 (dph-erase-screen)
                 (draw-dph-xy-axis)
                 (let ((x (point-h where)) (y (point-v where)))
                      (let ((x-dist (- x (car *dph-central-position*))) (y-dist (- y (cad ↩
↪r *dph-central-position*))))
                           (setf *dph-initial-position* (list
                                                               (- (car  *dph-initial-position*) ↩
↪ x-dist)
                                                               (- (cadr *dph-initial-position*) ↩
↪ y-dist)))
                           (draw-dph *shown-dph*)
                           (store-dph-hierarchy-graph)))))
;
;***************************************************************************
;                        store-dph-hierarchy-graph
;***************************************************************************
;* date     : Aug 20,1990
;* function : stores the drawing procedure of dph-graph as a picture
;* argument : none
;* var      : none
;***************************************************************************
;
(defun store-dph-hierarchy-graph ()
       (declare (special *dph-graph*))
       (ask *dph-graph* (start-picture))
       (draw-element-procedure)
       (ask *dph-graph* (have 'saved-pict (ask *dph-graph* (get-picture)))))
;
```

```
;(defun draw-dph-hierarchy ()
;       (dph-set-initial)
;       (format t "enter MOP name ==> ")
;       (let ((mop (read)))
;            (open-dph-window mop)
;            (define-redraw-window *dph-graph*)
;            (dph-zoom-out)
;            (draw-dph mop)
;            (setf *shown-dph* mop))
;       (store-dph-hierarchy-graph)
;       (setq *menu-no-dph* (menubar))
;       (m-dph-fun))
```

CBRE.Functions4

```
;*****************************************************************************
;                          pick-up
;*****************************************************************************
;* date     : Aug 17,1990
;* function : picks up pointed order of element in list
;* argument : n          order of number you want to pick up from list
;*            list       list
;* var      : result     counter
;*****************************************************************************
(defun pick-up (n list)
       (declare (object-variable result))
       (let ((result 1))
       (dolist (element list result)
               (if (eql result n) (return element))
               (setf result (+ result 1)))))
;
;*****************************************************************************
;                          Define pop-up-cases dialog window
;*****************************************************************************
;* date     * Aug 14,1990
;* function * defines *cases-dialog* window under *dialog*
;* argument * none
;* var      * none
;*****************************************************************************
;
(defun pop-up-new-mop-stack ()
       (declare (special *new-mop-dialog*))
       (setf *new-mop-dialog* (oneof *dialog*
                    :window-type       :tool
                    :window-title      "New MOPs"
                    :window-font       '("helveta" 12)
                    :window-size       #@(215 350)
                    :window-position   #@(410 40)
                    :default-button    t)))
;
(defobject *new-mop-sequence-dialog-item* *sequence-dialog-item*)
;
;*****************************************************************************
;                          make-sequence-dialog
;*****************************************************************************
;* date     * Aug 14,1990
;* function * defines dialog-items of *sequence-dialog-item*
;* argument * none
;* var      * none
;*****************************************************************************
;
(defun make-sequence-dialog ()
       (declare (special        *new-mop-stack*
                                *new-mop-dialog*)
                (object-variable table-sq))
       (let ((table-sq (mapcar #'string *new-mop-stack*)))
            (ask *new-mop-dialog*
                 (add-dialog-items (oneof *new-mop-sequence-dialog-item*
                              :dialog-item-action
                               #'(lambda ()
                                          (let ((n (+ 1 (point-v (car (selected-cells)))) ↳
↳))
                                                (dph (pick-up n *new-mop-stack*))))
                              :table-vscrollp         t
                              :table-hscrollp         nil
                              :table-dimensions       #@(1 50)
                              :visible-dimensions     #@(1 20)
                              :table-sequence         table-sq)))))
```

```
;
(defun show-new-mop-stack ()
       (pop-up-new-mop-stack)
       (make-sequence-dialog))
```

CBRE.Basic.MOPs

```
;
;*** Clear Memory
;
(clear-memory)
;
;*** Listing 3.21:Basic MOPs
;
(defmop m-event (m-root))
(defmop m-state (m-root))
(defmop m-act (m-root))
(defmop m-actor (m-root))
;
(defmop m-group (m-root))
(defmop m-empty-group (m-group))
(defmop i-m-empty-group (m-empty-group) instance)
;
(defmop m-function (m-root))
(defmop constraint-fn (m-function))
;
(defmop m-pattern (m-root) (abst-fn constraint-fn))
;
(defmop get-sibling (m-function))
;
(defmop m-case (m-root)
    (old m-pattern (calc-fn get-sibling)))
;
(defmop m-role (m-root))
;
(defmop not-constraint (constraint-fn))
(defmop m-not (m-pattern) (abst-fn not-constraint))
;
(defmop m-failed-solution (m-root))
```

CBRE.Floor.Defmop

```
;
;*** Clear Memory
;
(clear-memory)
;
;*** Basic MOPs
;
(defmop m-space-name (m-root) mop)
(defmop i-m-space101 (m-space-name) instance)
(defmop i-m-space102 (m-space-name) instance)
(defmop i-m-space103 (m-space-name) instance)
(defmop i-m-space104 (m-space-name) instance)
(defmop i-m-space105 (m-space-name) instance)
(defmop i-m-space106 (m-space-name) instance)
(defmop i-m-space107 (m-space-name) instance)
(defmop i-m-space108 (m-space-name) instance)
(defmop i-m-space109 (m-space-name) instance)
(defmop i-m-space110 (m-space-name) instance)
(defmop i-m-space111 (m-space-name) instance)
(defmop i-m-space112 (m-space-name) instance)
(defmop i-m-space113 (m-space-name) instance)
;
(defmop m-compass      (m-root))
(defmop north          (m-compass) instance)
(defmop east           (m-compass) instance)
(defmop south          (m-compass) instance)
(defmop west           (m-compass) instance)
;
(defmop m-cooperate    (m-root)
                       (space-name m-space-name)
                       (direction  m-compass))
;
(defmop m-size-minimum (m-root)
                       (value1   nil)
                       (value2   nil))
;
(defmop m-room-minimum (m-root)
                          (minimum  nil))
(defmop i-m-main-bed-room (m-room-minimum)
                          (minimum m-size-minimum
                                   (value1   12.0)
                                   (value2   12.0)))
(defmop i-m-bath-room     (m-room-minimum)
                          (minimum m-size-minimum
                                   (value1    8.0)
                                   (value2    5.0)))

(defmop i-m-family-room   (m-room-minimum)
                          (minimum m-size-minimum
                                   (value1   12.0)
                                   (value2   10.0)))
(defmop i-m-kitchen       (m-room-minimum)
                          (minimum m-size-minimum
                                   (value1   11.0)
                                   (value2    7.0)))
(defmop i-m-laundry       (m-room-minimum)
                          (minimum m-size-minimum
                                   (value1   10.0)
                                   (value2    7.0)))
(defmop i-m-bed-room      (m-room-minimum)
                          (minimum m-size-minimum
                                   (value1   11.0)
                                   (value2    8.7)))
```

```
(defmop i-m-hall            (m-room-minimum)
                            (minimum m-size-minimum
                                     (value1    m-cooperate
                                                (space-name i-m-space109)
                                                (direction  north))
                                     (value2    3.0)))
(defmop i-m-entrance-hall  (m-room-minimum)
                            (minimum m-size-minimum
                                     (value1    m-cooperate
                                                (space-name i-m-space112)
                                                (direction  west))
                                     (value2   3.0)))
(defmop i-m-living-room    (m-room-minimum)
                            (minimum m-size-minimum
                                     (value1   16.0)
                                     (value2   12.0)))
(defmop i-m-dining-room    (m-room-minimum)
                            (minimum m-size-minimum
                                     (value1   10.5)))
(defmop i-m-garage          (m-room-minimum)
                            (single m-size-minimum
                                     (value1   22.0)
                                     (value2   12.0))
                            (double m-size-minimum
                                     (value1   22.0)
                                     (value2   22.0)))
;
(defmop m-space-use (m-root))
(defmop m-space-need (m-root))
(defmop i-m-main-bed-room (m-space-use m-space-need) instance)
(defmop i-m-bath-room      (m-space-use m-space-need) instance)
(defmop i-m-family-room    (m-space-use) instance)
(defmop i-m-kitchen        (m-space-use m-space-need) instance)
(defmop i-m-laundry        (m-space-use) instance)
(defmop i-m-bed-room       (m-space-use m-space-need) instance)
(defmop i-m-hall           (m-space-use) instance)
(defmop i-m-entrance-hall (m-space-use m-space-need) instance)
(defmop i-m-living-room    (m-space-use m-space-need) instance)
(defmop i-m-dining-room    (m-space-use m-space-need) instance)
(defmop i-m-garage         (m-space-use m-space-need) instance)
;
(defmop m-space-minimum (m-root)
                        (space-name m-space-name)
                        (north   nil)
                        (east    nil)
                        (south   nil)
                        (west    nil))
;
(defmop m-space (m-root)
              (space-name m-space-name)
              (width   nil)
              (length  nil)
              (use     nil)
              (requirement nil))
;
(defmop m-sleeping-area (m-space)
              (space-name m-space-name)
              (width   nil)
              (length  nil)
              (no-of-peple nil)
              (use     nil)
              (requirement nil))
;
```

```
(defmop m-living-area (m-space)
               (space-name m-space-name)
               (width   nil)
               (length  nil)
               (no-of-peple nil)
               (use     nil)
               (requirement nil))
;
(defmop m-service-area (m-space)
               (space-name m-space-name)
               (width   nil)
               (length  nil)
               (use     nil)
               (requirement nil))
;
(defmop m-trafic-area (m-space)
               (space-name m-space-name)
               (width   nil)
               (length  nil)
               (use     nil)
               (requirement nil))
;
(defmop m-orientation (m-root)
                      (gl   nil))
;
(defmop m-north-orient (m-orientation)
                      (gl   nil))
(defmop m-east-orient  (m-orientation)
                      (gl   nil))
(defmop m-south-orient (m-orientation)
                      (gl   nil))
(defmop m-west-orient  (m-orientation)
                      (gl   nil))
;
(defmop m-direction   (m-root)
                      (north nil)
                      (east  nil)
                      (south nil)
                      (west  nil))
;
;
(defmop m-land (m-root)
               (width   nil)
               (length  nil)
               (orientation m-orientation))
;
(defmop m-building-area (m-land)
               (width  nil)
               (length nil))
;
;(defmop m-requirement (m-root))
;(defmop m-adjacency-req (m-requirement))
;(defmop m-sunlight-req  (m-requirement))
;(defmop m-noise-req     (m-requirement))
;(defmop m-privacy-req   (m-requirement))
;
(defmop m-function (m-root))
(defmop m-cite/c-fn       (m-function))
(defmop m-design/c-fn     (m-function))
(defmop m-constraint/c-fn (m-function))
;
(defmop select-repair-tech        (m-design/c-fn))
(defmop get-rooms                 (m-design/c-fn))
```

```
(defmop i-m-get-rooms              (get-rooms)           instance)
(defmop get-orientations           (m-design/c-fn))
(defmop i-m-get-orientations       (get-orientations)    instance)
(defmop compare-cite               (m-cite/c-fn))
(defmop i-m-compare-cite           (compare-cite)        instance)
(defmop get-before-land            (m-cite/c-fn))
(defmop i-m-get-before-land        (get-before-land)     instance)
(defmop get-after-land             (m-cite/c-fn))
(defmop i-m-get-after-land         (get-after-land)      instance)
(defmop partial-shrink-rooms       (m-design/c-fn))
(defmop i-m-partial-shrink-rooms   (partial-shrink-rooms)  instance)
(defmop eliminate-room             (m-design/c-fn))
(defmop i-m-eliminate-room         (eliminate-room)      instance)
(defmop eliminate-orientation      (m-design/c-fn))
(defmop i-m-eliminate-orientation  (eliminate-orientation) instance)
(defmop shrink-rooms               (m-design/c-fn))
(defmop i-m-shrink-rooms           (shrink-rooms)        instance)
(defmop extend-rooms               (m-design/c-fn))
(defmop i-m-shrink-rooms           (shrink-rooms)        instance)
;(defmop rotate                     (m-design/c-fn))
;(defmop change-adj                 (m-design/c-fn))
(defmop total-space                (m-constraint/c-fn))
(defmop i-m-total-space            (total-space)         instance)
(defmop news-length                (m-constraint/c-fn))
(defmop i-m-north-length           (news-length)         instance)
(defmop i-m-east-length            (news-length)         instance)
(defmop i-m-south-length           (news-length)         instance)
(defmop i-m-west-length            (news-length)         instance)
(defmop main-bed-room-space        (m-constraint/c-fn))
(defmop i-m-main-bed-room-space    (main-bed-room-space) instance)
(defmop bath-room-space            (m-constraint/c-fn))
(defmop i-m-bath-room-space        (bath-room-space)     instance)
(defmop family-room-space          (m-constraint/c-fn))
(defmop i-m-family-room-space      (family-room-space)   instance)
(defmop kitchen-space              (m-constraint/c-fn))
(defmop i-m-kitchen-space          (kitchen-space)       instance)
(defmop laundry-space              (m-constraint/c-fn))
(defmop i-m-laundry-space          (laundry-space)       instance)
(defmop bed-room-space             (m-constraint/c-fn))
(defmop i-m-bed-room-space         (bed-room-space)      instance)
(defmop hall-space                 (m-constraint/c-fn))
(defmop i-m-hall-space             (hall-space)          instance)
(defmop entrance-hall-space        (m-constraint/c-fn))
(defmop i-m-entrance-hall-space    (entrance-hall-space) instance)
(defmop living-room-space          (m-constraint/c-fn))
(defmop i-m-living-room-space      (living-room-space)   instance)
(defmop dining-room-space          (m-constraint/c-fn))
(defmop i-m-dining-room-space      (dining-room-space)   instance)
(defmop garage-space               (m-constraint/c-fn))
(defmop i-m-garage-space           (garage-space)        instance)
(defmop main-bed-room-space        (m-constraint/c-fn))
;
(defmop m-map    (m-root)
                 (abst m-root)
                 (spec m-root))
;
(defmop m-group (m-root))
;
(defmop m-space-organization (m-group)
                             (gl m-space))
;
(defmop m-cite/c-group       (m-group)
                             (gl m-cite/c-fn))
```

```
 (defmop m-design/c-group     (m-group)
                              (g1 m-design/c-fn))
 (defmop m-constraint/c-group (m-group)
                              (g1 m-constraint/c-fn))
 (defmop i-m-empty-group      (m-cite/c-group m-design/c-group m-constraint/c-group) insta ⤸
⤷nce)
 ;
 (defmop m-map-group          (m-group)
                              (g1 m-map))
 ;
 (defmop m-pattern (m-root)
                  (abst-fn m-function))
 ;
 (defmop m-condition (m-root))
 (defmop i-m-none            (m-condition) instance)
 (defmop i-m-smaller/BA      (m-condition) instance)
 (defmop i-m-same/BA         (m-condition) instance)
 (defmop i-m-bigger/BA       (m-condition) instance)
 (defmop i-m-different/OR    (m-condition) instance)
 (defmop i-m-same/OR         (m-condition) instance)
 (defmop i-m-different/AR    (m-condition) instance)
 (defmop i-m-same/AR         (m-condition) instance)
 ;
 (defmop m-design-knowledge (m-root))
 (defmop m-cite-tech        (m-design-knowledge))
 (defmop m-design-tech      (m-design-knowledge))
 (defmop m-constraint-tech  (m-design-knowledge))
 ;
 (defmop m-cite-check    (m-cite-tech)
                         (difference m-pattern (calc-fn compare-cite))
                         (before     m-pattern (calc-fn get-before-land))
                         (after      m-pattern (calc-fn get-after-land)))
 ;
 (defmop i-m-cite-difference (m-cite-check) instance)
 ;
 (defmop m-just-copy     (m-design-tech)
                         (reason      i-m-same/BA)
                         (room        m-pattern (calc-fn get-rooms))
                         (orientation m-pattern (calc-fn get-orientations)))
 ;
 (defmop i-m-just-copy   (m-just-copy) instance)
 ;
 (defmop m-partial-shrink (m-design-tech)
                          (reason      i-m-smaller/BA)
                          (room        m-pattern (calc-fn partial-shrink-rooms))
                          (orientation m-pattern (calc-fn get-orientations)))
 ;
 (defmop i-m-partial-shrink (m-partial-shrink) instance)
 ;
 (defmop m-eliminate     (m-design-tech)
                         (reason      i-m-smaller/BA)
                         (room        m-pattern (calc-fn eliminate-room))
                         (orientation m-pattern (calc-fn eliminate-orientation)))
 ;
 (defmop i-m-eliminate   (m-eliminate) instance)
 ;
 (defmop m-shrink        (m-design-tech)
                         (reason      i-m-smaller/BA)
                         (room        m-pattern (calc-fn shrink-rooms))
                         (orientation m-pattern (calc-fn get-orientations)))
 ;
 (defmop i-m-shrink      (m-shrink) instance)
 ;
```

```
(defmop m-extend        (m-design-tech)
                        (reason      i-m-bigger/BA)
                        (room        m-space-organization)
                        (orientation m-direction))
;
(defmop i-m-extend      (m-extend) instance)
;
(defmop m-rotate        (m-design-tech)
                        (reason      i-m-different/OR)
                        (room        m-space-organization)
                        (orientation m-direction))
;
(defmop i-m-rotate      (m-rotate) instance)
;
(defmop m-change-adj    (m-design-tech)
                        (reason      i-m-different/AR)
                        (room        m-space-organization)
                        (orientation m-direction))
;
(defmop m-constraint-check        (m-constraint-tech))
;
(defmop m-space-check             (m-constraint-check))
(defmop m-main-bed-room-space     (m-space-check)
                                  (check    m-pattern (calc-fn main-bed-room-space)))
(defmop m-bath-room-space         (m-space-check)
                                  (check    m-pattern (calc-fn bath-room-space)))
(defmop m-family-room-space       (m-space-check)
                                  (check    m-pattern (calc-fn family-room-space)))
(defmop m-kitchen-space           (m-space-check)
                                  (check    m-pattern (calc-fn kitchen-space)))
(defmop m-laundry-space           (m-space-check)
                                  (check    m-pattern (calc-fn laundry-space)))
(defmop m-bed-room-space          (m-space-check)
                                  (check    m-pattern (calc-fn bed-room-space)))
(defmop m-hall-space              (m-space-check)
                                  (check    m-pattern (calc-fn hall-space)))
(defmop m-entrance-hall-space     (m-space-check)
                                  (check    m-pattern (calc-fn entrance-hall-space)))
(defmop m-living-room-space       (m-space-check)
                                  (check    m-pattern (calc-fn living-room-space)))
(defmop m-dining-room-space       (m-space-check)
                                  (check    m-pattern (calc-fn dining-room-space)))
(defmop m-garage-space            (m-space-check)
                                  (check    m-pattern (calc-fn garage-space)))
(defmop m-news-length             (m-space-check)
                                  (check    m-pattern (calc-fn news-length)))
(defmop m-total-space             (m-space-check)
                                  (check    m-pattern (calc-fn total-space)))
;
(defmop m-sunlight-check          (m-constraint-check) mop)
(defmop i-m-sunlight              (m-sunlight-check)   instance)
(defmop m-noise-check             (m-constraint-check) mop)
(defmop i-m-noise                 (m-noise-check)      instance)
(defmop m-privacy-check           (m-constraint-check) mop)
(defmop i-m-privacy               (m-privacy-check)    instance)
;
(defmop m-design-steps  (m-root)
                        (cite-check       m-cite/c-group)
                        (design-change    m-design/c-group)
                        (constraint-check m-constraint/c-group))
;
(defmop m-layout        (m-root)
                        (building-area m-building-area)
```

```
                     (room m-space)
                     (orientation   m-orientation))
;
(defmop m-example    (m-root)
                     (building-area m-building-area)
                     (room m-space)
                     (orientation   m-orientation))
;
(defmop m-failure    (m-root)
                     (constraint    m-constraint/c-fn))
;
(defmop m-explanation (m-root)
                     (failure m-failure))
;
(defmop m-result     (m-root))
(defmop i-m-success  (m-result) instance)
(defmop i-m-failure  (m-result) instance)
;
(defmop m-repair     (m-root)
                     (repaired-solution m-pattern
                                       (calc-fn select-repair-tech)))
;
(defmop m-case       (m-root)
                     (case-name m-design-tech)
                     (result    m-result)
                     (condition m-condition)
                     (before    m-layout)
                     (after     m-example)
                     (step      m-design-steps))
;
(defmop i-m-layout001 (m-layout)
              (building-area m-building-area
                             (width  76.0)
                             (length 25.0))
              (room m-space-organization
                             (g1 m-sleeping-area
                                      (space-name i-m-space101)
                                      (width  15.0)
                                      (length 12.0)
                                      (no-of-people 2)
                                      (use i-m-main-bed-room))
                             (g2 m-sleeping-area
                                      (space-name i-m-space102)
                                      (width  8.0)
                                      (length 6.0)
                                      (no-of-people 1)
                                      (use i-m-bath-room))
                             (g3 m-sleeping-area
                                      (space-name i-m-space103)
                                      (width  8.0)
                                      (length 6.0)
                                      (no-of-people 1)
                                      (use i-m-bath-room))
                             (g4 m-living-area
                                      (space-name i-m-space104)
                                      (width  17.0)
                                      (length 12.0)
                                      (no-of-people 8)
                                      (use i-m-family-room))
                             (g5 m-service-area
                                      (space-name i-m-space105)
                                      (width   9.0)
                                      (length 12.0)
```

```
                              (use i-m-kitchen))
                    (g6 m-service-area
                              (space-name i-m-space106)
                              (width   7.0)
                              (length 12.0)
                              (use i-m-laundry))
                    (g7 m-service-area
                              (space-name i-m-space107)
                              (width  20.0)
                              (length 25.0)
                              (use i-m-garage))
                    (g8 m-sleeping-area
                              (space-name i-m-space108)
                              (width  11.5)
                              (length 13.0)
                              (no-of-people 1)
                              (use i-m-bed-room))
                    (g9 m-sleeping-area
                              (space-name i-m-space109)
                              (width  11.5)
                              (length  9.6)
                              (no-of-people 1)
                              (use i-m-bed-room))
                    (g10 m-trafic-area
                              (space-name i-m-space110)
                              (width  11.5)
                              (length  3.4)
                              (use i-m-hall))
                    (g11 m-trafic-area
                              (space-name i-m-space111)
                              (width   6.5)
                              (length 13.0)
                              (use i-m-entrance-hall))
                    (g12 m-living-area
                              (space-name i-m-space112)
                              (width  16.0)
                              (length 13.0)
                              (no-of-people 6)
                              (use i-m-living-room))
                    (g13 m-living-area
                              (space-name i-m-space113)
                              (width  10.5)
                              (length 13.0)
                              (no-of-people 6)
                              (use i-m-dining-room)))
     (orientation m-direction
                    (north m-north-orient
                              (g1 i-m-space101)
                              (g2 i-m-space102)
                              (g3 i-m-space104)
                              (g4 i-m-space105)
                              (g5 i-m-space106)
                              (g6 i-m-space107))
                    (east  m-east-orient
                              (g1 i-m-space107))
                    (south m-south-orient
                              (g1 i-m-space108)
                              (g2 i-m-space109)
                              (g3 i-m-space111)
                              (g4 i-m-space112)
                              (g5 i-m-space113)
                              (g6 i-m-space107))
                    (west  m-west-orient
                                                   (g1 i-m-space101)
                                                   (g2 i-m-space108))))
;
(defmop m-space-define (m-root))
(defmop width           (m-space-define) instance)
(defmop length          (m-space-define) instance)
;
(defmop m-space-group  (m-group)
                       (g1   m-space-name))
;
(defmop m-equal        (m-root)
                       (equal   nil))
;
(defmop m-equal-width   (m-equal)
                        (equal       width)
                        (argument1  nil)
                        (argument2  nil))
;
(defmop m-equal-width-g1   (m-equal-width)
                           (equal       width)
                           (argument1  nil)
                           (argument2  nil))
;
(defmop i-m-equal-width.1  (m-equal-width-g1)
                           (argument1 m-space-group
                                      (g1   i-m-space101)
                                      (g2   i-m-space102))
                           (argument2 m-space-group
                                      (g1   i-m-space101)
                                      (g2   i-m-space103))
                           (argument3 m-space-group
                                      (g1   i-m-space108)
                                      (g2   i-m-space110))
                           (argument4 m-space-group
                                      (g1   i-m-space108)
                                      (g2   i-m-space109)))
;
(defmop i-m-equal-width.2  (m-equal-width-g1)
                           (argument1 m-space-group
                                      (g1   i-m-space104)
                                      (g2   i-m-space105)
                                      (g3   i-m-space106))
                           (argument2 m-space-group
                                      (g1   i-m-space111)
                                      (g2   i-m-space112)
                                      (g3   i-m-space113)))
;
(defmop i-m-equal-width.3   (m-equal-width-g1)
                            (argument1 m-space-group
                                       (g1   i-m-space107))
                            (argument2 m-space-group
                                       (g1   i-m-space107)))
;
(defmop m-equal-length  (m-equal)
                        (equal       length)
                        (argument1  nil)
                        (argument2  nil))
;
(defmop m-equal-length-g1  (m-equal-length)
                           (equal       length)
                           (argument1  nil)
                           (argument2  nil))
;
```

```
(defmop m-equal-length-g2 (m-equal-length)
                          (equal       length)
                          (argument1  nil)
                          (argument2  nil))
;
(defmop i-m-equal-length.1 (m-equal-length-g1)
                           (argument1 m-space-group
                                    (g1   i-m-space101)
                                    (g2   i-m-space108))
                           (argument2 m-space-group
                                    (g1   i-m-space107)))
;
(defmop i-m-equal-length.2 (m-equal-length-g2)
                           (argument1 m-space-group
                                    (g1   i-m-space101))
                           (argument2 m-space-group
                                    (g1   i-m-space102)
                                    (g2   i-m-space103))
                           (argument3 m-space-group
                                    (g1   i-m-space104))
                           (argument4 m-space-group
                                    (g1   i-m-space105))
                           (argument5 m-space-group
                                    (g1   i-m-space106)))
;
(defmop i-m-equal-length.3 (m-equal-length-g2)
                           (argument1 m-space-group
                                    (g1   i-m-space108))
                           (argument2 m-space-group
                                    (g1   i-m-space110)
                                    (g2   i-m-space109))
                           (argument3 m-space-group
                                    (g1   i-m-space111))
                           (argument4 m-space-group
                                    (g1   i-m-space112))
                           (argument5 m-space-group
                                    (g1   i-m-space113)))
;
```

CBRE.Floor.Defun

```
;
;*** Set Initial Values
;
(defun initial-values ()
       (declare (special *example-mop*
                         *cite-method*
                         *design-tech*
                         *design-method*
                         *constraint-method*
                         *design-steps*
                         *reason*
                         *explanation*
                         *eliminate-room*
                         *measure-width*
                         *measure-length*
                         *distance-queue*
                         *warning-msg*
                         *shrinked-room-list*))
       (setf *example-mop*          nil)                  ;; example mop
       (setf *cite-method*          nil)                  ;; cite-check method
       (setf *design-tech*          nil)                  ;; design-change technique
       (setf *design-method*        nil)                  ;; desing-change method
       (setf *constraint-method*    nil)                  ;; constraint-check method
       (setf *design-steps*         nil)                  ;; design steps
       (setf *reason*               nil)                  ;; the reason chosed
       (setf *explanation*          nil)                  ;; explanation
       (setf *eliminate-room*       nil)                  ;; elminate rooms' group number ↲
↳ list
       (setf *measure-width*        nil)                  ;; measure value of width for l ↲
↳ayout output
       (setf *measure-length*       nil)                  ;; measure value of length for  ↲
↳layout output
       (setf *distance-queue*       nil)                  ;; queue of *distance*
       (setf *warning-msg*          nil)                  ;; queue of warning messages
       (setf *shrinked-room-list*   nil))                 ;; list of shrinked-room
;
(setf *space-check-table* nil)
(setf (get *space-check-table* 'i-m-main-bed-room) 'main-bed-room-space)
(setf (get *space-check-table* 'i-m-bath-room    ) 'bath-room-space)
(setf (get *space-check-table* 'i-m-family-room  ) 'family-room-space)
(setf (get *space-check-table* 'i-m-kitchen      ) 'kitchen-space)
(setf (get *space-check-table* 'i-m-laundry      ) 'laundry-space)
(setf (get *space-check-table* 'i-m-bed-room     ) 'bed-room-space)
(setf (get *space-check-table* 'i-m-hall         ) 'hall-space)
(setf (get *space-check-table* 'i-m-entrance-hall) 'entrance-hall-space)
(setf (get *space-check-table* 'i-m-living-room  ) 'living-room-space)
(setf (get *space-check-table* 'i-m-dining-room  ) 'dining-room-space)
(setf (get *space-check-table* 'i-m-garage       ) 'garage-space)
(setf (get *space-check-table* 'i-m-main-bed-room-space) 'i-m-main-bed-room)
(setf (get *space-check-table* 'i-m-bath-room-space    ) 'i-m-bath-room)
(setf (get *space-check-table* 'i-m-family-room-space  ) 'i-m-family-room)
(setf (get *space-check-table* 'i-m-kitchen-space      ) 'i-m-kitchen)
(setf (get *space-check-table* 'i-m-laundry-space      ) 'i-m-laundry)
(setf (get *space-check-table* 'i-m-bed-room-space     ) 'i-m-bed-room)
(setf (get *space-check-table* 'i-m-hall-space         ) 'i-m-hall)
(setf (get *space-check-table* 'i-m-entrance-hall-space) 'i-m-entrance-hall)
(setf (get *space-check-table* 'i-m-living-room-space  ) 'i-m-living-room)
(setf (get *space-check-table* 'i-m-dining-room-space  ) 'i-m-dining-room)
(setf (get *space-check-table* 'i-m-garage-space       ) 'i-m-garage)
;
(setf *news-direction* nil)
(setf (get *news-direction* 'north) 'm-north-orient)
(setf (get *news-direction* 'east ) 'm-east-orient)
```

```
(setf (get *news-direction* 'west ) 'm-west-orient)
(setf (get *news-direction* 'south) 'm-south-orient)
;
(setf *compass* nil)
(setf (get *compass* 'i-m-north-length) 'north)
(setf (get *compass* 'i-m-east-length ) 'east )
(setf (get *compass* 'i-m-south-length) 'south)
(setf (get *compass* 'i-m-west-length ) 'west )
;
(defun get-rooms (pattern mop)
       (declare (special *design-method*)
                (ignore  pattern
                         mop     ))
       (format t "~%*** Hello there! get-rooms ***")
; #DEBUG# => (print pattern)
; #DEBUG# => (print mop)
       (setq *design-method* (append *design-method* (mop-specs 'get-rooms)))
       (get-filler 'room (car (mop-specs 'm-layout))))
;
(defun get-orientations (pattern mop)
       (declare (special *design-method*)
                (ignore  pattern
                         mop    ))
       (format t "~%*** Hello there! get-orientations ***")
; #DEBUG# => (print pattern)
; #DEBUG# => (print mop)
       (setq *design-method* (append *design-method* (mop-specs 'get-orientations)))
       (get-filler 'orientation (car (mop-specs 'm-layout))))
;
(defun compare-cite (pattern mop)
       (declare (special *cite-method*
                         *reason*)
                (ignore  pattern
                         mop    ))
       (format t "~%*** Hello there! compare-step ***")
; #DEBUG# => (print pattern)
; #DEBUG# => (print mop)
       (setq *cite-method* (append *cite-method* (mop-specs 'compare-cite)))
       (let ((comp1 (role-filler 'building-area (car (mop-specs 'm-layout))))
             (comp2 (role-filler 'building-area (car (mop-specs 'm-example))))
             (ans1 'i-m-same/BA) (ans2 'i-m-different/BA))
             (setf *reason*
                  (cond ((eq comp1 comp2)
                         (format t "~%>>>>> Hey! it's same >>>>>")
                          ans1)
                         (t (format t "~%>>>>> I'm sorry. It's different. >>>>>")
                         (print comp1)
                         (print comp2)
                        ans2)))))
;
(defun get-before-land (pattern mop)
       (declare (special *cite-method*)
                (ignore  pattern
                         mop    ))
       (format t "~%*** Hello there! get-before-land ***")
; #DEBUG# => (print pattern)
; #DEBUG# => (print mop)
       (setq *cite-method* (append *cite-method* (mop-specs 'get-before-land)))
       (role-filler 'building-area (car (mop-specs 'm-layout))))
;
(defun get-after-land (pattern mop)
       (declare (special *cite-method*)
                (ignore  pattern
                       mop))
        (format t "~%*** Hello there! get-after-land ***")
 ; #DEBUG# => (print pattern)
 ; #DEBUG# => (print mop)
        (setq *cite-method* (append *cite-method* (mop-specs 'get-after-land)))
        (role-filler 'building-area (car (mop-specs 'm-example))))
 ;
 ;
 ;*** apply-design-tech
 ;
 (defun apply-design-tech (mop)
                   (add-filler mop  'room        'if-added
                               '(transport-rooms  !filler))
                   (add-filler mop  'orientation 'if-added
                               '(transport-orient !filler))
                   (get-filler 'room        mop)
                   (get-filler 'orientation mop))
 ;
 ;*** get-space-constraint
 ;
 (defun get-space-constraint ()
        (declare (special *example-mop*))
        (format t "~%+++ Hey! get-space-constraint +++")
        (let ((mop-list (mop-specs 'm-space-check)))
             (for (mop :in mop-list)
                      :do (let ((constraint-check (eval `(slots->mop '((example-name ,*ex
ample-mop*))
                                                          '(,mop) nil))))
                          (get-filler 'check constraint-check)))))

 ;
 ;*** select-design-tech
 ;
 (defun select-design-tech (size)
        (declare (special *design-tech*
                          *example-mop*))
        (format t "~%????? select-design-tech ?????")
        (let ((specs (mop-specs 'm-design-tech)))
             (setf *design-tech*
             (for (spec :in specs)
                       :when (eq size (role-filler 'reason spec))
                       :save spec)))
        (cond ((eq 1 (length *design-tech*))
               (let ((design-tech (eval `(slots->mop '((example-name ,*example-mop*))
                                                      ',*design-tech* nil))))
                    (apply-design-tech design-tech)))
              (t (format t "~%*** more design technique ***"))))
 ;
 (defun transport-rooms (rooms)
             (declare (special *example-mop*))
             (add-role-filler 'room        *example-mop* rooms))
 ;
 (defun transport-orient (orient)
             (declare (special *example-mop*))
             (add-role-filler 'orientation *example-mop* orient))
 ;
 ;*** calc-area
 ;
 (defun calc-area (space-mop)
        (* 0.1 (round (* (role-filler 'width  space-mop)
                   (role-filler 'length space-mop)) 0.1)))
 ;
 ;*** list-plus
```

```
;
(defun list-plus (list)
       (let ((result 0))
            (dolist (element list result)
                    (setq result (+ result element)))))
;
;*** calc-total-space
;
(defun calc-total ()
       (declare (special *example-mop*))
       (let ((room-list (mop-slots (role-filler 'room *example-mop*)))
             (result 0))
            (dolist (room room-list result)
                    (let ((width  (role-filler 'width  (cadr room)))
                          (length (role-filler 'length (cadr room))))
                          (setq result (+ result (* width length)))))
            (* 0.1 (round result 0.1))))
;
;*** remove-duplicate-room
;
;(defun remove-duplicate-room (zone)
;       (let ((space-list (mop-specs zone))
;             (result  nil))
;            (let ((identifyer (role-filler 'space-name (car space-list))))
;                 (setq result (append result (list (car space-list))))
;                 (setq space-list (cdr space-list))
;                 (loop
;                     (when (or (endp space-list)
;                               (eq identifyer (role-filler 'space-name (car space-list)) ↲
↳))
;                           (return result))
;                     (setq result (append result (list (car space-list))))
;                     (setq space-list (cdr space-list))))))
;
;*** total-space
;
(defun total-space (pattern mop)
       (declare (special *example-mop*
                         *constraint-method*)
                (ignore  pattern
                         mop))
       (format t "~%+++ calcurate total space +++")
       (let ((build-area (calc-area (role-filler 'building-area *example-mop*)))
             (total (calc-total))
             (passed 'passed) (not-passed 'not-passed))
             (format t "~%build-area  => ~s" build-area)
             (format t "~%total-space => ~s" total)
             (cond ((> total build-area)
                    (setq *constraint-method* (append *constraint-method* (mop-specs 'tot ↲
↳al-space)))
                    not-passed)
                   (t  passed))))
;
;*** shrink-rooms
;
(defun shrink-rooms (pattern mop)
       (declare (ignore  pattern
                         mop)
                (special *example-mop*
                         *design-method*))
       (let ((old-space-org (get-filler  'room                        'i-m-layout001))
             (before-width  (path-filler '(building-area width)  'i-m-layout001))
             (before-length (path-filler '(building-area length) 'i-m-layout001))
```

```
                (after-width   (path-filler '(building-area width)  *example-mop*))
                (after-length  (path-filler '(building-area length) *example-mop*)))
                (let ((rate1 (/ after-width  before-width))
                      (rate2 (/ after-length before-length))
                      (room-list (mapcar #'cadr (mop-slots old-space-org))))
                      (setq *design-method* (append *design-method* (mop-specs 'shrink-rooms ⤸
⤷)))
                      (let ((new-space-org (for (room :in room-list)
                                               :save (shrink-room room rate1 rate2))))
                           (format t "~%new-space-org => ~s" new-space-org)
                           (eval `(list->group ',new-space-org))))))
;
(defun shrink-room (room rate1 rate2)
      (let ((space-name (get-filler 'space-name      room))
            (width      (* rate1 (get-filler 'width  room)))
            (length     (* rate2 (get-filler 'length room)))
            (use        (get-filler 'use             room))
            (mop        (mop-absts room)))
            (eval `(slots->mop  '((space-name  ,space-name)
                                   (width       ,width)
                                   (length      ,length)
                                   (use         ,use)) ',mop t))))
;
;
;****************************************************************************
;                      get-group-number
;****************************************************************************
;* date     * Oct. 21,1990
;* function * returns group number of the room which has the filler of the role
;*          * to get
;* argument * role                : role name
;*          * filler              : the filler
;*          * mop                 : the name of example or layout
;* var      * space-organization
;*          * number-list         : 1 to the length of space-organization list
;*          * result              : space-name list
;****************************************************************************
;
(defun get-group-number (role filler mop)
       (let ((space-organization (role-filler 'room mop)))
            (let ((number-list (make-m-n (length (mop-slots space-organization)) 1))
                  (result nil))
                 (dolist (number number-list result)
                        (cond ((eq (eval `(path-filler '(,number ,role) ',space-organiz ⤸
⤷ation)) filler)
                               (setq result (append result
                                    (list number))))))
            )))
;
(defun garage-space (pattern mop)
      (declare (ignore pattern
                       mop)
               (special *example-mop*
                        *constraint-method*))
      (let ((group-name1 (car (get-group-number 'use 'i-m-garage 'i-m-layout001)))
            (group-name2 (car (get-group-number 'use 'i-m-garage *example-mop*)))
            (space-org1  (role-filler 'room 'i-m-layout001))
            (space-org2  (role-filler 'room *example-mop*))
            (passed     'passed)
            (not-passed 'not-passed))
           (let ((width1  (eval `(path-filler '(,group-name1 width)  ',space-org1)))
                 (length1 (eval `(path-filler '(,group-name1 length) ',space-org1)))
                 (width2  (eval `(path-filler '(,group-name2 width)  ',space-org2)))
```

```
                  (length2 (eval `(path-filler '(,group-name2 length) ',space-org2)))
                  (value1  nil)
                  (value2  nil)
                  (d-value1  (path-filler '(double value1) 'i-m-garage))
                  (d-value2  (path-filler '(double value2) 'i-m-garage))
                  (s-value1  (path-filler '(single value1) 'i-m-garage))
                  (s-value2  (path-filler '(single value2) 'i-m-garage)))
                  (cond ((and (>= width1  d-value1) (>= width1  d-value2)
                              (>= length1 d-value1) (>= length1 d-value2))
                           (setq value1 d-value1) (setq value2 d-value2))
                        (t (setq value1 s-value1) (setq value2 s-value2)))
                  (format t "~%garage minimum 1 => ~s" value1)
                  (format t "~%garage minimum 2 => ~s" value2)
                  (format t "~%garage width     => ~s" width2)
                  (format t "~%garage length    => ~s" length2)
                  (cond ((> width2 length2)
                         (format t "~% width >  length")
                         (cond ((and (>= width2  value1) (>= length2 value2)) passed)
                               (t (setq *constraint-method* (append *constraint-method*
                                                                  (mop-specs 'garage-space)) ↩
↪)
                                  not-passed)))
                        (t
                         (format t "~% width <= length")
                         (cond ((and (>= length2 value1) (>= width2  value2)) passed)
                               (t (setq *constraint-method* (append *constraint-method*
                                                                  (mop-specs 'garage-space)) ↩
↪)
                                  not-passed))))))
 ;
 ;*** dining-room-space
 ;
 (defun dining-room-space (pattern mop)
       (declare (ignore pattern
                        mop)
                (special *example-mop*
                         *constraint-method*))
       (let ((group-name1 (car (get-group-number 'use 'i-m-dining-room 'i-m-layout001)))
             (group-name2 (car (get-group-number 'use 'i-m-dining-room  *example-mop*)))
             (space-org1  (role-filler 'room 'i-m-layout001))
             (space-org2  (role-filler 'room  *example-mop*)))
            (let ((value1 (path-filler '(minimum value1) 'i-m-dining-room))
                  (value2 (+ 4.0 (eval `(path-filler '(,group-name1 no-of-people) ',space ↩
↪-org1))))
                  (work       nil)
                  (passed     'passed)
                  (not-passed 'not-passed))
            (cond ((< value1 value2) (setq work   value1)
                                     (setq value1 value2)
                                     (setq value2 work  )))
            (let ((width   (eval `(path-filler '(,group-name2 width)  ',space-org2)))
                  (length  (eval `(path-filler '(,group-name2 length) ',space-org2))))
                  (format t "~%dining-room minimum 1 => ~s" value1)
                  (format t "~%dining-room minimum 2 => ~s" value2)
                  (format t "~%dining-room width     => ~s" width)
                  (format t "~%dining-room length    => ~s" length)
                  (cond ((> width length)
                         (format t "~%width >  length")
                         (cond ((and (>= width  value1) (>= length value2)) passed)
                               (t (setq *constraint-method* (append *constraint-method*
                                                                 (mop-specs 'dining-room-spa ↩
↪ce)))
                                  not-passed)))
                  (t
                        (format t "~%width <= length")
                        (cond ((and (>= length value1) (>= width  value2)) passed)
                              (t (setq *constraint-method* (append *constraint-method*
                                                                 (mop-specs 'dining-room-spac ⤸
⤷e)))
                                   not-passed)))))))
 ;
 ;*** ordinary-check
 ;
 (defun ordinary-check (space-for-use)
        (declare (special *example-mop*
                          *constraint-method*
                          *space-check-table*))
       (let ((result nil))
          (dolist (group-name (get-group-number 'use space-for-use  *example-mop*) result) ⤸
⤷
                  (let ((space-org  (role-filler 'room  *example-mop*))
                        (value1  (path-filler '(minimum value1) space-for-use))
                        (value2  (path-filler '(minimum value2) space-for-use))
                        (work        nil)
                        (passed     'passed)
                        (not-passed 'not-passed))
                       (cond ((< value1 value2) (setq work   value1)
                                                (setq value1 value2)
                                                (setq value2 work  )))
                       (let  ((width   (eval `(path-filler '(,group-name width)  ',space-o ⤸
⤷rg)))
                              (length  (eval `(path-filler '(,group-name length) ',space-o ⤸
⤷rg))))
                              (format t "~%~s minimum 1 => ~s" space-for-use value1)
                              (format t "~%~s minimum 2 => ~s" space-for-use value2)
                              (format t "~%~s width     => ~s" space-for-use width)
                              (format t "~%~s length    => ~s" space-for-use length)
                              (cond ((> width length)
                                     (format t "~%width >  length")
                                     (cond ((and (>= width  value1) (>= length value2))
                                            (setq result passed))
                                           (t (setq *constraint-method*
                                                    (append *constraint-method*
                                                      (mop-specs (get *space-check-table* ⤸
⤷ space-for-use))))
                                            (setq result not-passed))))
                                    (t
                                    (format t "~%width <= length")
                                    (cond ((and (>= length value1) (>= width  value2))
                                           (setq result passed))
                                          (t (setq *constraint-method*
                                                   (append *constraint-method*
                                                     (mop-specs (get *space-check-table*  ⤸
⤷space-for-use))))
                                             (setq result not-passed)))))
                  )))))
 ;
 ;*** hall-check
 ;
 (defun hall-check (space-for-use)
        (declare (special *example-mop*
                          *constraint-method*))
        (let ((group-name (car (get-group-number 'use space-for-use  *example-mop*)))
              (space-org  (role-filler 'room  *example-mop*)))
             (let ((value2 (path-filler '(minimum value2) space-for-use))
                   (passed     'passed)
```

```
                (not-passed 'not-passed))
          (let  ((width   (eval `(path-filler '(,group-name width)  ',space-org)))
                 (length  (eval `(path-filler '(,group-name length) ',space-org))))
                 (format t "~%~s minimum 1 => ~s" space-for-use value2)
                 (format t "~%~s width     => ~s" space-for-use width)
                 (format t "~%~s length    => ~s" space-for-use length)
                 (cond ((> width length)
                        (format t "~%width >  length")
                        (cond ((>= length  value2) passed)
                              (t not-passed
                                 (setq *constraint-method* (append *constraint-method*
                                                                   (mop-specs 'hall-space)))) ⤶
⤷))
                 (t
                        (format t "~%width <= length")
                        (cond ((>= width   value2) passed)
                              (t not-passed
                                 (setq *constraint-method* (append *constraint-method*
                                                                   (mop-specs 'hall-space)))) ⤶
⤷)))))))
 ;
 ;*** main-bed-room-space
 ;
 (defun main-bed-room-space (pattern mop)
        (declare (ignore pattern
                         mop))
        (ordinary-check 'i-m-main-bed-room))
 ;
 ;*** bath-room-space
 ;
 (defun bath-room-space     (pattern mop)
        (declare (ignore pattern
                         mop))
        (ordinary-check 'i-m-bath-room))
 ;
 ;*** family-room-space
 ;
 (defun family-room-space   (pattern mop)
        (declare (ignore pattern
                         mop))
        (ordinary-check 'i-m-family-room))
 ;
 ;*** kitchen-space
 ;
 (defun kitchen-space       (pattern mop)
        (declare (ignore pattern
                         mop))
        (ordinary-check 'i-m-kitchen))
 ;
 ;*** laundry-space
 ;
 (defun laundry-space       (pattern mop)
        (declare (ignore pattern
                         mop))
        (ordinary-check 'i-m-laundry))
 ;
 ;*** bed-room-space
 ;
 (defun bed-room-space      (pattern mop)
        (declare (ignore pattern
                         mop))
        (ordinary-check 'i-m-bed-room))
 ;
 ;*** living-room-space
 ;
 (defun living-room-space   (pattern mop)
        (declare (ignore pattern
                         mop))
        (ordinary-check 'i-m-living-room))
 ;
 ;*** hall-space
 ;
 (defun hall-space           (pattern mop)
        (declare (ignore pattern
                         mop))
        (hall-check     'i-m-hall))
 ;
 ;*** entrance-hall-check
 ;
 (defun entrance-hall-space (pattern mop)
        (declare (ignore pattern
                         mop))
        (hall-check     'i-m-entrance-hall))
 ;
 ;*** eliminate-room
 ;
 (defun eliminate-room (pattern mop)
        (declare (ignore pattern
                         mop)
                 (special *eliminate-room*
                          *design-method* ))
        (setq *eliminate-room* (get-remove-gnum))
        (let ((room-list (mop-slots (role-filler 'room 'i-m-layout001)))
              (remove-gnum (car *eliminate-room*))
              (result nil))
             (setq *design-method* (append *design-method* (mop-specs 'eliminate-room)))
             (let ((new-room-list (dolist (room room-list result)
                                    (cond ((eq remove-gnum (car room)))
                                          (t (setq result (append result (cdr room)) ↲
↳))))))
                  (list->group new-room-list))))
 ;
 ;*** get-remove-gnum
 ;
 (defun get-remove-gnum ()
        (declare (special *example-mop*))
        (let ((width1  (path-filler '(building-area width)   'i-m-layout001))
              (length1 (path-filler '(building-area length) 'i-m-layout001))
              (width2  (path-filler '(building-area width)   *example-mop*))
              (length2 (path-filler '(building-area length) *example-mop*)))
             (let ((area1 (* width1 length1))
                   (area2 (* width2 length2)))
                  (let ((difference (- area1 area2))
                        (result nil))
                       (format t "~%layout  area => ~s " area1)
                       (format t "~%example area => ~s " area2)
                       (format t "~%difference   => ~s " difference)
                  (dolist (room (get-space-no-need) result)
                          (let ((gnum (car (get-group-number 'use room 'i-m-layout001))))
                               (let ((width3  (eval `(path-filler '(room ,gnum width)  'i ↲
↳-m-layout001)))
                                     (length3 (eval `(path-filler '(room ,gnum length) 'i ↲
↳-m-layout001))))
                                     (format t "~%room area    => ~s" (* width3 length3))
                                     (cond ((> (* width3 length3) difference)
                                            (setq result (append result (list gnum))
                                                (format t "~%Hey! Remove this => ~s" room)))) ↵
↳))
                    ))))
 ;
 ;*** get-space-no-need
 ;
 (defun get-space-no-need ()
        (let ((space-organization (role-filler 'room 'i-m-layout001)))
             (let ((slot-list (mop-slots space-organization)))
                  (let ((room-list (for (slot :in slot-list)
                                   :save (role-filler 'use (cadr slot))))
                       (result nil))
                       (dolist (room room-list result)
                               (cond ((eq (find room (mop-specs 'm-space-need)) nil)
                                      (setq result (append result (list room))))))
                       ))))
 ;
 ;*** eliminate-orientation
 ;
 (defun eliminate-orientation (pattern mop)
        (declare (ignore  pattern
                          mop)
                 (special *eliminate-room*
                          *design-method*))
                 (let ((result nil))
                      (setq *design-method* (append *design-method* (mop-specs 'eliminate- ↵
↳orientation)))
                      (let ((space-name-list (dolist (room *eliminate-room* result)
                                                (let ((space-name (eval `(path-filler '(room ↵
↳,room space-name) 'i-m-layout001))))
                                                     (format t "~%Eliminate : ~s from orient ↵
↳ation" space-name)
                                                     (setq result (append result (list space ↵
↳-name)))))))
                           (let ((north (eliminate-news 'north space-name-list))
                                 (east  (eliminate-news 'east  space-name-list))
                                 (west  (eliminate-news 'west  space-name-list))
                                 (south (eliminate-news 'south space-name-list)))
                                (eval `(slots->mop  '((north ,north)
                                                      (east  ,east)
                                                      (west  ,west)
                                                      (south ,south)) '(m-direction) t)) ↵
↳
                 ))))
 ;
 (defun eliminate-news (direction space-name-list)
        (declare (special *news-direction*))
        (let ((room-list (eval `(mop-slots (path-filler '(orientation ,direction) 'i-m-lay ↵
↳out001))))
              (result1 nil)
              (news (get *news-direction* direction)))
             (dolist (space-name space-name-list result1)
                     (let ((result2 nil))
                          (dolist (room room-list result2)
                                  (cond ((eq space-name (cadr room)))
                                        (t (setq result2 (append result2 (cdr room))))))
                          (setq room-list result2))
                          (setq result1 room-list))
 ; #DEBUG# => (format t "~%~S" room-list)
             (cond ((null room-list) 'i-m-empty-group)
                   (t (eval `(slots->mop
                              ',(for (x :in room-list)
                                     (i :in (make-m-n 1 (length room-list)))
```

```
                                      :save (make-slot i x))
                               '(,news) t))))
              ))
 ;
 ;*** width-length-check
 ;
 (defun width-length-check (direction worl)
     (declare (special *example-mop*
                       *measure-width*
                       *measure-length*))
     (let ((value (eval `(path-filler '(building-area ,worl) *example-mop*)))
          (slots (mop-slots (eval `(path-filler '(orientation ,direction) *example-mop*)) ⤸
⤷))
          (passed     'passed)
          (not-passed 'not-passed)
          (result 0))
          (format t "~%Check => ~s of ~s" direction worl)
          (format t "~%~s" value)
 ; #DEBUG# => (format t "~%~s" slots)
          (dolist (slot slots result)
                 (let ((space-name (cadr slot)))
                      (let ((gnum (car (get-group-number 'space-name space-name *example ⤸
⤷-mop*))))
                           (setq result (+ result (eval `(path-filler '(room ,gnum ,wor ⤸
⤷l) *example-mop*)))))))
          (format t "~%~s" result)
          (cond ((eq worl 'width)
                 (setq *measure-width*  (append *measure-width*  (list result)))
                 (format t "~%*measure-width*  => ~s" *measure-width*))
                (t
                 (setq *measure-length* (append *measure-length* (list result)))
                 (format t "~%*measure-length* => ~s" *measure-length*)))
          (cond ((> result value) not-passed)
                 (t passed))))
 ;
 ;*** news-length
 ;
 (defun news-length (pattern mop)
        (declare (ignore  pattern
                          mop)
                 (special *constraint-method*
                          *news-length*))
        (let ((passed     'passed)
              (not-passed 'not-passed)
              (switch nil))
        (cond ((eq (width-length-check 'north 'width ) not-passed)
               (setq *constraint-method* (append *constraint-method* '(i-m-north-length)))
               (setq switch t)))
        (cond ((eq (width-length-check 'east  'length) not-passed)
               (setq *constraint-method* (append *constraint-method* '(i-m-east-length)))
               (setq switch t)))
        (cond ((eq (width-length-check 'south 'width ) not-passed)
               (setq *constraint-method* (append *constraint-method* '(i-m-south-length)))
               (setq switch t)))
        (cond ((eq (width-length-check 'west  'length) not-passed)
               (setq *constraint-method* (append *constraint-method* '(i-m-west-length)))
               (setq switch t)))
        (cond ((eq switch t) not-passed)
               (t            passed))))
 ;
 ;*** demon : select-design-tech
 ;
 (add-filler 'i-m-cite-difference 'difference  'if-added
            '(select-design-tech !filler))
;
;*** give-land
;
(defun give-land (slot)
       (declare (special *example-mop*))
       (let ((building-area (slots->mop slot '(m-building-area) nil)))
            (let ((width  (role-filler 'width  building-area))
                  (length (role-filler 'length building-area)))
            (format t "~%##############################")
            (format t "~%#     < Building  Area >     #")
            (format t "~%#       width  => ~s       #" width)
            (format t "~%#       length => ~s       #" length)
            (format t "~%##############################")
            (setf *example-mop* (eval `(slots->mop '((building-area ,building-area))
                        '(m-example)
                        nil))))))
;
;***************************************************************************
;                     start-design
;***************************************************************************
;* date     * Oct. 25,1990
;* function * starts from cite check step or gets previous design technology from case me
mory
;* argument * none
;*          * design-tech  :  instance of design technology
;***************************************************************************
;
(defun start-design ()
       (declare (special  *example-mop*
                          *design-tech*))
       (setq *design-tech* (mop-absts (role-filler 'case-name (car (mop-specs 'm-case))))
)
       (cond ((eq *design-tech* nil)
              (format t "~%+++ Now we gonna start designing from cite check step. +++")
              (get-filler 'before     (car (mop-specs 'm-cite-check)))
              (get-filler 'after      (car (mop-specs 'm-cite-check)))
              (get-filler 'difference (car (mop-specs 'm-cite-check))))
             (t
              (format t "~%+++ Now we gonna get previous design technique from case-memor
y. +++")
              (let ((design-tech (eval `(slots->mop '((example-name ,*example-mop*)) ',*d
esign-tech* nil))))
              (apply-design-tech design-tech)))))
;
;*** store-case
;
(defun store-case ()
      (declare (special *design-tech*
                        *cite-method*
                        *reason*
                        *example-mop*
                        *design-method*
                        *constraint-method*
                        *design-steps*))
      (format t "~%+++ Now we gonna store this case. +++")
      (let ((before (car (mop-specs 'm-layout)))
            (after  (car (mop-specs 'm-example)))
            (none   'i-m-none)
            (case-name        (car (mop-specs (car *design-tech*))))
            (result           (which-case  *example-mop*))
            (cite-method      (list->group *cite-method*))
            (design-method    (list->group *design-method*))
```

```
              (constraint-check (list->group *constraint-method*)))
              (print cite-method)
              (print design-method)
              (print constraint-check)
              (setq *design-steps* (eval `(slots->mop '((cite-check       ,cite-method)
                                                        (design-change    ,design-method)
                                                        (constraint-check ,constraint-check ⤸
⤷))
                                                       '(m-design-steps) nil)))
              (let ((reason (cond ((eq *reason* nil) none)
                                  (t  *reason*))))
              (print *design-steps*)
              (print result)
              (print reason)
              (eval `(slots->mop '((case-name ,case-name)
                                   (result    ,result)
                                   (reason    ,reason)
                                   (before    ,before)
                                   (after     ,after)
                                   (steps     ,*design-steps*)) '(m-case) nil))))
         (draw-layout)                                          ;; draw layout output
         (break "+++ after store case to memory +++"))
 ;
 ;*** which-case
 ;
 (defun which-case (example)
        (let ((space-mop-list (mop-specs 'm-space-check)))
             (let ((space-list   (for (mop :in space-mop-list)
                                          :save (mop-specs mop)))
                   (result nil)
                   (not-passed nil)
                   (success 'i-m-success)
                   (failure 'i-m-failure))
                   (let ((space-check-result (for (instance-list :in space-list)
                                                :save (dolist (instance instance-list result)
                                                              (cond ((eq  example (role-filler ⤸
⤷ 'example-name instance))
                                                                         (setq result (role-filler ⤸
⤷ 'check instance)))
                                                                        (t)))))
                   (print space-check-result)
                   (let ((result1 nil))
                        (dolist (element space-check-result result1)
                                (cond ((eq element 'not-passed)
                                       (setq not-passed 't))
                                      (t))))
                   (cond ((eq not-passed nil)
                          (format t "~%+++ This case stored as a good-case. +++")
                           success)
                          (t
                          (format t "~%+++ This case stored as a bad-case.  +++")
                           failure))))))
 ;
 ;**************************************************************************
 ;                      list-compress
 ;**************************************************************************
 ;* date     * Oct. 20,1990
 ;* function * returns the list without nil
 ;* argument * list
 ;* var      * reult
 ;*             input  ==> (nil nil a nil b nil c nil nil)
 ;*             return ==> (a b c)
 ;**************************************************************************
```

```
;
(defun list-compress (list)
       (let ((result nil))
            (dolist (element list result)
                    (cond ((eq element nil))
                          (t (setq result (append result (list element)))))))))
;
(defun remove-slot (slot mop)
      (erase-slot mop slot))
;
(defun select-repair-tech (pattern mop)
       (declare (ignore  pattern
                         mop)
                (special *explanation*))
       (format t "~%+++ Let's get another technique. +++")
       (let ((condition (role-filler 'condition *explanation*))
             (design-tech-list (mop-specs 'm-design-tech))
             (result1 nil))
            (dolist (design-tech design-tech-list result1)
                    (let ((instance-list (mop-specs design-tech))
                         (result2 nil))
                         (dolist (instance instance-list result2)
                                 (cond ((eq condition (get-filler 'reason instance))
                                       (setq result2 (append result2 (list instance))))
                                       (t)))
                    (setq result1 (append result1 result2))))
                    result1))
;
;***************************************************************************
;                        set-explanation
;***************************************************************************
;* date     * Oct. 24,1990
;* function * sets m-explanation instance
;* argument * none
;*          * mop       :   mop  name
;* var      * constraint   : not-passed constraint check list
;*          * failure      : m-failure instnace
;*          * condition    : m-condition instance
;*          * design-tech  : design technology which was used in this step
;*          * abst1        : abstruct mop name
;*          * spec1        : instance of above abstruct mop
;*          * abst2        : abstruct mop name
;*          * spec2        : instance of above abstruct mop
;***************************************************************************
;
(defun set-explanation ()
       (declare (special *constraint-method*
                         *design-tech*
                         *explanation*))
       (cond ((eq *constraint-method* nil)
              (format t "~%+++ no need to set explanation mop +++"))
             (t
              (format t "~%+++ explanation mop will be set +++")
              (let ((constraint (list->group *constraint-method*)))
                   (let ((failure (slots->mop `((constraint ,constraint))
                                               '(m-failure) t))
                         (condition 'i-m-smaller/BA)
                         (design-tech (car (mop-specs (car *design-tech*)))))
                         (let ((abst1 (car (mop-absts condition)))
                               (spec1 condition)
                               (abst2 (car (mop-absts design-tech)))
                               (spec2 (car (reverse (mop-specs (car (mop-absts design-t
ech)))))))
```

```
                                   (let ((mapping (slots->mop (eval `(forms->slots '((g1 m- ⤸
⤷map instance
                                                                               (abst ⤸
⤷,abst1) (spec ,spec1))
                                                                               (g2 m- ⤸
⤷map instance
                                                                               (abst ⤸
⤷,abst2) (spec ,spec2)))))
                                                                             '(m-map- ⤸
⤷group) t)))
                                       (print mapping)
                                       (setq *explanation*
                                               (slots->mop `((failure     ,failure)
                                                             (condition   ,condition)
                                                             (design-tech ,design-tech)
                                                             (mapping     ,mapping))
                                                            '(m-explanation) t)))))))))
 ;
 ;***********************************************************************
 ;                    apply-reapir
 ;***********************************************************************
 ;* date     * Oct. 25,1990
 ;* function * apply repair technology
 ;* argument * tech          :  repair technology name
 ;*          * building-area :  example's building area value
 ;*          * mop-tech      :  abstruct mop name of repaire technology
 ;***********************************************************************
 ;
 (defun apply-repair (tech)
       (declare (special *example-mop*
                         *design-tech*
                         *design-method*))
       (let ((building-area (get-filler 'building-area *example-mop*)))
            (initial-values)
            (setq *design-tech* (mop-absts tech))
            (setq *example-mop* (eval `(slots->mop '((building-area ,building-area)
                                                     (repair-tech   ,tech))
                                                   '(m-example) nil))))
            (let ((mop-tech (mop-absts tech)))
                 (let ((design-tech (eval `(slots->mop '((example-name ,*example-mop*))
                                                       ',mop-tech t))))
                      (print design-tech)
                      (apply-design-tech design-tech))))
 ;
 ;***********************************************************************
 ;                    get-repaired-solution
 ;***********************************************************************
 ;* date     * Oct. 25,1990
 ;* function * searches repaired solution
 ;* argument * none
 ;*          * instance      :  instance of repaire mop
 ;***********************************************************************
 ;
 (defun get-repaired-solution ()
       (declare (special *explanation*
                         *repaired-instance*))
       (let ((instance (eval `(slots->mop '((explanation ,*explanation*)) '(m-repair) t)) ⤸
⤷))
            (setq *repaired-instance*     nil)
            (setq *repaired-instance* instance)
            (format t "~%*repaied-instance* => ~s" *repaired-instance*)
            (and (get-filler 'repaired-solution instance) instance)))
 ;
```

```
;*****************************************************************************
;                       repaired-steps
;*****************************************************************************
;* date     * Oct. 25,1990
;* function * go on repair procedures
;* argument * none
;*          * (1) apply repaired technique
;*          * (2) check space constraints
;*          * (3) store to case memory
;*****************************************************************************
;
(defun repaired-steps (repaired-tech)
       (apply-repair repaired-tech)
       (get-space-constraint)
       (store-case)
       (set-explanation))
```

CBRE.Floor.Defun1

```
 (defun get-space-minimum (space-for-use)
        (let ((result nil))
             (dolist (gnum (get-group-number 'use space-for-use 'i-m-layout001) result)
 ; #DEBUG# => (format t "~%~s" gnum)
                     (let ((value1 (path-filler '(minimum value1) space-for-use))
                           (value2 (path-filler '(minimum value2) space-for-use))
                           (space-name (eval `(path-filler '(room ,gnum space-name) 'i-m-l ↩
↪ayout001)))
                           (work 0))
                     (cond ((< value1 value2) (setq work   value1)
                                              (setq value1 value2)
                                              (setq value2 work  )))
                     (format t "~%~s minimum 1 => ~s" space-for-use value1)
                     (format t "~%~s minimum 2 => ~s" space-for-use value2)
                     (let ((width  (eval `(path-filler '(room ,gnum width)  'i-m-layout001 ↩
↪)))
                           (length (eval `(path-filler '(room ,gnum length) 'i-m-layout001 ↩
↪))))
                           (cond ((> width length)
                                  (format t "~%width >  length")
                                  (eval `(slots->mop  '((space-name ,space-name)
                                                        (north ,value1)
                                                        (east  ,value2)
                                                        (south ,value1)
                                                        (west  ,value2)) '(m-space-minimum) ↩
↪ t)))
                                 (t
                                  (format t "~%width <= length")
                                  (eval `(slots->mop  '((space-name ,space-name)
                                                        (north ,value2)
                                                        (east  ,value1)
                                                        (south ,value2)
                                                        (west  ,value1)) '(m-space-minimum) ↩
↪ t)))))))))
 ;
 (defun get-garage-minimum ()
        (let ((gnum (car (get-group-number 'use 'i-m-garage 'i-m-layout001))))
             (let ((width  (eval `(path-filler '(room ,gnum width)  'i-m-layout001)))
                   (length (eval `(path-filler '(room ,gnum length) 'i-m-layout001)))
                   (space-name (eval `(path-filler '(room ,gnum space-name) 'i-m-layout001 ↩
↪)))
                   (value1  nil)
                   (value2  nil)
                   (d-value1  (path-filler '(double value1) 'i-m-garage))
                   (d-value2  (path-filler '(double value2) 'i-m-garage))
                   (s-value1  (path-filler '(single value1) 'i-m-garage))
                   (s-value2  (path-filler '(single value2) 'i-m-garage)))
                   (cond ((and (>= width  d-value1) (>= width  d-value2)
                               (>= length d-value1) (>= length d-value2))
                            (setq value1 d-value1) (setq value2 d-value2))
                         (t (setq value1 s-value1) (setq value2 s-value2)))
                   (format t "~%garage minimum 1 => ~s" value1)
                   (format t "~%garage minimum 2 => ~s" value2)
                   (cond ((> width length)
                          (format t "~%width >  length")
                          (eval `(slots->mop  '((space-name ,space-name)
                                                (north ,value1)
                                                (east  ,value2)
                                                (south ,value1)
                                                (west  ,value2)) '(m-space-minimum) t)))
                         (t
                          (format t "~%width <= length")
                          (eval `(slots->mop  '((space-name ,space-name)
                                              (north ,value2)
                                              (east  ,value1)
                                              (south ,value2)
                                              (west  ,value1)) '(m-space-minimum) t))))) ⤸
⤷))
 ;
 (defun get-dining-minimum ()
       (let ((gnum (car (get-group-number 'use 'i-m-dining-room 'i-m-layout001))))
            (let ((value1 (path-filler '(minimum value1) 'i-m-dining-room))
                  (value2 (+ 4.0 (eval `(path-filler '(room ,gnum no-of-people) 'i-m-layo ⤸
⤷ut001))))
                  (space-name (eval `(path-filler '(room ,gnum space-name) 'i-m-layout001 ⤸
⤷)))
                  (work        nil))
            (cond ((< value1 value2) (setq work   value1)
                                     (setq value1 value2)
                                     (setq value2 work  )))
            (let ((width   (eval `(path-filler '(room ,gnum width)  'i-m-layout001)))
                  (length  (eval `(path-filler '(room ,gnum length) 'i-m-layout001))))
                  (format t "~%dining-room minimum 1 => ~s" value1)
                  (format t "~%dining-room minimum 2 => ~s" value2)
                  (cond ((> width length)
                         (format t "~%width >  length")
                         (eval `(slots->mop  '((space-name ,space-name)
                                              (north ,value1)
                                              (east  ,value2)
                                              (south ,value1)
                                              (west  ,value2)) '(m-space-minimum) t)))
                        (t
                         (format t "~%width <= length")
                         (eval `(slots->mop  '((space-name ,space-name)
                                              (north ,value2)
                                              (east  ,value1)
                                              (south ,value2)
                                              (west  ,value1)) '(m-space-minimum) t))))) ⤸
⤷)))
 ;
 (defun hall-minimum (space-for-use)
       (let ((gnum (car (get-group-number 'use space-for-use 'i-m-layout001))))
            (let ((space-name1 (eval `(path-filler '(room ,gnum space-name) 'i-m-layout00 ⤸
⤷1)))
                  (space-name2 (path-filler '(minimum value1 space-name) space-for-use))
                  (direction   (path-filler '(minimum value1 direction)  space-for-use)))
                  (let ((value1 (car (for (instance :in (mop-specs 'm-space-minimum))
                                              :when (eq (role-filler 'space-name instanc ⤸
⤷e) space-name2)
                                              :save (role-filler direction instance))))
                        (value2 (path-filler '(minimum value2) space-for-use)))
                        (let  ((width   (eval `(path-filler '(room ,gnum width)  'i-m-lay ⤸
⤷out001)))
                               (length  (eval `(path-filler '(room ,gnum length) 'i-m-lay ⤸
⤷out001))))
                               (format t "~%~s minimum 1 => ~s" space-for-use value1)
                               (format t "~%~s minimum 2 => ~s" space-for-use value2)
                               (cond ((> width length)
                                      (format t "~%width >  length")
                                      (eval `(slots->mop  '((space-name ,space-name1)
                                                           (north ,value1)
                                                           (east  ,value2)
                                                           (south ,value1)
                                                           (west  ,value2)) '(m-space-mi ⤸
⤷nimum) t)))
                                     (t
                                            (format t "~%width <= length")
                                            (eval `(slots->mop  '((space-name ,space-name1)
                                                                  (north ,value2)
                                                                  (east  ,value1)
                                                                  (south ,value2)
                                                                  (west  ,value1)) '(m-space-min ↩
↪imum) t)))))))))
 ;
 (defun check-room-minimum ()
        (get-space-minimum 'i-m-main-bed-room)
        (get-space-minimum 'i-m-bath-room)
        (get-space-minimum 'i-m-family-room)
        (get-space-minimum 'i-m-kitchen)
        (get-space-minimum 'i-m-laundry)
        (get-space-minimum 'i-m-bed-room)
        (get-space-minimum 'i-m-living-room)
        (get-dining-minimum)
        (get-garage-minimum)
        (hall-minimum 'i-m-entrance-hall)
        (hall-minimum 'i-m-hall))
 ;
 (defun calc-minimum-length (direction)
        (let ((room-list (eval `(mop-slots (path-filler '(orientation ,direction) 'i-m-lay ↩
↪out001))))
              (result nil))
              (dolist (room room-list result)
                      (let ((space-name (cadr room)))
                           (setq result
                             (append result (for (instance :in (mop-specs 'm-space-minimu ↩
↪m))
                                                             :when (eq (role-filler 'space-name ↩
↪instance) space-name)
                                                             :save (role-filler direction instan ↩
↪ce)))))
               result)))
 ;
 (defun length-limit ()
        (declare (special *example-mop*
                          *width-limit*
                          *length-limit*))
        (let ((north-list (calc-minimum-length 'north))
              (east-list  (calc-minimum-length 'east ))
              (south-list (calc-minimum-length 'south))
              (west-list  (calc-minimum-length 'west )))
              (let ((north-length (list-plus north-list))
                    (east-length  (list-plus east-list))
                    (south-length (list-plus south-list))
                    (west-length  (list-plus west-list)))
                    (format t "~%north-length => ~s" north-length)
                    (format t "~%east-length  => ~s" east-length)
                    (format t "~%south-length => ~s" south-length)
                    (format t "~%west-length  => ~s" west-length)
                   (let ((width1  (max north-length south-length))
                         (length1 (max east-length  west-length ))
                         (width2  (path-filler '(building-area width)  *example-mop*))
                         (length2 (path-filler '(building-area length) *example-mop*)))
                         (format t "~%width -limit => ~s" width1)
                         (format t "~%length-limit => ~s" length1)
                         (cond ((< width2 width1)
                                (format t "~%@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@")
                                (format t "~%@ Stop! This Exceeds width minimum  @")
                                (format t "~%@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@")))
                         (cond ((< length2 length1)
                                (format t "~%@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@")
                                (format t "~%@ Stop! This Exceeds length minimum @")
                                (format t "~%@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@")))
                          (cond ((or (< width2 width1) (< length2 length2)))
                                (t
                                 (setq *width-limit*   (- width2  width1 ))
                                 (setq *length-limit*  (- length2 length1))))
               ))))
;
;(defun room-2ext-walls (direct1 direct2)
;      (let ((direct1-rooms (mapcar #'cadr (mop-slots
;                                          (eval `(path-filler '(orientation ,direct1) 'i ↵
↳-m-layout001)))))
;           (direct2-rooms (mapcar #'cadr (mop-slots
;                                          (eval `(path-filler '(orientation ,direct2) 'i ↵
↳-m-layout001)))))
;           (result nil))
;           (dolist (room direct1-rooms result)
;                  (cond ((eq (member room direct2-rooms) nil) nil)
;                        (t (setq result (append result (list room)))))))
;
;
;**********************************************************************
;                      get-minimum-length
;**********************************************************************
;* date     * Oct 30,1990
;* function * returns minimum length of room which has space-name
;* argument * none
;*          * space-name :  name of space
;*          * direction  :  one of (north east south west)
;**********************************************************************
;
(defun get-minimum-length (space-name direction)
      (for (instance :in (mop-specs 'm-space-minimum))
                     :when (eq (role-filler 'space-name instance) space-name)
                     :save (role-filler direction instance)))
;
;**********************************************************************
;                      get-space-length
;**********************************************************************
;* date     * Oct 30,1990
;* function * returns width or length of room which has space-name
;* argument * none
;*          * space-name :  name of space
;*          * worl       :  width or length
;**********************************************************************
;
(defun get-space-length   (space-name worl)
      (let ((gnum (car (get-group-number 'space-name space-name 'i-m-layout001))))
           (eval `(path-filler '(room ,gnum ,worl) 'i-m-layout001))))
;
(defun return-worl (direction)
      (let ((width  'width)
            (length 'length))
           (cond ((or (eq direction 'north) (eq direction 'south))
                  width)
                 (t length))))
;
(defun return-news (worl)
     (let ((north 'north)
           (east  'east))
          (cond ((eq worl 'width) north)
                (t east))))
```

```
;
(defun return-room-length (space-name direction)
       (declare (special *distance*))
       (let ((length (get-space-length   space-name (return-worl direction)))
             (mini   (car (get-minimum-length space-name direction))))
            (let ((differ (- length *distance*)))
                 (format t "~%~s of ~s ~s <minimum : ~s > => ~s" direction space-name len ↩
↪gth mini differ)
                 (cond ((>= differ  mini)
                        (setq *distance* 0) differ)
                       (t
                        (setq *distance* (- *distance* (- length mini))) mini)))))
;
(defun calc-group-minimum (group-list worl)
       (let ((space-name-list (for (element :in group-list)
                                        :save (cadr element)))
             (result 0))
            (dolist (space-name space-name-list result)
                    (setq result (+ result (car (get-minimum-length space-name (return-ne ↩
↪ws worl))))))))
;
(defun get-group-length  (mop)
       (let ((worl (role-filler 'equal (car (mop-absts mop))))
             (result 0)
             (space-name-list (for (element :in (mop-slots (role-filler 'argument1 mop)))
                                        :save (cadr element))))
            (dolist (space-name space-name-list result)
                    (setq result (+ result (get-space-length space-name worl))))))
;
(defun get-group-minimum (mop worl)
       (let ((slots (mop-slots mop)))
            (let ((minimum (role-slot 'minimum slots))
                  (current (role-slot 'current slots)))
                  (let ((argument-list (remove minimum slots)))
                       (setq argument-list (remove current argument-list))
; #DEBUG# => (format t "~&~s" argument-list)
                       (let ((mini-list (for (argument :in argument-list)
                                     :save (calc-group-minimum (mop-slots (cadr argum ↩
↪ent)) worl))))
            (format t "~%group-minimum => ~s" mini-list)
            (eval (append '(max) mini-list)))))))
;
(defun set-group-minimum (mop)
       (let ((mini (get-group-minimum mop (role-filler 'equal (car (mop-absts mop))))))
            (add-filler mop 'minimum 'value mini)))
;
(defun set-group-length  (mop)
       (let ((length (get-group-length mop)))
            (add-filler mop 'current 'value length)))
;
(defun group-current&minimum ()
 (let ((result1 nil))
     (let ((equal-worl (dolist (specs (mop-specs 'm-equal) result1)
                            (setq result1 (append result1 (mop-specs specs)))))
           (result2 nil))
          (format t "~%group-current&minimum => ~s" equal-worl)
           (dolist (worl equal-worl result2)
                   (let ((result3 nil))
                        (dolist (mop (mop-specs worl) result3)
                                (set-group-minimum mop)
                                (set-group-length  mop)))))))
;
;*******************************************************************************
```

```
;                      equal-group
;********************************************************************************
;* date     * Nov 6,1990
;* function * returns mop-name under m-equal which has space-name
;* argument * space-name :  name of space (ex. i-m-space101)
;*          * mop        :  name of mop under m-equal (ex. i-m-equal-width.1)
;********************************************************************************
;
(defun equal-group (space-name mop)
       (let ((argument-list (cdr (mop-slots mop))))
             (let ((result (for (argument :in argument-list)
                             :save (let ((group-list (mop-slots (cadr argument))))
                                         (let ((space-name-list (for (element :in group-li ↩
↪st)
                                                                       :save (cadr ↩
↪element))))
                                               (format t "~%equal-group => ~s" space-name- ↩
↪list)
                                               (cond ((eq (member space-name space-name-li ↩
↪st) nil) nil)
                                                      (t t)))))))
                   (cond ((eq (member 't result) nil) nil)
                          (t mop)))))
;
;********************************************************************************
;                      get-equal-group
;********************************************************************************
;* date     * Nov 6,1990
;* function * returns list of mop-name under m-equal which has space-name
;* argument * space-name :  name of space
;*          * direction  :  one of (north east south west)
;********************************************************************************
;
(defun get-equal-group (space-name direction)
       (let ((worl (return-worl direction))
             (result nil))
            (format t "~%worl => ~s" worl)
            (cond ((eq worl 'width)
                   (list-compress
                     (dolist (mop (mop-specs 'm-equal-width) result)
                             (setq result (append result (list (equal-group space-name mo ↩
↪p))))))
                  (t
                   (list-compress
                     (dolist (mop (mop-specs 'm-equal-length) result)
                             (setq result (append result (list (equal-group space-name mo ↩
↪p))))))))))
;
;
;*** check-perimeter
;
(defun check-perimeter ()
       (let ((not-passed 'not-passed)
             (result nil))
       (cond ((eq (check-perimeter-fn 'north 'width ) not-passed)
              (setq result (append result '(i-m-north-length)))))
       (cond ((eq (check-perimeter-fn 'east  'length) not-passed)
              (setq result (append result '(i-m-east-length)))))
       (cond ((eq (check-perimeter-fn 'south 'width ) not-passed)
              (setq result (append result '(i-m-south-length)))))
       (cond ((eq (check-perimeter-fn 'west  'length) not-passed)
              (setq result (append result '(i-m-west-length)))))
       result))
```

```
 ;
 ;*** check-perimeter-fn
 ;
 (defun check-perimeter-fn (direction worl)
     (declare (special *example-mop*))
     (let ((value (eval `(path-filler '(building-area ,worl) *example-mop*)))
           (slots (mop-slots (eval `(path-filler '(orientation ,direction) 'i-m-layout001) ⤸
⤷)))
           (passed     'passed)
           (not-passed 'not-passed)
           (result 0))
           (format t "~%Check => ~s of ~s" direction worl)
           (format t "~%~s" value)
 ; #DEBUG# => (format t "~%~s" slots)
           (dolist (slot slots result)
                   (let ((space-name (cadr slot)))
                        (let ((gnum (car (get-group-number 'space-name space-name 'i-m-lay ⤸
⤷out001))))
                             (setq result (+ result (eval `(path-filler '(room ,gnum ,wor ⤸
⤷l) 'i-m-layout001)))))))
           (format t "~%~s" result)
           (cond ((> result value) not-passed)
                 (t passed))))
 ;
 (defun get-conflict-direction ()
        (declare (special *compass*
                          *direction*))
        (setq *direction* nil)
        (let ((constraints (check-perimeter)))
             (format t "~%get-conflict-deirection : ~s" constraints)
             (for (constraint :in constraints)
                              :save (let ((direction (get *compass* constraint)))
                                         (setq *direction* (append *direction* (list dir ⤸
⤷ection)))
                                         (format t "~%Conflict direction => ~s" directio ⤸
⤷n)
                                         (cond ((or (eq direction 'north) (eq direction  ⤸
⤷'south))
                                                (set-distance direction 'width)
                                                (mapcar #'mop-specs (mop-specs 'm-equal- ⤸
⤷width)))
                                               (t
                                                (set-distance direction 'length)
                                                (mapcar #'mop-specs (mop-specs 'm-equal- ⤸
⤷length))))
                                        ))))
 ;
 (defun set-distance (direction worl)
     (declare (special *example-mop*
                       *distance*
                       *distance-queue*))
     (let ((value (eval `(path-filler '(building-area ,worl) *example-mop*)))
           (slots (mop-slots (eval `(path-filler '(orientation ,direction) 'i-m-layout001) ⤸
⤷)))
           (result 0))
           (format t "~%Check => ~s of ~s" direction worl)
           (format t "~%~s" value)
 ; #DEBUG# => (format t "~%~s" slots)
           (dolist (slot slots result)
                   (let ((space-name (cadr slot)))
                        (let ((gnum (car (get-group-number 'space-name space-name 'i-m-lay ⤸
⤷out001))))
                             (setq result (+ result (eval `(path-filler '(room ,gnum ,wor ⤸
↳l) 'i-m-layout001)))))))
          (let ((dist (- result value)))
                (setq *distance* dist)
                (setq *distance-queue* (append *distance-queue* (list dist))))))
 ;
 (defun partial-shrink-rooms (pattern mop)
        (declare (ignore  pattern
                          mop)
                 (special *direction*
                          *design-method*
                          *distance*
                          *distance-queue*
                          *new-room-list*))
        (setq *design-method* (append *design-method* (mop-specs 'partial-shrink-rooms)))
        (check-room-minimum)
        (length-limit)
        (group-current&minimum)
        (setq *new-room-list* (mop-slots (role-filler 'room 'i-m-layout001)))
        (let ((conflict-direction (get-conflict-direction))
              (result1 *direction*))
              (dolist (conflict-list conflict-direction result1)
                      (format t "~%*distance-queue* => ~s" *distance-queue*)
                      (format t "~%conflict-list => ~s" conflict-list)
                      (let ((result2 nil))
                             (dolist (mop-list conflict-list result2)
                               (let ((direction (car result1)))
                                     (setq *distance* (car *distance-queue*))
                                     (format t "~%partial-shrink direction => ~s" directi ↲
↳on)
                                     (cond ((or (eq direction 'north) (eq direction 'sout ↲
↳h))
                                            (shrink-equal-group mop-list 'width))
                                           (t
                                            (shrink-equal-group mop-list 'length)))))
                                    (setq result1 (cdr result1))
                                    (setq *distance-queue* (cdr *distance-queue*)))))
        (let ((room-list (for (element :in *new-room-list*)
                               :save (cadr element))))
              (format t "~%new-room-list => ~s" room-list)
              (list->group room-list)))
 ;
 (defun shrink-equal-group (mop-list worl)
        (declare (special *distance*
                          *alpha*))
        (format t "~%mop-list => ~s" mop-list)
        (for (mop :in mop-list)
                 :when (> *distance* 0)
                 :do   (let ((value1 (- (role-filler 'current mop) (role-filler 'minimum ↲
↳mop))))
                            (cond ((>= value1 *distance*)
                                   (setq *alpha* *distance*))
                                  (t
                                   (setq *alpha* value1)))
                                   (format t "~%*distance* => ~s" *distance*)
                                   (format t "~%value1     => ~s" value1)
                                   (format t "~%*alpha*    => ~s" *alpha*)
                                   (setq *distance* (- *distance* *alpha*))
                                   (let ((result nil)
                                         (slots (mop-slots mop)))
                                         (let ((minimum (role-slot 'minimum slots))
                                               (current (role-slot 'current slots)))
                                               (let ((argument-list (remove minimum slots ↲
↳)))
```

```
                                                     (format t "~%argument-list => ~s" arg ↲
↳ument-list)
                                                      (setq argument-list (remove current ↲
↳argument-list))
                                                      (dolist (argument argument-list resu ↲
↳lt)
                                                              (format t "~% ~s ----------> ↲
↳ " (car  argument))
                                                      (let ((space-name-list (mop-slots (c ↲
↳adr argument))))
                                                           (format t "~%~s" space-name-lis ↲
↳t)
                                                           (shrink-partial-room space-name ↲
↳-list *alpha* worl)))))))))
 ;
 (defun shrink-partial-room (space-name-list shrink-value worl)
        (declare (special *shrinked-room-list*))
        (let ((alpha shrink-value))
             (for (slot :in space-name-list)
                        :when (> alpha 0)
                        :do   (let ((space-name (cadr slot)))
                                   (let ((mini (cond ((eq worl 'width)
                                                       (car (get-minimum-length space-name ↲
↳ 'north)))
                                                      (t
                                                       (car (get-minimum-length space-name ↲
↳ 'east)))))
                                         (beta 0)
                                         (length (get-space-length space-name worl)))
                                         (let ((value (- length mini)))
                                              (cond ((>= value alpha)
                                                     (setq beta alpha))
                                                    (t
                                                     (setq beta value)))
                                              (format t "~%shrink-partial-room ~s beta = ↲
↳> ~s" space-name beta)
                                              (cond ((eq beta 0))
                                                    (t
                                                     (setq *shrinked-room-list* (append ↲
↳*shrinked-room-list* (list space-name)))))
                                              (format t "~%shrinked-room-list => ~s" *sh ↲
↳rinked-room-list*)
                                              (setq alpha (- alpha beta))
                                              (let ((new-length (- length beta)))
                                                   (create-shrinked-room space-name wor ↲
↳l new-length)))))))
 ;
 (defun create-shrinked-room (space-name worl new-length)
        (declare (special *example-mop*
                          *new-room-list*))
        (let ((result nil))
             (let ((gnum (dolist (element *new-room-list* result)
                                 (cond ((eq space-name (role-filler 'space-name (cadr elem ↲
↳ent)))
                                        (setq result (car element))))))
 ; #DEBUG# => (format t "~%gnum => ~s" gnum)
                  (let ((s-name (cadr (role-slot gnum *new-room-list*))))
                       (let ((width  (role-filler 'width  s-name))
                             (length (role-filler 'length s-name))
                             (use    (role-filler 'use    s-name))
                             (mop    (mop-absts s-name)))
                             (format t "~%old width  => ~s" width)
                             (format t "~%old length => ~s" length)
```

```
                                        (if (eq worl 'width)
                                            (setq  width new-length)
                                            (setq length new-length))
                                         (format t "~%new width  => ~s" width)
                                         (format t "~%new length => ~s" length)
                                        (let ((new-room (eval `(slots->mop '((space-name ,space-name ↩
↪)
                                                                                (width      ,width)
                                                                                (length     ,length)
                                                                                (use        ,use))
                                                                                ',mop t)))
                                        (result nil))
                                        (format t "~%new-room => ~s" new-room)
                                        (setq *new-room-list*
                                           (dolist (room *new-room-list* result)
                                                   (cond ((eq gnum (car room))
                                                          (setq result (append result (list (list gn ↩
↪um new-room)))))
                                                         (t
                                                          (setq result (append result (list room)))) ↩
↪)))))))))
 ;
 ;%%%%% * %%%%% * %%%%% * %%%%% * %%%%% * %%%%% * %%%%% * %%%%% * %%%%% * %%%%% * %%%%% * ↩
↪ %%%%% * %%%%%
 ;
 (format t "~%####################################")
 (format t "~%#                                 #")
 (format t "~%#    enter (example-layout001)    #")
 (format t "~%#                                 #")
 (format t "~%####################################")
 (defun example-layout001 ()
        (declare (special *example-mop*
                          *repaired-instance*))
 ;##### stage 1 ##########################################
 ;
 (format t "~%*** <step  1> sets building-area example 1 << width and length >>")
 ;
 (initial-values)
 ;
 (setq *example-mop* 'i-m-layout001)
 ;
 (news-length 'dummy-argument1 'dummy-argument2)          ;; This function is necessary fo ↩
↪r setting
 ;                                                        ;; *measure-length* and *measure ↩
↪-width*
 ;
 (draw-layout)                                            ;; Draws i-m-lyout001 layout gra ↩
↪ph
 ;
 (give-land '((width  76.0) (length  25.0)))
 ;
 (format t "~%*** <step  2> starts design procedures")
 ;
 (start-design)
 ;
 (format t "~%*** <step  3> checks space constraints")
 ;
 (get-space-constraint)
 ;
 (format t "~%*** <step  4> stores this case to case memory")
 ;
 (store-case)
 ;
```

```
;##### stage 2 ############################################
;
(format t "~%*** <step  5> sets building-area example 2 << width and length >>")
;
(initial-values)
;
(give-land '((width  72.2) (length  25.0)))
;
(format t "~%*** <step  6> gets preveous design technique from case memory")
;
(start-design)
;
(format t "~%*** <step  7> checks space constraints")
;
(get-space-constraint)
;
(format t "~%*** <step  8> stores this case to case memory")
;
(store-case)
;
(format t "~%*** <step  9> sets explanation mop instance")
;
(set-explanation)
;
(format t "~%*** <step  10> searches repaired solutions")
;
(get-repaired-solution)
;
;==> return (i-m-shrink i-m-eliminate) under m-repair
;
(format t "~%***<step  11> applys repaire procedures")
;
(mapcar #'repaired-steps (role-filler 'repaired-solution *repaired-instance*))
;
;##### stage 3 ############################################
;
(format t "~%*** <step 12> sets building-area example 3 << width and length >>")
;
(initial-values)
;
(give-land '((width  65.0) (length  24.0)))
;
(format t "~%*** <step 13> gets preveous design technique from case memory")
;
(start-design)
;
(format t "~%*** <step 14> checks space constraints")
;
(get-space-constraint)
;
(format t "~%*** <step 15> stores this case to case memory")
;
(store-case)
;
)
```

CBRE.Floor.Output.Defmop

```
;
;*******************************************************************************
;* MOPs definition for graphics
;*******************************************************************************
;* date     * Nov 15,1990
;*******************************************************************************
;
(defmop m-room-rectangle (m-root)
        (top-lx   nil)                        ;; x coordinate for top left corner of rec
tangle
        (top-ly   nil)                        ;; y coordinate for top left corner of rec
tangle
        (btm-rx   nil)                        ;; x coordinate for bottom right corner of
 rectangle
        (btm-ry   nil)                        ;; y coordinate for bottom right corner of
 rectangle
        (width    nil)                        ;; width  of rectangle
        (length   nil))                       ;; length of rectangle
;
(defmop rect101 (m-room-rectangle)            ;; rect101  - data for graphics -
        (top-lx   nil)
        (top-ly   nil)
        (btm-rx   nil)
        (btm-ry   nil)
        (width    nil)
        (length   nil))
;
(defmop rect102 (m-room-rectangle)            ;; rect102  - data for graphics -
        (top-lx   nil)
        (top-ly   nil)
        (btm-rx   nil)
        (btm-ry   nil)
        (width    nil)
        (length   nil))
;
(defmop rect103 (m-room-rectangle)            ;; rect103  - data for graphics -
        (top-lx   nil)
        (top-ly   nil)
        (btm-rx   nil)
        (btm-ry   nil)
        (width    nil)
        (length   nil))
;
(defmop rect104 (m-room-rectangle)            ;; rect104  - data for graphics -
        (top-lx   nil)
        (top-ly   nil)
        (btm-rx   nil)
        (btm-ry   nil)
        (width    nil)
        (length   nil))
;
(defmop rect105 (m-room-rectangle)            ;; rect105  - data for graphics -
        (top-lx   nil)
        (top-ly   nil)
        (btm-rx   nil)
        (btm-ry   nil)
        (width    nil)
        (length   nil))
;
(defmop rect106 (m-room-rectangle)            ;; rect106  - data for graphics -
        (top-lx   nil)
        (top-ly   nil)
        (btm-rx   nil)
        (btm-ry   nil)
        (width    nil)
        (length   nil))
;
(defmop rect107 (m-room-rectangle)            ;; rect107  - data for graphics -
        (top-lx   nil)
        (top-ly   nil)
        (btm-rx   nil)
        (btm-ry   nil)
        (width    nil)
        (length   nil))
;
(defmop rect108 (m-room-rectangle)            ;; rect108  - data for graphics -
        (top-lx   nil)
        (top-ly   nil)
        (btm-rx   nil)
        (btm-ry   nil)
        (width    nil)
        (length   nil))
;
(defmop rect109 (m-room-rectangle)            ;; rect109  - data for graphics -
        (top-lx   nil)
        (top-ly   nil)
        (btm-rx   nil)
        (btm-ry   nil)
        (width    nil)
        (length   nil))
;
(defmop rect110 (m-room-rectangle)            ;; rect110  - data for graphics -
        (top-lx   nil)
        (top-ly   nil)
        (btm-rx   nil)
        (btm-ry   nil)
        (width    nil)
        (length   nil))
;
(defmop rect111 (m-room-rectangle)            ;; rect111  - data for graphics -
        (top-lx   nil)
        (top-ly   nil)
        (btm-rx   nil)
        (btm-ry   nil)
        (width    nil)
        (length   nil))
;
(defmop rect112 (m-room-rectangle)            ;; rect112  - data for graphics -
        (top-lx   nil)
        (top-ly   nil)
        (btm-rx   nil)
        (btm-ry   nil)
        (width    nil)
        (length   nil))
;
(defmop rect113 (m-room-rectangle)            ;; rect113  - data for graphics -
        (top-lx   nil)
        (top-ly   nil)
        (btm-rx   nil)
        (btm-ry   nil)
        (width    nil)
        (length   nil))
;
(defmop m-rectangle  (m-root) mop)
(defmop top-lx       (m-rectangle) instance)        ;; x coordinate for top left corner
(defmop top-ly       (m-rectangle) instance)        ;; y coordinate for top right corner
```

```
 (defmop btm-rx        (m-rectangle) instance)         ;; x coordinate for top left corner
 (defmop btm-ry        (m-rectangle) instance)         ;; y coordinate for top right corner
 ;
 (defmop m-coordinate  (m-root)
                       (space-name  nil)               ;; name of space
                       (coordinate  nil))              ;; one of top-lx, top-ly, top-rx, an ↩
↪d top-ry
 ;
 (defmop m-coord-equal  (m-root)                       ;; MOPs for equal coordinate value
                        (argument1   nil)              ;; names of space and coordinate
                        (length    0))                 ;; length of side
 ;
 (defmop i-m-coord-equal.1 (m-coord-equal)
         (argument1 m-coordinate                       ;; =  top-lx of i-m-space101
                    (space-name i-m-space101)
                    (coordinate top-lx))
         (argument2 m-coordinate
                    (space-name i-m-space108)          ;; =  top-lx of i-m-space108
                    (coordinate top-lx))
         (length    0))
 ;
 (defmop i-m-coord-equal.2 (m-coord-equal)
         (argument1 m-coordinate
                    (space-name i-m-space102)          ;;  = top-lx of i-m-space102
                    (coordinate top-lx))
         (argument2 m-coordinate
                    (space-name i-m-space103)          ;;  = top-lx of i-m-space103
                    (coordinate top-lx))
         (argument3 m-coordinate
                    (space-name i-m-space101)          ;;  = btm-rx of i-m-space101
                    (coordinate btm-rx))
         (length    0))
 ;
 (defmop i-m-coord-equal.3 (m-coord-equal)
         (argument1 m-coordinate
                    (space-name i-m-space104)          ;;    top-lx of i-m-space104
                    (coordinate top-lx))
         (argument2 m-coordinate
                    (space-name i-m-space102)          ;;  = btm-rx of i-m-space102
                    (coordinate btm-rx))
         (argument3 m-coordinate
                    (space-name i-m-space103)          ;;  = btm-rx of i-m-space103
                    (coordinate btm-rx))
         (argument4 m-coordinate
                    (space-name i-m-space111)          ;;  = top-lx of i-m-space111
                    (coordinate top-lx))
         (argument5 m-coordinate
                    (space-name i-m-space109)          ;;  = btm-rx of i-m-space109
                    (coordinate btm-rx))
         (argument6 m-coordinate
                    (space-name i-m-space110)          ;;  = btm-rx of i-m-space110
                    (coordinate btm-rx))
         (length    0))
 ;
 (defmop i-m-coord-equal.4 (m-coord-equal)
         (argument1 m-coordinate
                    (space-name i-m-space105)          ;;  = top-lx of i-m-space105
                    (coordinate top-lx))
         (argument2 m-coordinate
                    (space-name i-m-space104)          ;;  = btm-rx of i-m-space104
                    (coordinate btm-rx))
         (length    0))
 ;
```

```
(defmop i-m-coord-equal.5 (m-coord-equal)
        (argument1 m-coordinate
                   (space-name i-m-space106)          ;;  = top-lx of i-m-space106
                   (coordinate top-lx))
        (argument2 m-coordinate
                   (space-name i-m-space105)          ;;  = btm-rx of i-m-space105
                   (coordinate btm-rx))
        (length    0))
;
(defmop i-m-coord-equal.6 (m-coord-equal)
        (argument1 m-coordinate
                   (space-name i-m-space107)          ;;  = top-lx of i-m-space107
                   (coordinate top-lx))
        (argument2 m-coordinate
                   (space-name i-m-space106)          ;;  = btm-rx of i-m-space106
                   (coordinate btm-rx))
        (argument3 m-coordinate
                   (space-name i-m-space113)          ;;  = btm-rx of i-m-space113
                   (coordinate btm-rx))
        (length    0))
;
(defmop i-m-coord-equal.7 (m-coord-equal)
        (argument1 m-coordinate
                   (space-name i-m-space109)          ;;  = top-lx of i-m-space109
                   (coordinate top-lx))
        (argument2 m-coordinate
                   (space-name i-m-space110)          ;;  = top-lx of i-m-space110
                   (coordinate top-lx))
        (argument3 m-coordinate
                   (space-name i-m-space108)          ;;  = btm-rx of i-m-space108
                   (coordinate btm-rx))
        (length    0))
;
(defmop i-m-coord-equal.8 (m-coord-equal)
        (argument1 m-coordinate
                   (space-name i-m-space112)          ;;  = top-lx of i-m-space112
                   (coordinate top-lx))
        (argument2 m-coordinate
                   (space-name i-m-space111)          ;;  = btm-rx of i-m-space111
                   (coordinate btm-rx))
        (length    0))
;
(defmop i-m-coord-equal.9 (m-coord-equal)
        (argument1 m-coordinate
                   (space-name i-m-space113)          ;;  = top-lx of i-m-space113
                   (coordinate top-lx))
        (argument2 m-coordinate
                   (space-name i-m-space112)          ;;  = btm-rx of i-m-space112
                   (coordinate btm-rx))
        (length    0))
;
(defmop i-m-coord-equal.10 (m-coord-equal)
        (argument1 m-coordinate
                   (space-name i-m-space101)          ;;  = top-ly of i-m-space101
                   (coordinate top-ly))
        (argument2 m-coordinate
                   (space-name i-m-space102)          ;;  = top-ly of i-m-space102
                   (coordinate top-ly))
        (argument3 m-coordinate
                   (space-name i-m-space104)          ;;  = top-ly of i-m-space104
                   (coordinate top-ly))
        (argument4 m-coordinate
                   (space-name i-m-space105)          ;;  = top-ly of i-m-space105
```

```
                (coordinate top-ly))
        (argument5 m-coordinate
                (space-name i-m-space106)          ;;  = top-ly of i-m-space106
                (coordinate top-ly))
        (argument6 m-coordinate
                (space-name i-m-space107)          ;;  = top-ly of i-m-space107
                (coordinate top-ly))
        (length    0))
;
(defmop i-m-coord-equal.11 (m-coord-equal)
        (argument1 m-coordinate
                (space-name i-m-space103)          ;;  = top-ly of i-m-space103
                (coordinate top-ly))
        (argument2 m-coordinate
                (space-name i-m-space102)          ;;  = btm-ry of i-m-space102
                (coordinate btm-ry))
        (length    0))
;
(defmop i-m-coord-equal.12 (m-coord-equal)
        (argument1 m-coordinate
                (space-name i-m-space108)          ;;  = top-ly of i-m-space108
                (coordinate top-ly))
        (argument2 m-coordinate
                (space-name i-m-space101)          ;;  = btm-ry of i-m-space101
                (coordinate btm-ry))
        (argument3 m-coordinate
                (space-name i-m-space110)          ;;  = top-ly of i-m-space110
                (coordinate top-ly))
        (argument4 m-coordinate
                (space-name i-m-space103)          ;;  = btm-ry of i-m-space103
                (coordinate btm-ry))
        (argument5 m-coordinate
                (space-name i-m-space111)          ;;  = top-ly of i-m-space111
                (coordinate top-ly))
        (argument6 m-coordinate
                (space-name i-m-space112)          ;;  = top-ly of i-m-space112
                (coordinate top-ly))
        (argument7 m-coordinate
                (space-name i-m-space104)          ;;  = btm-ry of i-m-space104
                (coordinate btm-ry))
        (argument8 m-coordinate
                (space-name i-m-space113)          ;;  = top-ly of i-m-space113
                (coordinate top-ly))
        (argument9 m-coordinate
                (space-name i-m-space105)          ;;  = btm-ry of i-m-space105
                (coordinate btm-ry))
        (argument10 m-coordinate
                (space-name i-m-space106)          ;;  = btm-ry of i-m-space106
                (coordinate btm-ry))
        (length    0))
;
(defmop i-m-coord-equal.13 (m-coord-equal)
        (argument1 m-coordinate
                (space-name i-m-space109)          ;;  = top-ly of i-m-space109
                (coordinate top-ly))
        (argument2 m-coordinate
                (space-name i-m-space110)          ;;  = btm-ry of i-m-space110
                (coordinate btm-ry))
        (length    0))
;
(defmop i-m-coord-equal.14 (m-coord-equal)
        (argument1 m-coordinate
                (space-name i-m-space108)          ;;  = btm-ry of i-m-space108
```

```
                    (coordinate btm-ry))
        (argument2 m-coordinate
                    (space-name i-m-space109)          ;;  = btm-ry of i-m-space109
                    (coordinate btm-ry))
        (argument3 m-coordinate
                    (space-name i-m-space111)          ;;  = btm-ry of i-m-space111
                    (coordinate btm-ry))
        (argument4 m-coordinate
                    (space-name i-m-space112)          ;;  = btm-ry of i-m-space112
                    (coordinate btm-ry))
        (argument5 m-coordinate
                    (space-name i-m-space113)          ;;  = btm-ry of i-m-space113
                    (coordinate btm-ry))
        (argument6 m-coordinate
                    (space-name i-m-space107)          ;;  = btm-ry of i-m-space107
                    (coordinate btm-ry))
        (length   0))
```

CBRE.Floor.Output.Defun

```
;################################################################################
;#####                        defrecord for color graphics                #####
;################################################################################
;
(eval-when (eval compile load)
  (require 'traps)
  (require 'records)
  (require 'quickdraw)

  (defrecord RGBColor
    (red :integer)
    (green :integer)
    (blue :integer))

  (defrecord ColorSpec
    (value :integer)
    (rgb :RGBColor))

  (defrecord (ColorTable :handle)
    (ctSeed :longint)
    (transIndex :integer)
    (ctSize :integer)
    (ctTable :colorSpec))

  (defrecord (PixMap :handle)
    (baseAddr :pointer)
    (rowBytes :integer)
    (bounds :rect)
    (xersion :integer)
    (packType :integer)
    (packSize :longint)
    (hRes :longint)
    (vRes :longint)
    (pixelType :integer)
    (pixelSize :integer)
    (cmpCount :integer)
    (cmpSize :integer)
    (planeBytes :longint)
    (Table :handle)
    (Reserved :longint))

  (defrecord (PixPat :handle)
    (patType :integer)
    (patMap :handle)
    (patData :handle)
    (patXData :handle)
    (patXValid :integer)
    (patXMap :handle)
    (pat1Data :pattern))

  (defrecord CGrafPort
    (device :integer)
    (portPixMap :handle)
    (portVersion :integer)
    (grafVars :handle)
    (chExtra :integer)
    (plLocHFrac :integer)
    (portRect :rect)
    (visRgn (region :handle))
    (clipRgn (region :handle))
    (bkPixPat :handle)
    (rgbFgColor :RGBColor)
    (rgbBkColor :RGBColor)
```

```
     (pnLoc point)
     (pnSize point)
     (pnMode integer)
     (pnPixPat :handle)
     (fillPixPat :handle)
     (pnVis integer)
     (txFont integer)
     (txFace integer)
     (txMode integer)
     (txSize integer)
     (spExtra longint)
     (fgColor longint)
     (bkcolor longint)
     (colrBit integer)
     (patStretch integer)
     (picSave handle)
     (rgnSave handle)
     (polySave handle)
     (grafProcs pointer))

   (defrecord (GrafDevice :handle)
     (RefNum :integer)
     (ID :integer)
     (Type :integer)
     (ITable :handle)
     (ResPref :integer)
     (SearchProc :pointer)
     (CompProc :pointer)
     (Flags :integer)
     (PMap :handle)
     (RefCon :longint)
     (NextGD :handle)
     (Rect :rect)
     (Mode :longint)
     (CCBytes :integer)
     (CCDepth :integer)
     (CCXData :handle)
     (CCXMask :handle)
     (Reserved :longint))

   (defrecord (CCrsr :handle)
     (Type :integer)
     (Map (:PixMap :handle))
     (Data :handle)
     (XData :handle)
     (XValid :integer)
     (XHandle :handle)
     (1Data :bits16)
     (Mask :bits16)
     (HotSpot :point)
     (XTable :longint)
     (ID :longint))

   (defrecord (CIcon :handle)
     (PMap :PixMap)
     (Mask :BitMap)
     (BMap :BitMap)
     (Data :handle)
     (MaskData :integer)))

(defobject *color-window* *window*)

(defobfun (color-window-p *window*) ()
    (eq #xc000 (logand #xc000 (rref wptr grafport.portbits.rowbytes))))

  (defobfun (exist *color-window*) (args)
    (let ((window-position (getf args :window-position #@(100 50)))
          (window-size (getf args :window-size #@(300 300))))
      (usual-exist
       (init-list-default
        args
        :window-position window-position
        :window-size window-size
        :wptr
        (rlet ((r :rect :topleft window-position
                  :bottomright (add-points window-position window-size)))
          (with-pstrs ((sp (getf args :window-title "Color Window")))
            (_NewCWindow :ptr nil
                         :ptr r
                         :ptr sp
                         :word (if (getf args :window-show) -1 0)
                         :word 0
                         :ptr nil
                         :word -1
                         :long 0
                         :ptr))))))
    (set-window-layer (getf args :window-layer 0)))

  (defobfun (set-bgcolor *color-window*) (red green blue)
    (rlet ((color :rgbcolor :red red :green green :blue blue))
      (with-port wptr
        (_rgbbackcolor :ptr color)
        (_eraserect :ptr (rref wptr window.portrect)))))

  (defobfun (set-fgcolor *color-window*) (red green blue)
    (rlet ((color :rgbcolor :red red :green green :blue blue))
      (with-port wptr
        (_rgbforecolor :ptr color))))

  (defobfun (set-pen-rgb-pat *color-window*) (red green blue)
    (let ((pixpat (%get-ptr wptr #x3a)))
      (rlet ((rgb :rgbcolor :red red :green green :blue blue))
        (_MakeRGBPat :ptr pixpat :ptr rgb))
      (_PenPixPat :ptr pixpat)))
  ;
  ;###############################################################################
  ;
  (setq *multiplier* 3)                                ;; initial value of *mulitiplier* ↩
↪3.0
  (setq *layout-origine* '(100 225))                   ;; origine of room layout
  ;
  ;*******************************************************************************
  ;* tranformation table between space name and rectangular number
  ;*******************************************************************************
  (setf *room-rectangle* nil)
  (setf (get *room-rectangle* 'i-m-space101) 'rect101)   ;; i-m-spacexxx  => recxxx
  (setf (get *room-rectangle* 'i-m-space102) 'rect102)
  (setf (get *room-rectangle* 'i-m-space103) 'rect103)
  (setf (get *room-rectangle* 'i-m-space104) 'rect104)
  (setf (get *room-rectangle* 'i-m-space105) 'rect105)
  (setf (get *room-rectangle* 'i-m-space106) 'rect106)
  (setf (get *room-rectangle* 'i-m-space107) 'rect107)
  (setf (get *room-rectangle* 'i-m-space108) 'rect108)
  (setf (get *room-rectangle* 'i-m-space109) 'rect109)
  (setf (get *room-rectangle* 'i-m-space110) 'rect110)
  (setf (get *room-rectangle* 'i-m-space111) 'rect111)
```

```
(setf (get *room-rectangle* 'i-m-space112) 'rect112)
(setf (get *room-rectangle* 'i-m-space113) 'rect113)
;
(setf (get *room-rectangle* 'rect101) 'i-m-space101)   ;; rectxxx       => i-m-spacexxx
(setf (get *room-rectangle* 'rect102) 'i-m-space102)
(setf (get *room-rectangle* 'rect103) 'i-m-space103)
(setf (get *room-rectangle* 'rect104) 'i-m-space104)
(setf (get *room-rectangle* 'rect105) 'i-m-space105)
(setf (get *room-rectangle* 'rect106) 'i-m-space106)
(setf (get *room-rectangle* 'rect107) 'i-m-space107)
(setf (get *room-rectangle* 'rect108) 'i-m-space108)
(setf (get *room-rectangle* 'rect109) 'i-m-space109)
(setf (get *room-rectangle* 'rect110) 'i-m-space110)
(setf (get *room-rectangle* 'rect111) 'i-m-space111)
(setf (get *room-rectangle* 'rect112) 'i-m-space112)
(setf (get *room-rectangle* 'rect113) 'i-m-space113)
;
;***********************************************************************
;                       open-output-window
;***********************************************************************
;* date     * Nov 10,1990
;* function * define output graphic window parameters
;* argument * none
;* var      * none
;***********************************************************************
;
(defun open-output-window ()
       (declare (special *output-graph*))
       (setf *output-graph* (oneof *color-window*
                 :window-position #@(200  40)
                 :window-size     #@(400 400)
                 :window-font     '("courier" 15)
                 :window-type     :tool
                 :window-title    "Output Window")))
;
;***********************************************************************
;                       get-coord-equal
;***********************************************************************
;* date     * Nov 10,1990
;* function * returens mop name under m-coord-equal which has space-name and
;*          * lx-ly
;* argument * space-name    : the name of the space < ex. i-m-space101 >
;*          * lx-ly         : one of top-lx, top-ly, btm-rx, and btm-ry
;***********************************************************************
;
(defun get-coord-equal (space-name lx-ly)
       (let ((coord-equal-list (mop-specs 'm-coord-equal))
             (result1 nil))
             (dolist (coord-equal coord-equal-list result1)
                     (let ((argument-list (reverse (cdr (reverse (mop-slots coord-equal)) ⤸
⤷)))
                           (result2 nil))
                           (dolist (argument argument-list result2)
                                   (let ((arg (car argument)))
                                        (cond  ((eq (eval `(path-filler '(,arg space-na ⤸
⤷me) ',coord-equal))  space-name)
                                                (cond ((eq (eval `(path-filler '(,arg c ⤸
⤷oordinate) ',coord-equal)) lx-ly)
                                                       (setq result1 coord-equal)))))) ⤸
⤷))))
;
;***********************************************************************
;                       calc-btm-rx
```

```
;**************************************************************************
;* date     * Nov 10,1990
;* function * calculates x-coordinate of bottom right corner of the rectangle
;* argument * rect-name     : the name of the rectangle < ex. rect101 >
;*          * top-lx        : x-coordinate of the top left corner of the rectangle
;**************************************************************************
;
(defun calc-btm-rx (rect-name top-lx)
       (declare (special *room-rectangle*
                         *multiplier*))
; #DEBUG# => (format t "~%--- calc-btm-rx ---")
; #DEBUG# => (format t "~% ~s top-lx : ~s  width : ~s" rect-name top-lx (car (get-values ↩
↪rect-name 'width)))
       (let ((space-name (get *room-rectangle* rect-name)))
             (let ((coord-equal (get-coord-equal space-name 'btm-rx))
                   (width (car (get-values rect-name 'width))))
; #DEBUG# => (format t "~%coord-equal : ~s " coord-equal)
                   (cond ((eq coord-equal nil)
                          (replace-value rect-name 'btm-rx
                             (round (+ top-lx (* *multiplier* width)))))
                         (t
                          (cond ((eq (role-filler 'length coord-equal) 0)
                                 (cond ((eq width nil)
                                        (replace-value rect-name 'btm-rx nil))
                                       (t
                                        (replace-value rect-name 'btm-rx
                                           (round (+ top-lx (* *multiplier* width)))
                                           :demons t))))
                                (t
                                 (replace-value rect-name 'btm-rx
                                    (round (role-filler 'length coord-equal)))))))))
;
;**************************************************************************
;                         calc-btm-ry
;**************************************************************************
;* date     * Nov 10,1990
;* function * calculates y-coordinate of bottom right corner of the rectangle
;* argument * rect-name     : the name of the rectangle < ex. rect101 >
;*          * top-ly        : y-coordinate of the top left corner of the rectangle
;**************************************************************************
;
(defun calc-btm-ry (rect-name top-ly)
       (declare (special *room-rectangle*
                         *multiplier*))
; #DEBUG# => (format t "~%--- calc-btm-ry ---")
; #DEBUG# => (format t "~% ~s top-ly : ~s length : ~s" rect-name top-ly (car (get-values ↩
↪rect-name 'length)))
       (let ((space-name (get *room-rectangle* rect-name)))
             (let ((coord-equal (get-coord-equal space-name 'btm-ry))
                   (length (car (get-values rect-name 'length))))
; #DEBUG# => (format t "~%coord-equal : ~s " coord-equal)
                   (cond ((eq coord-equal nil)
                          (replace-value rect-name 'btm-ry
                             (round (+ top-ly (* *multiplier* length)))))
                         (t
                          (cond ((eq (role-filler 'length coord-equal) 0)
                                 (cond ((eq length nil)
                                        (replace-value rect-name 'btm-ry nil))
                                       (t
                                        (replace-value rect-name 'btm-ry
                                           (round (+ top-ly (* *multiplier* length)))
                                           :demons t))))
                                (t
                                    (replace-value rect-name 'btm-ry
                                       (round (role-filler 'length coord-equal)))))))))
 ;
 ;********************************************************************************
 ;                        set-coord-equal
 ;********************************************************************************
 ;* date     * Nov 10,1990
 ;* function * sets the coordinate value of bottom right corner of the rectangle
 ;* argument * rect-name      : the name of the rectangle < ex. rect101 >
 ;*          * rx-ry          : one of btm-rx or btm-ry
 ;*          * value          : the vlaue of the coordinate
 ;********************************************************************************
 ;
 (defun set-coord-equal (rect-name rx-ry value)
         (declare (special *room-rectangle*
                           *multiplier*))
 ; #DEBUG# => (format t "~%--- set-coord-equal ---")
 ; #DEBUG# => (format t "~% ~s : ~s : ~s" rect-name rx-ry value)
         (let ((space-name (get *room-rectangle* rect-name)))
               (let ((coord-equal (get-coord-equal space-name rx-ry)))
                     (cond ((eq (car (get-values coord-equal 'length)) 0)
                            (replace-value coord-equal 'length value :demons t)))
                     (cond ((eq rx-ry 'btm-rx)
                            (cond ((eq (car (get-values rect-name 'top-lx)) nil)
                                    (let ((coord-equal1 (get-coord-equal space-name 'top-lx) ↩
↪)
                                          (width (car (get-values rect-name 'width))))
                                          (cond ((eq width nil))
                                                (t
                                                (cond ((eq (car (get-values coord-equal1 'le ↩
↪ngth)) 0)
 ; #DEBUG# => (format t "~%replace : ~s" coord-equal1)
                                                       (replace-value coord-equal1 'length
                                                          (round (- value (* *multiplier* wi ↩
↪dth)))
                                                           :demons t))))))))
                           ((eq rx-ry 'btm-ry)
                            (cond ((eq (car (get-values rect-name 'top-ly)) nil)
                                    (let ((coord-equal2 (get-coord-equal space-name 'top-ly) ↩
↪)
                                          (length (car (get-values rect-name 'length))))
                                          (cond ((eq length nil))
                                               (t
                                                (cond ((eq (car (get-values coord-equal2 'le ↩
↪ngth)) 0)
 ; #DEBUG# => (format t "~%replace : ~s" coord-equal2)
                                                       (replace-value coord-equal2 'length
                                                          (round (- value (* *multiplier* le ↩
↪ngth)))
                                                           :demons t))))))))))))
 ;
 ;********************************************************************************
 ;                        demons
 ;********************************************************************************
 ;* date     * Nov 10,1990
 ;* function * sets x-coordinate value of bottom right corner of the rectangle
 ;*          * when x-coordinate value of top left corner was set
 ;* argument * !frame         : current frame name = the name of the rectangle
 ;*          * !filler        : current filler     = x-coordinate value of
 ;*          *                                       top left corner
 ;********************************************************************************
 ;
 (add-filler 'rect101 'top-lx 'if-added                          ;; for rect101
    '(calc-btm-rx !frame !filler))
(add-filler 'rect101 'top-ly 'if-added
    '(calc-btm-ry !frame !filler))
(add-filler 'rect101 'btm-rx 'if-added
    '(set-coord-equal !frame !slot !filler))
(add-filler 'rect101 'btm-ry 'if-added
    '(set-coord-equal !frame !slot !filler))
;
(add-filler 'rect102 'top-lx 'if-added                    ;; for rect102
    '(calc-btm-rx !frame !filler))
(add-filler 'rect102 'top-ly 'if-added
    '(calc-btm-ry !frame !filler))
(add-filler 'rect102 'btm-rx 'if-added
    '(set-coord-equal !frame !slot !filler))
(add-filler 'rect102 'btm-ry 'if-added
    '(set-coord-equal !frame !slot !filler))
;
(add-filler 'rect103 'top-lx 'if-added                    ;; for rect103
    '(calc-btm-rx !frame !filler))
(add-filler 'rect103 'top-ly 'if-added
    '(calc-btm-ry !frame !filler))
(add-filler 'rect103 'btm-rx 'if-added
    '(set-coord-equal !frame !slot !filler))
(add-filler 'rect103 'btm-ry 'if-added
    '(set-coord-equal !frame !slot !filler))
;
(add-filler 'rect104 'top-lx 'if-added                    ;; for rect104
    '(calc-btm-rx !frame !filler))
(add-filler 'rect104 'top-ly 'if-added
    '(calc-btm-ry !frame !filler))
(add-filler 'rect104 'btm-rx 'if-added
    '(set-coord-equal !frame !slot !filler))
(add-filler 'rect104 'btm-ry 'if-added
    '(set-coord-equal !frame !slot !filler))
;
(add-filler 'rect105 'top-lx 'if-added                    ;; for rect105
    '(calc-btm-rx !frame !filler))
(add-filler 'rect105 'top-ly 'if-added
    '(calc-btm-ry !frame !filler))
(add-filler 'rect105 'btm-rx 'if-added
    '(set-coord-equal !frame !slot !filler))
(add-filler 'rect105 'btm-ry 'if-added
    '(set-coord-equal !frame !slot !filler))
;
(add-filler 'rect106 'top-lx 'if-added                    ;; for rect106
    '(calc-btm-rx !frame !filler))
(add-filler 'rect106 'top-ly 'if-added
    '(calc-btm-ry !frame !filler))
(add-filler 'rect106 'btm-rx 'if-added
    '(set-coord-equal !frame !slot !filler))
(add-filler 'rect106 'btm-ry 'if-added
    '(set-coord-equal !frame !slot !filler))
;
(add-filler 'rect107 'top-lx 'if-added                    ;; for rect107
    '(calc-btm-rx !frame !filler))
(add-filler 'rect107 'top-ly 'if-added
    '(calc-btm-ry !frame !filler))
(add-filler 'rect107 'btm-rx 'if-added
    '(set-coord-equal !frame !slot !filler))
(add-filler 'rect107 'btm-ry 'if-added
    '(set-coord-equal !frame !slot !filler))
;
(add-filler 'rect108 'top-lx 'if-added                    ;; for rect108
     '(calc-btm-rx !frame !filler))
 (add-filler 'rect108 'top-ly 'if-added
     '(calc-btm-ry !frame !filler))
 (add-filler 'rect108 'btm-rx 'if-added
     '(set-coord-equal !frame !slot !filler))
 (add-filler 'rect108 'btm-ry 'if-added
     '(set-coord-equal !frame !slot !filler))
;
 (add-filler 'rect109 'top-lx 'if-added                          ;; for rect109
     '(calc-btm-rx !frame !filler))
 (add-filler 'rect109 'top-ly 'if-added
     '(calc-btm-ry !frame !filler))
 (add-filler 'rect109 'btm-rx 'if-added
     '(set-coord-equal !frame !slot !filler))
 (add-filler 'rect109 'btm-ry 'if-added
     '(set-coord-equal !frame !slot !filler))
;
 (add-filler 'rect110 'top-lx 'if-added                          ;; for rect110
     '(calc-btm-rx !frame !filler))
 (add-filler 'rect110 'top-ly 'if-added
     '(calc-btm-ry !frame !filler))
 (add-filler 'rect110 'btm-rx 'if-added
     '(set-coord-equal !frame !slot !filler))
 (add-filler 'rect110 'btm-ry 'if-added
     '(set-coord-equal !frame !slot !filler))
;
 (add-filler 'rect111 'top-lx 'if-added                          ;; for rect111
     '(calc-btm-rx !frame !filler))
 (add-filler 'rect111 'top-ly 'if-added
     '(calc-btm-ry !frame !filler))
 (add-filler 'rect111 'btm-rx 'if-added
     '(set-coord-equal !frame !slot !filler))
 (add-filler 'rect111 'btm-ry 'if-added
     '(set-coord-equal !frame !slot !filler))
;
 (add-filler 'rect112 'top-lx 'if-added                          ;; for rect112
     '(calc-btm-rx !frame !filler))
 (add-filler 'rect112 'top-ly 'if-added
     '(calc-btm-ry !frame !filler))
 (add-filler 'rect112 'btm-rx 'if-added
     '(set-coord-equal !frame !slot !filler))
 (add-filler 'rect112 'btm-ry 'if-added
     '(set-coord-equal !frame !slot !filler))
;
 (add-filler 'rect113 'top-lx 'if-added                          ;; for rect113
     '(calc-btm-rx !frame !filler))
 (add-filler 'rect113 'top-ly 'if-added
     '(calc-btm-ry !frame !filler))
 (add-filler 'rect113 'btm-rx 'if-added
     '(set-coord-equal !frame !slot !filler))
 (add-filler 'rect113 'btm-ry 'if-added
     '(set-coord-equal !frame !slot !filler))
;
;**********************************************************************************
;                         demons
;**********************************************************************************
;* date     * Nov 10,1990
;* function * sets x or y coordinate value of top left or bottom right corner
;*          * when they were not set yet
;* argument * !frame        : current frame name = the name of the rectangle
;*          * !filler       : current filler     = x or y coordinate value of
;*          *                                      top left or bottom right corner
;**********************************************************************************
```

```
 ;
 (add-filler 'i-m-coord-equal.1 'length  'if-added
     '(replace-topbtm !frame !filler))
 ;
 (add-filler 'i-m-coord-equal.2 'length  'if-added
     '(replace-topbtm !frame !filler))
 ;
 (add-filler 'i-m-coord-equal.3 'length  'if-added
     '(replace-topbtm !frame !filler))
 ;
 (add-filler 'i-m-coord-equal.4 'length  'if-added
     '(replace-topbtm !frame !filler))
 ;
 (add-filler 'i-m-coord-equal.5 'length  'if-added
     '(replace-topbtm !frame !filler))
 ;
 (add-filler 'i-m-coord-equal.6 'length  'if-added
     '(replace-topbtm !frame !filler))
 ;
 (add-filler 'i-m-coord-equal.7 'length  'if-added
     '(replace-topbtm !frame !filler))
 ;
 (add-filler 'i-m-coord-equal.8 'length  'if-added
     '(replace-topbtm !frame !filler))
 ;
 (add-filler 'i-m-coord-equal.9 'length  'if-added
     '(replace-topbtm !frame !filler))
 ;
 (add-filler 'i-m-coord-equal.10 'length 'if-added
     '(replace-topbtm !frame !filler))
 ;
 (add-filler 'i-m-coord-equal.11 'length 'if-added
     '(replace-topbtm !frame !filler))
 ;
 (add-filler 'i-m-coord-equal.12 'length 'if-added
     '(replace-topbtm !frame !filler))
 ;
 (add-filler 'i-m-coord-equal.13 'length 'if-added
     '(replace-topbtm !frame !filler))
 ;
 ;*****************************************************************************
 ;                          replace-topbtm
 ;*****************************************************************************
 ;* date     * Nov 10,1990
 ;* function * sets x or y coordinate value of top left or bottom right corner
 ;*          * when they were not set yet
 ;* argument * coord-equal   : mop name under m-coord-equal < ex. i-m-coord-equal.1 >
 ;*          * value         : coordinate value
 ;*****************************************************************************
 ;
 (defun replace-topbtm (coord-equal value)
        (declare (special *room-rectangle*))
 ; #DEBUG# => (format t "~%--- replace-topbtm ---")
 ; #DEBUG# => (format t "~%~s : ~s " coord-equal value)
        (let ((argument-list (reverse (cdr (reverse (mop-slots coord-equal)))))
              (result nil))
              (dolist (argument argument-list result)
                    (let ((coordinate (role-filler 'coordinate (cadr argument)))
                          (rect-name (get *room-rectangle* (role-filler 'space-name (cad ⟩
⟨r argument)))))
                         (cond ((eq coordinate 'top-lx)
                                (cond ((eq (car (get-values rect-name 'top-lx)) nil)
                                       (replace-value rect-name 'top-lx value :demons t) ⟩
¬)))
                                ((eq coordinate 'top-ly)
                                 (cond ((eq (car (get-values rect-name 'top-ly)) nil)
                                       (replace-value rect-name 'top-ly value :demons t) ¬
¬)))
                                ((eq coordinate 'btm-rx)
                                 (cond ((eq (car (get-values rect-name 'btm-rx)) nil)
 ; #DEBUG# => (format t "~% ~s btm-rx" rect-name)
                                       (replace-value rect-name 'btm-rx value :demons t) ¬
¬)))
                                ((eq coordinate 'btm-ry)
                                 (cond ((eq (car (get-values rect-name 'btm-ry)) nil)
 ; #DEBUG# => (format t "~% ~s btm-ry" rect-name)
                                       (replace-value rect-name 'btm-ry value :demons t) ¬
¬))))))))
 ;
 ;****************************************************************************
 ;                      get-room-worl
 ;****************************************************************************
 ;* date     * Nov 12,1990
 ;* function * returns width or length of the room
 ;* argument * none
 ;****************************************************************************
 ;
 (defun get-room-worl ()
        (declare (special *example-mop*
                          *room-rectangle*))
        (let ((space-list (mop-specs 'm-space-name))
              (result nil))
             (dolist (space-name space-list result)
                    (let ((gnum (car (get-group-number 'space-name space-name *example-m ¬
¬op*))))
 ; #DEBUG# => (format t "~%gnum => ~s" gnum)
                         (let ((width  (eval `(path-filler '(room ,gnum width ) *example ¬
¬-mop*)))
                               (length (eval `(path-filler '(room ,gnum length) *example ¬
¬-mop*)))
                               (rect   (get *room-rectangle* space-name)))
 ; #DEBUG# => (format t "~%name => ~s : width = ~s --- length = ~s" rect width length)
                               (replace-value rect 'width  width)
                               (replace-value rect 'length length)
                               )))))
 ;
 ;****************************************************************************
 ;                      draw-room
 ;****************************************************************************
 ;* date     * Nov 12,1990
 ;* function * draws rectangular room in *output-graph* window
 ;* argument * rect-name     : the name of rectangle < ex. rect101 >
 ;* var      * top-lx        : x-coordinate of top left corner of the rectangle
 ;*          * top-ly        : y-coordinate of top left corner of the rectangle
 ;*          * btm-rx        : x-coordinate of bottom right corner of the rectangle
 ;*          * btm-ry        : y-coordinate of top left corner of the rectangle      ¬
¬
 ;****************************************************************************
 ;
 (defun draw-room (rect-name)
       (declare (special *output-graph*))
                (let ((top-lx (role-filler 'top-lx rect-name))
                      (top-ly (role-filler 'top-ly rect-name))
                      (btm-rx (role-filler 'btm-rx rect-name))
                      (btm-ry (role-filler 'btm-ry rect-name)))
                (cond ((eq (or (eq top-lx nil)
                             (eq top-ly nil)
                             (eq btm-rx nil)
                             (eq btm-ry nil)) t))
                      ( t
                       (ask *output-graph* (window-select))
                       (ask *output-graph* (set-pen-rgb-pat 31300 40500 36100))
                       (ask *output-graph* (move-to top-lx top-ly))
                       (ask *output-graph* (paint-rect top-lx top-ly btm-rx btm-ry))
                       (ask *output-graph* (set-pen-pattern *black-pattern*))
                       (ask *output-graph* (set-pen-size '#@(1 1)))
                       (ask *output-graph* (move-to top-lx top-ly)
                       (ask *output-graph* (frame-rect top-lx top-ly btm-rx btm-ry))))) ↩
↪)
 ;
 ;*************************************************************************************
 ;                      paint-room
 ;*************************************************************************************
 ;* date     * Dec 17,1990
 ;* function * paints rectangular room in *output-graph* window
 ;* argument * rect-name     : the name of rectangle < ex. rect101 >
 ;*          * red           : red number
 ;*          * green         : green number
 ;*          * blue          : blue number
 ;* var      * top-lx        : x-coordinate of top left corner of the rectangle
 ;*          * top-ly        : y-coordinate of top left corner of the rectangle
 ;*          * btm-rx        : x-coordinate of bottom right corner of the rectangle
 ;*          * btm-ry        : y-coordinate of top left corner of the rectangle      ↩
↪
 ;*************************************************************************************
 ;
 (defun paint-room (rect-name red green blue)
       (declare (special *output-graph*))
              (cond ((eq rect-name nil))
                    (t
                     (let ((top-lx (role-filler 'top-lx rect-name))
                           (top-ly (role-filler 'top-ly rect-name))
                           (btm-rx (role-filler 'btm-rx rect-name))
                           (btm-ry (role-filler 'btm-ry rect-name)))
                           (cond ((eq (or (eq top-lx nil)
                                 (eq top-ly nil)
                                 (eq btm-rx nil)
                                 (eq btm-ry nil)) t))
                                ( t
                                 (ask *output-graph* (window-select))
                                 (ask *output-graph* (set-pen-rgb-pat red green blue)) ↩
↪
                                 (ask *output-graph* (move-to top-lx top-ly))
                                 (ask *output-graph* (paint-rect top-lx top-ly btm-rx ↩
↪btm-ry))
                                 (ask *output-graph* (set-pen-pattern *black-pattern*) ↩
↪)
                                 (ask *output-graph* (set-pen-size '#@(1 1)))
                                 (ask *output-graph* (move-to top-lx top-ly)
                                 (ask *output-graph* (frame-rect top-lx top-ly btm-rx ↩
↪btm-ry))))))))
 ;
 ;*************************************************************************************
 ;                      erase-room
 ;*************************************************************************************
 ;* date     * Nov 12,1990
 ;* function * erases rectangular room in *output-graph* window
 ;* argument * rect-name     : the name of rectangle < ex. rect101 >
 ;* var      * top-lx        : x-coordinate of top left corner of the rectangle
;*          * top-ly        : y-coordinate of top left corner of the rectangle
;*          * btm-rx        : x-coordinate of bottom right corner of the rectangle
;*          * btm-ry        : y-coordinate of top left corner of the rectangle      ↴
↳
;*********************************************************************************
;
(defun erase-room (rect-name)
      (declare (special *output-graph*))
               (cond ((eq rect-name nil))
                     (t
                      (let ((top-lx (role-filler 'top-lx rect-name))
                            (top-ly (role-filler 'top-ly rect-name))
                            (btm-rx (role-filler 'btm-rx rect-name))
                            (btm-ry (role-filler 'btm-ry rect-name)))
                            (cond ((eq (or (eq top-lx nil)
                                   (eq top-ly nil)
                                   (eq btm-rx nil)
                                   (eq btm-ry nil)) t))
                                  ( t
                                   (ask *output-graph* (window-select))
                                   (ask *output-graph* (set-pen-pattern *white-pattern*) ↴
↳)
                                   (ask *output-graph* (move-to top-lx top-ly)
                                   (ask *output-graph* (paint-rect top-lx top-ly btm-rx  ↴
↳btm-ry))))))))
;
;*********************************************************************************
;                      draw-rect
;*********************************************************************************
;* date     * Nov 12,1990
;* function * draws room layout in *output-graph* window
;* argument * none
;* var      * argument-list : list of argument-slot under example                     ↴
↳
;*********************************************************************************
;
(defun draw-rect ()
      (declare (special *example-mop*
                        *output-graph*
                        *space-name-list*
                        *room-position*
                        *rect-name-list*))
      (without-interrupts
      (let ((argument-list (mop-slots (role-filler 'room *example-mop*)))
            (result nil))
            (setq *space-name-list* (for (argument   :in argument-list)
                                         :save (role-filler 'space-name (cadr argument))) ↴
↳)
            (setq *rect-name-list*  (for (space-name :in *space-name-list*)
                                         :save (get *room-rectangle* space-name)))
                     (dolist (rect-name *rect-name-list* result)
                             (draw-room rect-name)
                             (let ((top-lx (role-filler 'top-lx rect-name))
                                   (top-ly (role-filler 'top-ly rect-name))
                                   (btm-rx (role-filler 'btm-rx rect-name))
                                   (btm-ry (role-filler 'btm-ry rect-name)))
                                   (setf result (append result (list (list top-lx top-l ↴
↳y btm-rx btm-ry))))))
            (setf *room-position* result)))
            (defobfun (window-click-event-handler *output-graph*) (where)      ;; get c ↴
↳lick point
                     (declare (special *beep-sound*
                                       *example-mop*
```

```
                                              *output-graph*
                                              *room-position*
                                              *space-name-list*
                                              *rect-name-list*
                                              *click-room*
                                              *room-rectangle*
                                              *pop-up-slots*))
                            (erase-room *click-room*)
                            (draw-room  *click-room*)
                            (let ((xx (point-h where))
                                  (yy (point-v where))
                                  (result 0)
                                  (space-position 0))
                                  (setf *beep-sound* nil)
                                  (dolist (element *room-position* result)
                                          (let ((x-min (first  element)) (x-max (third element ⤵
⤷))
                                                (y-min (second element)) (y-max (fourth elemen ⤵
⤷t)))
                                               (cond ((and (and (> xx x-min) (< xx x-max))
                                                           (and (> yy y-min) (< yy y-max)))
                                                      (setf *beep-sound* (or *beep-sound*  t))
                                                         (let ((result1 0) (space-order *space ⤵
⤷-name-list*))
                                                              (dotimes (count space-position r ⤵
⤷esult1)
                                                                       (setf space-order (cdr  ⤵
⤷space-order)))
                                                              (let ((space-name (car space-ord ⤵
⤷er)))
                                                                   (let ((gnum (car (get-group ⤵
⤷-number 'space-name space-name *example-mop*))))
                                                                         (setq *click-room* (g ⤵
⤷et *room-rectangle* space-name))
                                                                         (ask *output-graph* ( ⤵
⤷set-pen-pattern *gray-pattern*))
                                                                         (ask *output-graph* ( ⤵
⤷move-to    x-min y-min))
                                                                         (ask *output-graph* ( ⤵
⤷paint-rect x-min y-min x-max y-max))
                                                                         (pop-up-slots (eval ` ⤵
⤷(path-filler '(room ,gnum) *example-mop*)))
                                                                         (window-frame-color * ⤵
⤷pop-up-slots* 4600 23300 23500)
                                                                         (define-redraw-window ⤵
⤷ *pop-up-slots*)
                                                                         (print-pop-up-slots ( ⤵
⤷eval `(path-filler '(room ,gnum) *example-mop*)))))))
                                                     (t (setf result (+ 1 result))
                                                        (setf *beep-sound* (or *beep-sound*   ⤵
⤷nil))

                                                        (setf space-position result))))))
                            (when (eql *beep-sound* nil) (ed-beep))))
;
;*********************************************************************************
;*                      draw-compass
;*********************************************************************************
;* date     * May 22, 1990
;* function * Draws a compass at the upper right corner
;* argument * none
;* var      * x  : x coordinate of a frame-oval (upper left corner)
;             y  : y coordinate of a frame-oval (upper left corner)
;*********************************************************************************
```

```
;
(defun draw-compass (x y)
          (declare (special *output-graph*))
          (ask *output-graph* (window-select))
          (ask *output-graph* (set-pen-pattern *black-pattern*))
          (ask *output-graph* (set-pen-size '#@(1 1)))
          (ask *output-graph* (move-to x y))
          (ask *output-graph* (frame-oval x y (+ x 30) (+ y 30)))      ;; draw circle
          (ask *output-graph* (move-to (- x 10) (+ y 15)))
          (ask *output-graph* (line-to (+ x 40) (+ y 15)))
          (ask *output-graph* (move-to (+ x 15) (- y 20)))             ;; draw arrow
          (ask *output-graph* (line-to (+ x 15) (+ y 45)))
          (ask *output-graph* (move-to (+ x 15) (- y 20)))
          (ask *output-graph* (line-to (+ x 10) (- y  5)))
          (ask *output-graph* (line-to (+ x 20) (- y  5)))
          (ask *output-graph* (move-to (+ x 10) (- y 25)))
          (ask *output-graph* (set-window-font '("times" 15)))
          (ask *output-graph* (princ "N" *output-graph*)))
;
;*************************************************************************************
;                      window-frame-color
;*************************************************************************************
;* date     * Nov 12,1990
;* function * sets frame color of window
;* argument * window        : the name of window
;* var      * red           : red value
;*          * green         : green value
;*          * blue          : blue value
;*************************************************************************************
;
(defun window-frame-color (window red green blue)
       (let ((color (make-color red green blue)))
            (ask window (set-part-color :frame     color))
            (ask window (set-part-color :hilite    color))
            (ask window (set-part-color :text      color))))
;
;*************************************************************************************
;                      calc-measure
;*************************************************************************************
;* date     * Nov 20,1990
;* function * calculates coordinate of scale measure lines
;* argument * none
;* var      * top-lx        : x-coordinate of top left corner of the rectangle
;*          * top-ly        : y-coordinate of top left corner of the rectangle
;*          * btm-rx        : x-coordinate of bottom right corner of the rectangle
;*          * btm-ry        : y-coordinate of top left corner of the rectangle
;*************************************************************************************
;
(defun calc-measure ()
       (declare (special *room-position*
                         *t-list*
                         *v-list*))
       (let ((result 0)
             (top-lx-list nil)
             (top-ly-list nil)
             (btm-rx-list nil)
             (btm-ry-list nil))
            (dolist (element *room-position* result)
                    (setq top-lx-list (append top-lx-list (list (first  element))))
                    (setq top-ly-list (append top-ly-list (list (second element))))
                    (setq btm-rx-list (append btm-rx-list (list (third  element))))
                    (setq btm-ry-list (append btm-ry-list (list (fourth element)))))
```

```
                   (let ((min-x (eval (append '(min) top-lx-list)))
                         (min-y (eval (append '(min) top-ly-list)))
                         (max-x (eval (append '(max) btm-rx-list)))
                         (max-y (eval (append '(max) btm-ry-list))))
                        (let ((point1x min-x) (point1y min-y)
                              (point2x min-x) (point2y max-y)
                              (point3x max-x) (point3y max-y))
                            (setq *t-list* (append *t-list*
                                          (list (list (- point1x 30)   point1y     ⤸
⤷(- point1x 10)    point1y))))
                            (setq *t-list* (append *t-list*
                                          (list (list (- point1x 20)   point1y     ⤸
⤷(- point1x 20) (+ point1y 25)))))
                            (setq *t-list* (append *t-list*
                                          (list (list (- point2x 30)   point2y     ⤸
⤷(- point2x 10)    point2y))))
                            (setq *t-list* (append *t-list*
                                          (list (list (- point2x 20)   point2y     ⤸
⤷(- point2x 20) (- point2y 25)))))
                            (setq *t-list* (append *t-list*
                                          (list (list    point2x    (+ point2y 10) ⤸
⤷   point2x     (+ point2y 30)))))
                            (setq *t-list* (append *t-list*
                                          (list (list    point2x    (+ point2y 20) ⤸
⤷(+ point2x 75) (+ point2y 20)))))
                            (setq *t-list* (append *t-list*
                                          (list (list    point3x    (+ point3y 10) ⤸
⤷   point3x     (+ point3y 30)))))
                            (setq *t-list* (append *t-list*
                                          (list (list    point3x    (+ point3y 20) ⤸
⤷(- point3x 75) (+ point3y 20)))))
                            (setq *v-list* (append *v-list*
                                          (list (list (- point1x 30) (+ (round (* 0 ⤸
⤷.5 (+ point1y point2y))) 5)))))
                            (setq *v-list* (append *v-list*
                                          (list (list (- (round (* 0.5 (+ point2x p ⤸
⤷oint3x))) 10)  (+ point2y 25)))))
                  ))))
;
;***********************************************************************
;                     draw-measure
;***********************************************************************
;* date     * Nov 20,1990
;* function * draws scale measures in *output-graph* window
;* argument * none
;***********************************************************************
;
(defun draw-measure ()
      (declare (special *t-list*
                        *v-list*
                        *measure-length*
                        *measure-width*
                        *output-graph*))
      (let ((result 0))
           (ask *output-graph* (window-select))
           (ask *output-graph* (set-pen-pattern *black-pattern*))
           (ask *output-graph* (set-pen-size '#@(1 1)))
           (dolist (element *t-list* result)
                   (let ((x1 (first element)) (y1 (second element))
                         (x2 (third element)) (y2 (fourth element)))
                        (ask *output-graph* (move-to x1 y1))
                        (ask *output-graph* (line-to x2 y2))))
           (let ((result (list (eval (append '(max) *measure-length*))
                         (eval (append '(max) *measure-width*)))))
               (dolist (element *v-list* result)
                 (let ((x (first element)) (y (second element))
                     (value (car result)))
                     (ask *output-graph* (move-to x y))
                     (ask *output-graph* (set-window-font '("times" 10)))
                     (ask *output-graph* (princ value *output-graph*))
                     (setq result (cdr result)))))))
;
;*********************************************************************************
;                       draw-output-frame
;*********************************************************************************
;* date     * Nov 28,1990
;* function * draws rectangular frame around room layout output in *output-graph* window
;* argument * none
;*********************************************************************************
;
(defun draw-output-frame ()
       (declare (special *output-graph*))
       (ask *output-graph* (window-select))
       (ask *output-graph* (set-pen-rgb-pat 23400 30200 27200))
       (ask *output-graph* (set-pen-size '#@(2 2)))
       (ask *output-graph* (move-to 10 150))
       (ask *output-graph* (frame-rect 10 175 390 375)))
;
;*********************************************************************************
;                       put-case-info
;*********************************************************************************
;* date     * Nov 30,1990
;* function * puts the information of the case for output
;* argument * rect-name     : the name of rectangle < ex. rect101 >
;* var      * case          : the name of case
;*          * result        : the result of case
;*          * reason        : the reason of failure
;*          * width         : width of building area
;*          * length        : length of building area
;*********************************************************************************
;
(defun put-case-info ()
       (declare (special *output-graph*))
       (let ((case-name (car (mop-specs 'm-case))))
            (let ((case     (car (mop-absts (role-filler 'case-name case-name))))
                  (result   (role-filler 'result    case-name))
                  (reason   (role-filler 'reason    case-name))
                  (width    (path-filler '(after building-area width)  case-name))
                  (length   (path-filler '(after building-area length) case-name)))
                  (ask *output-graph* (window-select))
                  (ask *output-graph* (set-pen-pattern *black-pattern*))
                  (ask *output-graph* (set-pen-size '#@(1 1)))
                  (ask *output-graph* (set-window-font '("times" 10)))
                  (ask *output-graph* (move-to  10  20))
                  (ask *output-graph* (princ "CASE NAME" *output-graph*))
                  (ask *output-graph* (move-to 100  20))
                  (ask *output-graph* (princ ":" *output-graph*))
                  (ask *output-graph* (move-to 110  20))
                  (ask *output-graph* (princ case *output-graph*))
                  (ask *output-graph* (move-to  10  32))
                  (ask *output-graph* (princ "RESULT" *output-graph*))
                  (ask *output-graph* (move-to 100  32))
                  (ask *output-graph* (princ ":" *output-graph*))
                  (ask *output-graph* (move-to 110  32))
                  (ask *output-graph* (princ result *output-graph*))
                  (ask *output-graph* (move-to  10  44))
```

```
                    (ask *output-graph* (princ "REASON" *output-graph*))
                    (ask *output-graph* (move-to 100  44))
                    (ask *output-graph* (princ ":" *output-graph*))
                    (ask *output-graph* (move-to 110  44))
                    (ask *output-graph* (princ reason *output-graph*))
                    (ask *output-graph* (move-to  10  56))
                    (ask *output-graph* (princ "REQUIRED AREA" *output-graph*))
                    (ask *output-graph* (move-to  10  68))
                    (ask *output-graph* (princ "       WIDTH" *output-graph*))
                    (ask *output-graph* (move-to 100  68))
                    (ask *output-graph* (princ ":" *output-graph*))
                    (ask *output-graph* (move-to 110  68))
                    (ask *output-graph* (princ width *output-graph*))
                    (ask *output-graph* (move-to  10  80))
                    (ask *output-graph* (princ "       LENGTH" *output-graph*))
                    (ask *output-graph* (move-to 100  80))
                    (ask *output-graph* (princ ":" *output-graph*))
                    (ask *output-graph* (move-to 110  80))
                    (ask *output-graph* (princ length *output-graph*))
                    (ask *output-graph* (move-to 15 127)
                    (ask *output-graph* (princ "Click the inside of the room !" *output-gra ⤸
⤷ph*))
                    (ask *output-graph* (set-pen-pattern *dark-gray-pattern*))
                    (ask *output-graph* (set-pen-size '#@(2 2)))
                    (ask *output-graph* (move-to 10 115))
                    (ask *output-graph* (frame-rect 10 115 180 135))))))
 ;
 ;*****************************************************************************
 ;                     paint-conflict-area
 ;*****************************************************************************
 ;* date     * Dec 17,1990
 ;* function * paints conflict areas which are hit at constraint-check
 ;* argument * none
 ;*****************************************************************************
 ;
 (defun paint-conflict-area ()
       (declare (special *example-mop*
                         *compass*
                         *warning-msg*
                         *space-check-table*))
       (setq *warning-msg* nil)                                    ;; clear *warning- ⤸
⤷msg*
       (let ((constraint (path-filler '(steps constraint-check) (car (mop-specs 'm-case)) ⤸
⤷)))

           (cond ((eq constraint 'i-m-empty-group)
                  (format t "~%*** This is empty ***"))
                 (t
                  (let ((slots (mop-slots constraint))
                        (result nil))
                        (let ((area-list (for (slot :in slots)
                                              :save (cadr slot))))
                             (princ area-list)
                             (dolist (element area-list result)
                                 (cond ((eq element 'i-m-total-space)
                                        (format t "~%*** total check ***")
                                        (conflict-total-size)
                                        (setq *warning-msg* (append *warning-msg* '(tota ⤸
⤷l-space))))
                                       ((or (eq element 'i-m-north-length)
                                            (eq element 'i-m-east-length)
                                            (eq element 'i-m-south-length)
                                            (eq element 'i-m-west-length))
```

```
                                                  (format t "~%*** length check *** ~S" elemen ⤸
⤷t)
                                                  (setq *warning-msg* (append *warning-msg* '( ⤸
⤷side-length)))
                                                  (let ((direction (get *compass* element)))
                                                       (let ((slots (mop-slots (eval `(path-fi ⤸
⤷ller '(orientation ,direction) ',*example-mop*)))))
                                                            (let ((space-list (for (slot :in s ⤸
⤷lots)
                                                                                          :save ⤸
⤷(cadr slot))))
 ; #DEBUG# => (format t "~%~s" space-list)
                                                                 (for (space-name :in space-l ⤸
⤷ist)
                                                                           :do (let ((rect-n ⤸
⤷ame (get *room-rectangle* space-name)))
                                                                                     (confli ⤸
⤷ct-length rect-name direction)))))))
                                             (t
                                              (let ((gnum-list (get-group-number 'use (get *sp ⤸
⤷ace-check-table* element) *example-mop*)))
 ; #DEBUG# => (format t "~%=> ~s" gnum-list)
                                                      (setq *warning-msg* (append *warning-msg ⤸
⤷* '(room-size)))
                                                      (for (gnum :in gnum-list)
                                                                :do (let ((space-name (eval ` ⤸
⤷(path-filler '(room ,gnum space-name) ',*example-mop*))))
 ; #DEBUG# => (format t "~%------ ~s" space-name)
                                                                        (paint-room (get *ro ⤸
⤷om-rectangle* space-name) 40500 22700 23100))))))
                                         )))
                         (format t "~%warning-msg => ~s" *warning-msg*)))))
 ;
 ;*********************************************************************************
 ;                        paint-shrinked-room
 ;*********************************************************************************
 ;* date     * Feb 12,1991
 ;* function * paints partially shrinked rooms
 ;* argument * none
 ;*********************************************************************************
 ;
 (defun paint-shrinked-room ()
        (declare (special *shrinked-room-list*
                          *warning-msg*
                          *room-rectangle*))
        (cond ((eq *shrinked-room-list* nil))
              (t
               (remove-duplicates *shrinked-room-list*)
               (for (space-name :in *shrinked-room-list*)
                                :do (paint-room (get *room-rectangle* space-name) 63000 58 ⤸
⤷300 22000))
               (setq *warning-msg* (append *warning-msg* '(partial-shrink)))
               (format t "~%warning-msg => ~s" *warning-msg*))))
 ;
 ;*********************************************************************************
 ;                        conflict-total-size
 ;*********************************************************************************
 ;* date     * Dec 17,1990
 ;* function * draws warining rectangle when result is bigger than required total size
 ;* argument * none
 ;*********************************************************************************
 ;
 (defun conflict-total-size ()
       (declare (special *layout-origine*
                         *example-mop*
                         *multiplier*
                         *output-graph*))
       (let ((top-lx (car  *layout-origine*))
             (top-ly (cadr *layout-origine*))
             (width  (round (* *multiplier* (path-filler '(building-area width)  *example ⤸
⤷-mop*))))
             (length (round (* *multiplier* (path-filler '(building-area length) *example ⤸
⤷-mop*)))))
             (ask *output-graph* (window-select))
             (ask *output-graph* (set-pen-size '#@(2 2)))
             (ask *output-graph* (set-pen-rgb-pat 40500 22700 23100))
             (ask *output-graph* (move-to top-lx top-ly))
                        (ask *output-graph* (frame-rect top-lx top-ly (+ top-lx width) (+ ⤸
⤷ top-ly length)))))
 ;
 ;*************************************************************************************
 ;                     conflict-length
 ;*************************************************************************************
 ;* date     * Dec 17,1990
 ;* function * draw warning triangle when result broke required side length
 ;* argument * rect-name     : name of rectangle < ex. rect101 >
 ;*          * direction     : name of direction < one of north, east, south, and west >  ⤸
⤷
 ;*************************************************************************************
 ;
 (defun conflict-length (rect-name direction)
       (let ((top-lx (role-filler 'top-lx rect-name))
             (top-ly (role-filler 'top-ly rect-name))
             (btm-rx (role-filler 'btm-rx rect-name))
             (btm-ry (role-filler 'btm-ry rect-name))
             (half-btm 5)
             (height   5))
            (cond ((eq direction 'north)
                   (let ((point1x (+ top-lx (round (* 0.5 (- btm-rx top-lx)))))
                         (point1y (- top-ly 2)))
                        (let ((point2x (+ point1x half-btm))
                              (point2y (- point1y height  )))
                             (let ((point3x (- point1x half-btm))
                                   (point3y point2y))
                                  (paint-triangle point1x point1y
                                                  point2x point2y
                                                  point3x point3y)))))
                  ((eq direction 'east)
                   (let ((point1x (+ btm-rx 2))
                         (point1y (- btm-ry (round (* 0.5 (- btm-ry top-ly))))))
                        (let ((point2x (+ point1x height))
                              (point2y (+ point1y half-btm)))
                             (let ((point3x (+ point1x height))
                                   (point3y (- point1y half-btm)))
                                  (paint-triangle point1x point1y
                                                  point2x point2y
                                                  point3x point3y)))))
                  ((eq direction 'south)
                   (let ((point1x (- btm-rx (round (* 0.5 (- btm-rx top-lx)))))
                         (point1y (+ btm-ry 2)))
                        (let ((point2x (- point1x half-btm))
                              (point2y (+ point1y height)))
                             (let ((point3x (+ point1x half-btm))
                                   (point3y (+ point1y height)))
                                  (paint-triangle point1x point1y
                                                  point2x point2y
```

```
                                                point3x point3y)))))
                    ((eq direction 'west)
                     (let ((point1x (- top-lx 2))
                           (point1y (- top-ly (rounf (* 0.5 (- btm-ry top-ly))))))
                          (let ((point2x (- point1x height))
                                (point2y (- point1y half-btm)))
                               (let ((point3x (- point1x height))
                                     (point3y (- point1y half-btm)))
                                    (paint-triangle point1x point1y
                                                    point2x point2y
                                                    point3x point3y))))))))
;
;*********************************************************************************
;                        paint-triangle
;*********************************************************************************
;* date     * Dec 17,1990
;* function * draw painted triangle with color
;* argument * point1x      : x coordinate of first  vertex
;*          * point1y      : y coordinate of first  vertex
;*          * point2x      : x coordinate of second vertex
;*          * point2y      : y coordinate of second vertex
;*          * point3x      : x coordinate of third  vertex
;*          * point3y      : y coordinate of third  vertex
;*********************************************************************************
;
(defun paint-triangle (point1x point1y point2x point2y point3x point3y)
       (declare (special *output-graph*))
                        (ask *output-graph* (window-select))
                        (ask *output-graph* (set-pen-rgb-pat 40500 22700 23100))
                        (ask *output-graph* (move-to point1x point1y))
                        (ask *output-graph* (start-polygon))
                        (ask *output-graph* (line-to point2x point2y))
                        (ask *output-graph* (line-to point3x point3y))
                        (ask *output-graph* (line-to point1x point1y))
                        (let ((polygon (ask *output-graph* (get-polygon))))
                             (ask *output-graph* (paint-polygon polygon))
                             (ask *output-graph* (set-pen-pattern *black-pattern*))
                             (ask *output-graph* (set-pen-size '#@(1 1)))
                             (ask *output-graph* (move-to point1x point1y)
                             (ask *output-graph* (frame-polygon polygon)))))
;
;*********************************************************************************
;                        warning-message
;*********************************************************************************
;* date     * Feb 12,1991
;* function * draw warning message on output-graphics window
;* argument * startx       : x coordinate of warning start point
;*          * starty       : y coordinate of warning start point                  ⤸
⤷
;*********************************************************************************
;
(defun warning-message (startx starty)
       (declare (special *output-graph*
                         *warning-msg*))
                        (setq *warning-msg* (remove-duplicates *warning-msg*))
                        (ask *output-graph* (window-select))
                        (let ((result 0))
                             (dolist (warning *warning-msg* result)
                                     (cond ((eq warning 'total-space)
                                            (ask *output-graph* (set-pen-rgb-pat 40500 22 ⤸
⤷700 23100))
                                            (ask *output-graph* (move-to startx (+ starty ⤸
⤷ result)))
```

```
↩ (+ startx 10) (+ starty 10)))

↩" 10)))

↩attern*))

↩ 10 (+ starty result))))

↩Area Size" *output-graph*))



↩700 23100))

↩ result)))

↩ (+ starty result)

↩ (+ 10 (+ starty result))))

↩" 10)))

↩attern*))

↩ result)))

↩ (+ starty result)

↩ (+ 10 (+ starty result))))

↩ 10 (+ starty result))))

↩ize" *output-graph*))



↩700 23100))

↩ result)))

↩t)

↩ result))

↩ result)))

↩" 10)))

↩attern*))

↩ 10 (+ starty result))))

↩ength" *output-graph*))



↩300 22000))

↩ result)))
```

```
 (ask *output-graph* (frame-rect startx starty ↩

 (ask *output-graph* (set-pen-size '#@(1 1)))
 (ask *output-graph* (set-window-font '("times ↩

 (ask *output-graph* (set-pen-pattern *black-p ↩

 (ask *output-graph* (move-to (+ startx 20) (+ ↩

 (ask *output-graph* (princ "Warning! : Total  ↩

 (setq result (+ result 20)))
((eq warning 'room-size)
 (ask *output-graph* (set-pen-rgb-pat 40500 22 ↩

 (ask *output-graph* (move-to startx (+ starty ↩

 (ask *output-graph* (paint-rect startx        ↩

                                   (+ startx 10) ↩

 (ask *output-graph* (set-pen-size '#@(1 1)))
 (ask *output-graph* (set-window-font '("times ↩

 (ask *output-graph* (set-pen-pattern *black-p ↩

 (ask *output-graph* (move-to startx (+ starty ↩

 (ask *output-graph* (frame-rect startx        ↩

                                   (+ startx 10) ↩

 (ask *output-graph* (move-to (+ startx 20) (+ ↩

 (ask *output-graph* (princ "Warning! : Room S ↩

 (setq result (+ result 20)))
((eq warning 'side-length)
 (ask *output-graph* (set-pen-rgb-pat 40500 22 ↩

 (ask *output-graph* (move-to startx (+ starty ↩

 (paint-triangle (+ startx  5) (+ starty resul ↩

                 startx        (+ 10 (+ starty ↩

                 (+ startx 10) (+ 10 (+ starty ↩

 (ask *output-graph* (set-pen-size '#@(1 1)))
 (ask *output-graph* (set-window-font '("times ↩

 (ask *output-graph* (set-pen-pattern *black-p ↩

 (ask *output-graph* (move-to (+ startx 20) (+ ↩

 (ask *output-graph* (princ "Warning! : Side L ↩

 (setq result (+ result 20)))
((eq warning 'partial-shrink)
 (ask *output-graph* (set-pen-rgb-pat 63000 58 ↩

 (ask *output-graph* (move-to startx (+ starty ↩
                                                (ask *output-graph* (paint-rect startx        ↲
↳ (+ starty result)
                                                                                (+ startx 10) ↲
↳ (+ 10 (+ starty result))))
                                                (ask *output-graph* (set-pen-size '#@(1 1)))
                                                (ask *output-graph* (set-window-font '("times ↲
↳" 10)))
                                                (ask *output-graph* (set-pen-pattern *black-p ↲
↳attern*))
                                                (ask *output-graph* (frame-rect startx        ↲
↳  (+ starty result)
                                                                                (+ startx 10) ↲
↳  (+ 10 (+ starty result))))
                                                (ask *output-graph* (move-to (+ startx 20) (+ ↲
↳ 10 (+ starty result))))
                                                (ask *output-graph* (princ "Shrinked!" *outpu ↲
↳t-graph*))
                                                (setq result (+ result 20)))))))
;
;*************************************************************************************
;                       init-room-rectangle
;*************************************************************************************
;* date     * Dec 17,1990
;* function * initializes instances of m-room-rectangle and m-coord-equal
;* argument * none
;*************************************************************************************
;
(defun init-room-rectangle ()
       (let ((rect-name-list (mop-specs 'm-room-rectangle))
             (result nil))
            (dolist (rect-name rect-name-list result)
                    (replace-value rect-name 'top-lx nil)
                    (replace-value rect-name 'top-ly nil)
                    (replace-value rect-name 'btm-rx nil)
                    (replace-value rect-name 'btm-ry nil)
                    (replace-value rect-name 'width  nil)
                    (replace-value rect-name 'length nil)))
       (let ((coord-equal-list (mop-specs 'm-coord-equal))
             (result nil))
            (dolist (coord-equal coord-equal-list result)
                    (replace-value coord-equal 'length 0))))
;
;*************************************************************************************
;                       store-output-graph
;*************************************************************************************
;* date     * Nov 25,1990
;* function * stores the procedure of drawing on *output-graph* window
;* argument * none
;*************************************************************************************
;
(defun store-output-graph ()
       (declare (special *output-graph*))
       (ask *output-graph* (start-picture))
       (put-case-info)
       (draw-compass 350 40)
       (draw-output-frame)
       (draw-rect)
       (paint-conflict-area)
       (paint-shrinked-room)
       (warning-message 240 115)
       (draw-measure)
       (ask *output-graph* (have 'saved-pict (ask *output-graph* (get-picture)))))
;
```

```
;*******************************************************************************
;                          draw-layout
;*******************************************************************************
;* date     * Nov 30,1990
;* function * makes *output-graph* window to run drawing functions
;* argument * none
;*******************************************************************************
;
(defun draw-layout ()
       (declare (special *room-position*
                         *space-name-list*
                         *rect-name-list*
                         *click-room*
                         *t-list*
                         *v-list*
                         *measure-length*
                         *measure-width*
                         *layout-origine*
                         *output-graph*))
       (init-room-rectangle)
       (get-room-worl)
                                                          ;; x coordinate of the origin of la ⤸
⤷yout output
       (replace-value 'i-m-coord-equal.1  'length (car  *layout-origine*) :demons t)
                                                          ;; y coordinate of the origin of la ⤸
⤷yout output
       (replace-value 'i-m-coord-equal.10 'length (cadr *layout-origine*) :demons t)
       (open-output-window)
 ;
       (setq *room-position*   nil)                       ;; the list of the top left corner  ⤸
⤷and the bottom
                                                          ;; right corner of coordinates of t ⤸
⤷he rectangle
       (setq *space-name-list* nil)                       ;; the name of the space      < ex. ⤸
⤷ space101 >
       (setq *rect-name-list*  nil)                       ;; the name of the rectangle  < ex. ⤸
⤷ rect101  >
       (setq *click-room*      nil)                       ;; the name of the rectangle which  ⤸
⤷was clicked
       (setq *t-list*          nil)                       ;; the list of measure line coordin ⤸
⤷ates
       (setq *v-list*          nil)                       ;; the list of measure values
 ;
       (define-redraw-window *output-graph*)
       (window-frame-color *output-graph* 15000 9300 29000)
       (put-case-info)
       (draw-compass 350 45)
       (draw-output-frame)
       (draw-rect)
       (paint-conflict-area)
       (paint-shrinked-room)
       (warning-message 240 115)
       (calc-measure)
       (draw-measure)
       (store-output-graph))
```