Integrating AI and Conventional Computing

By

Darren William Miller

A Thesis
Submitted to the Faculty of Graduate Studies
in Partial Fulfillment of the Requirements
for the Degree of

MASTER OF SCIENCE

Department of Computer Science
University of Manitoba
Winnipeg, Manitoba

ISBN 0-315-76876-2

INTEGRATING AI and CONVENTIONAL COMPUTING

BY

DARREN WILLIAM MILLER

A thesis submitted to the Faculty of Graduate Studies of
the University of Manitoba in partial fulfillment of the requirements
of the degree of

MASTER OF SCIENCE

Abstract

Many tasks that programmers need to implement may be solved using conventional computing techniques. More complex tasks, however, do not conform as nicely to such techniques. EX-C is an extended programming environment developed around the third generation programming language, C. It provides a fully integrated development environment for conventional and rule-based programming. This thesis discusses the components of EX-C, how they are implemented, and how they are used. These components include language extensions to C, a translator program called RBX that creates C code from extended C code, a host of run-time support routines that include the rule base inference engines, and an integrated shell program (KA) that automates many tasks and ties the package together.

Acknowledgements

I would like to thank Steve Schmidt for all his contributions to this project. Steve wrote and designed the original forward chaining system and designed the initial rule base language extensions.

I would also like to thank Pierre Doyle for his help in writing the initial version of the RBX translator.

As well, I would like to thank Dr. David Scuse and Dr. Harry Lakser for sitting on my comittee.

I would especially like to thank my supervisor, Dr. Mark Evans, for his leadership, guidance, and abundance of ideas. Without him, this thesis would not have been possible.

Lastly, I would like to thank my parents for all the support and guidance they have given me over the years of my life.

TABLE OF CONTENTS

Chapter One : Introduction

Until recently, most expert systems for micro-computers have been developed using either special purpose expert system building tools (GURU, PC-PLUS, NEXPERT-OBJECT, etc.) or the traditional artificial intelligence languages LISP and PROLOG [Jackson, 1991]. As the field has matured, there has been a trend towards integrating expert systems technology with conventional computing [Hu, 1989], [Butler et al., 1988], [Cohen, 1989]. This trend is evident in newer commercial tools which are providing links to conventional programming languages (C, Pascal, COBOL, etc.) and database systems (DBASE, ORACLE, INGRES, etc.) [Payne, et al., 1990]. This thesis explores the integration of expert systems technology with conventional computing from a different perspective. It details a set of extensions to an existing C language programming environment for IBM-compatible computers to facilitate the use of rule-based programming techniques. This development environment, called EX-C, allows application programs to be constructed using both conventional and expert systems techniques. Subsets of applications best suited to expert systems techniques can be represented using appropriate expert system modules, while more conventionally oriented components can be represented using conventional techniques [Franke, 1990]. Consequently, programming flexibility is increased and a natural integration of expert systems techniques and conventional techniques can be achieved based on the characteristics of target applications [Brown, 1990].

1.1 Problems With PC-based Expert System Tools:

The popularity of commercial rule-based expert system development tools (or shells) has grown dramatically [Rauch-Hinden, 1988], [Firebaugh, 1988]. (For the purposes of this thesis, let us concentrate on micro-computer-based tools only.) However, application systems have faced certain limitations imposed by the tools they were developed with (their

author tools) [Mettrey, 1991], [Fox, 1990]. These limitations include: poor explanation facilities (i.e. explanations at the level the knowledge is represented, which is not necessarily how the domain expert views the knowledge), user interface standards imposed by the author tool (rather than the developer), requirements that run-time versions of the author tools be present in order to use applications, unnatural or impossible integration with existing systems, and fixed or difficult to modify inference engines [Mettrey, 1991].

First generation knowledge-based systems were typically stand-alone with respect to previously-developed systems (i.e. they are not integrated into previously-developed systems) [Waterman, 1986]. As a result, they required their author tool to be present in order to function. This implies that the shell (at least a run-time version) must be purchased by each user of the application. Imagine if a run-time version of Cobol was needed for every user of a Cobol application. This limitation imposed by many tools affects both independent and in-house developers in their ability to distribute their completed systems.

All expert system development tools utilize some subset of AI programming techniques. However, most tools do not provide conventional programming facilities, and if they do, they are limited or unnatural in the way they must be used. Some tools provide conventional programming with an internal procedural language. For example, GURU provides a pseudo procedural language that can be used in conjunction with its rule-based programming language [Holsapple, et al., 1986]. Thus, many knowledge-based systems developed with these tools (especially rule-based systems) are implemented completely non-procedurally. Since, many aspects of a complete computer system can and should be implemented conventionally, it does not make sense to attempt to do so using AI programming technology.

The interface of a computer system is a very important component [Rich, 1984]. Often, the

most natural layout of a system's data input screens, displays, and reports is as unique as the system itself. Thus, it is usually necessary to customize these parts of a system. This is traditionally done most naturally with procedural code. With expert system development tools, user interface code is generally built in and hidden from the developer. Inference engines may prompt for data as it is required in a fashion that cannot be controlled by the developer. Most tools usually provide facilities for viewing data and results after and during rule base execution, but again these facilities are usually built in and unchangeable [Liebowitz, 1988]. Thus, it is difficult to provide a good, customized user interface to a knowledge-based computer system because of limitations imposed by the tools. These limitations will become even more prevalent as more and more knowledge-based systems become available. Their users will have to adapt to the many varieties of poor interfaces. As standards for user interfaces evolve, we may see many tools adopting them which may lessen the impact of this current limitation.

One important characteristic of knowledge-based systems is their ability to explain how their results were obtained [Waterman, 1986]. However, the explanation facilities found in most commercial tools are limited [Mettrey, 1991]. Often the user can view a listing of the rules fired or not fired and display the values of various variables and their attributes [Turban, 1988]. However, such *explanations* explain at the knowledge representation level (the level at which knowledge is encoded), not at the knowledge level (the level at which the domain expert understands the problem). Mechanisms are needed to generate more intelligent explanations based on this information and other information that many tools do not provide. Such explanations would typically be simple to generate using a good procedural language, but again many of these tools do not provide a procedural language, or do not give developers access to internal information such as which rules were and were not fired, and which clauses of those rules failed and succeeded at each of those instances. Thus, such intelligent or flexible explanations are difficult if not impossible to generate.

This limitation can affect the acceptance of a knowledge-based system and its ability to properly explain its operation [Waterman, 1988]. For example, a typical user would rather be given a explanation more like what a domain expert could give rather than the listing of a rule that fired to produce a particular result. The latter, more primitive, explanation may be fine for a user who understands AI technology, but will most likely be unacceptable to one who does not. The rule trace that results from a rule base call must be available for both the tool to generate a generic explanation (this is what most tools provide), but also must be available to a developer's code so that customized explanations may be generated.

Many tools provide a fixed set of inference engines used to process the knowledge in a knowledge base [Gervarter, 1987]. Almost exclusively, these expert system development tools do not provide a mechanism to modify their inference engines (other than modifying their source code). Only some of the low-end tools provide their source code. However, particular applications may require specialized means of processing their knowledge bases. Most likely, these applications will require the addition of meta-knowledge to "fake" these modifications. However, this is an unnatural way of building a system that can lead to extra effort needed to test and maintain such meta-knowledge. On the other hand, other tools offer many options and switches that can control the way their inference engines work. Such tools generally are expensive. An example of such a tool is NEXPERT-OBJECT by Neuron Data (see Figure 1.1). However, it is often unclear how combinations of these switches interact with one another. A user of such a tool typically requires extensive experience with that tool before such switches can be used to effectively alter the way its inference engines work. Also, no matter how complex a tool may be, it will still be fixed in the number of different combinations that may be used.

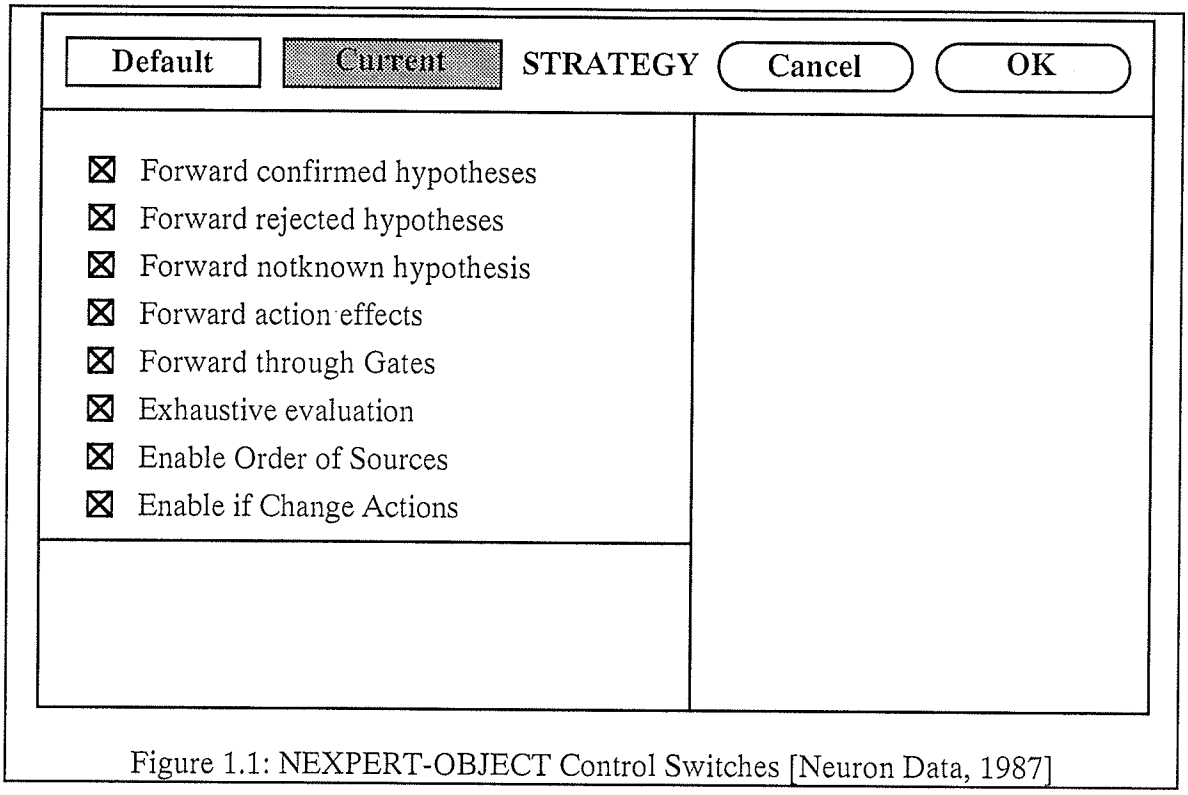

Figure 1.1: NEXPERT-OBJECT Control Switches [Neuron Data, 1987]

A growing concern with knowledge-based systems is the ability to integrate them with existing systems [Jakobson et al., 1986]. An increasing number of tools allow their knowledge bases both to call and be called by external procedures and functions [Amir, 1989]. However, the data used by the knowledge-based component is usually only loosely coupled to the data used by the other components of the system. For example, MDBS's GURU uses an interface to C called *KC* [Holsapple, et al., 1986]. This interface allows a C program to copy data from a GURU table file. The C program can then modify it and write it back if necessary. No direct access of these tables is given to the C program. A more desirable mechanism would be one that allows a tighter coupling between the data used by a knowledge base and the data used by the rest of the application or the external routines that may be called by the knowledge base. More and more tools are providing integration facilities [Leaman, 1989]. However very few exhibit the *tight coupling* of data discussed above.

The goal of this thesis is to provide a tool that allows AI programming techniques (specifically rule-based programming and the ability to produce custom explanations) to be used in conjunction with conventional programming techniques. In addition to this, the tool should provide an easy to use prototyping environment that allows an initial prototype of a rule base to be entered, tested, and completed with as little work as possible. The tool should then allow such rule bases to be easily embedded within applications being developed concurrently.

1.2 Short Survey of Commercial Tools:

To provide more basis for the work presented in this thesis, let us consider some of the commercial tools available for use with microcomputers. The four tools summarized here are 1st CLASS FUSION, GURU, CLIPS, and NEXPERT-OBJECT.

1.2.1 1st CLASS FUSION:

Let us start with 1st CLASS FUSION by 1st-Class Expert Systems, Incorporated. This tool represents rules as decision trees. Each knowledge base consists of one such rule which can be built from a set of examples or by hand (see Figure 1.2). This allows each knowledge base to produce only one answer or result. Knowledge bases can be linked together to form a network. This is done through the use of meta-rules. Each decision tree can be entered manually or induced from a set of examples.

The *Adviser* (the system that actually runs applications) uses a question and answer style approach to get needed values. Text can be linked to all *factors* and the values associated with those factors to provide the system with a more user-friendly way with which to acquire information. Prompt screens can be created to get additional variable values by adding special code to the text associated with rule factors. This method can also be used to create reports, send values to be added to those reports, and produce help screens.

Although these facilities are flexible, they are also unnatural. Rather than these types of code statements being stated explicitly within the actual rules or in some other construct, they are hidden within text that is attached to the various factors within the rules.

1st CLASS FUSION provides a limited means of explanation. It is possible to export pictorial forms of decision trees to text files. These text files can be displayed through special calls that can be placed within text attached to various factors. Such explanations are primitive.

Lastly, 1st CLASS FUSION provides a mechanism to export the decision tree of a knowledge base to C or Pascal code. This code can then be modified to enhance the actions the tree performs at each of its leaves. However, if the tree ever needs to be modified within 1st CLASS FUSION and then exported again, all additions made to the original exported source file will be lost.

Factors:

Cred_rating	YearsAtJob	Income	Address	DECISION
excellent	#.#	#.#	suburban	approved
good			urban	denied
poor				

Examples:

	Cred_rating	YearsAtJob	Income	Address	DECISION	Weight
1.	excellent	1.	30000.	suburban	approved	[1.00]
2.	excellent	1.	30000.	urban	approved	[1.00]
3.	execllent	1.	20000.	suburban	denied	[1.00]
4.	good	3.	40000.	suburban	approved	[1.00]
5.	good	3.	20000.	urban	denied	[1.00]
6.	good	2.	30000.	urban	denied	[1.00]
7.	poor	*	*	urban	denied	[1.00]

Rule Created:

Income??

< 25000.00 : ——— denied

≥ 25000.00 : Cred_rating??

excellent : ——— approved

good : YearsAtJob??

< 2.50 : ——— denied

≥ 2.50 : ——— approved

poor : ——— denied

Figure 1.2 : 1st CLASS FUSION Example Knowledge Base

1.2.2 GURU:

Let us now discuss GURU from Micro Data Base Systems Incorporated. According to Firebaugh, GURU supports a wide range of knowledge representation methods and prefabricated object classes [Firebaugh, 1988]. Forward and backward chaining inference engines can be used that are integrated with relational data base management systems, spreadsheets, graphics and other business-related software packages [Firebaugh, 1988]. GURU contains four separate user interface levels for the system developer [Firebaugh, 1988]. See Figure 1.3 for a specification of GURU's rule syntax. Certainty factors in GURU are also supported. A certainty factor is "a number that measures the certainty or confidence one has that a fact or rule is valid" [Waterman,1986]. GURU allows a certainty

factor on the premises of rules and individual actions of rules. This, in turn, allows the values of variables to have certainty factors associated with them as well. Various calculation methods are allowed for the combining certainty factors. [Holsapple, et al., 1986]

GURU has a limited explanation facility. It allows the user of a developed system to view rule traces and their associated text strings, but detailed explanations cannot easily be built. Another problem with GURU is its limitation of eight characters to an object name. This seems to be a petty argument, but to build complex systems, you need complex names that help to self-document what the associated objects actually represent.

Finally, GURU supports a method for calling C programs from within its rules and a method for C programs to call GURU knowledge bases. However, the mechanisms used to do this use a loose coupling of the data being used by the knowledge base and the C programs. Data must be copied to and from a knowledge base and a C program through the use of special built in routines that read and write to GURU's built in data base. Often the purpose of calling procedural code is to allow complex calculations to be performed in an efficient manner. This purpose is defeated if time must be wasted by calling routines to first retrieve and later set particular data elements of a knowledge base's working memory data base files.

```
RULE : <rule-name>
       [PRIORITY : <number>]
       IF :          <condition-1>
        {AND/OR}     <condition-2>
        {AND/OR}     <condition-n>
       THEN:         <action-1>
                     <action-2>
                     <action-n>
       [REASON : <any text>]
       [COMMENTS : <any text>]
```

Figure 1.3 : GURU's Rule Syntax
(GURU also provides a window-oriented rule editor that outputs its rules in this format.)

1.2.3 CLIPS:

CLIPS is an expert system development tool designed specifically to provide high portability, low cost, and easy integration with external systems [Lyndon B. Johnson Space Center, 1989]. It uses a forward chaining mechanism based on the Rete algorithm [Mettrey, 1991]. It is written in C and provides full integration with C programs through special built in functions that load and execute rules and probe and modify CLIPS's database of facts. As well, CLIPS provides a feature that outputs its rules as C code which may be linked with a user's other C code. In this mode however, there is still no direct link (or mechanism to perform such a link) between the database of facts used by a rule base and data structures that may exist elsewhere in such a system. The programmer must copy data into a CLIPS data structure before invoking CLIPS and then copy it back after CLIPS returns.

The rule syntax of CLIPS has a LISP feel to it (see Figure 1.4). The *defrule* construct is used to define a rule. The premise (left-hand side) of the rule is encapsulated with square brackets and individual *patterns* of the premise are enclosed within parentheses. The actions of the right-hand side of a rule are delimited in the same fashion. Facts may be added and deleted from the fact data base through special function calls within the actions of rules. Facts may also be predefined with the *deffacts* construct.

```
(defrule        Rule-Name
        "Optional Documentation String"
        (condition-1)  ;        The left-hand side is composed of
        (condition-2)  ;        zero or more conditions
        (condition-n)  ;        each enclosed in parenthesis.
=>
        (action-1)              ;       The right-hand side is composed of
        (action-2)              ;       zero or more actions.
        (action-n)      )
```

Figure 1.4 : Rule Syntax of CLIPS [Mettrey, 1991]

The ease with which CLIPS rule bases may be embedded within conventional programs is a reason for its recent popularity. The flexibility provided by this method enhances the ability of developers to create complete knowledge-based systems.

1.2.4 NEXPERT-OBJECT:

NEXPERT-OBJECT is a tool available from Neuron Data. It supports rule and object oriented programming (see Figure 1.5 for NEXPERT-OBJECT's rule syntax). As well, it is able to integrate with many third-party database, spreadsheet, and other such packages. Third-party programs may access its knowledge bases and associated working memories through library calls that read and write to various files. NEXPERT-OBJECT knowledge bases may call third generation programs only by spawning a new process [Neuron Data, 1987].

NEXPERT-OBJECT is a very sophisticated tool, but with its sophistication comes high complexity, price, and memory requirements. The high complexity of NEXPERT-OBJECT makes it a difficult tool to use. Its price is significant enough to scare away many would-be developers. Added to this is its requirement for a run-time version to be purchased with every distributed copy of applications developed in NEXPERT-OBJECT.

Explanations in NEXPERT-OBJECT include the ability to show the current rule, any text that was defined within that rule, and the ability to browse through the hypothesis links of rules. Such explanations, as mentioned earlier, exist only at the knowledge representation level, not the knowledge level.

```
            <rule-name>
if          <condition-1>
      AND   <condition-2>
      AND   <condition-n>
then        <action-1>
      AND   <action-2>
      AND   <action-n>
```

Figure 1.5 : Syntax of NEXPERT-OBJECT Rules
(NEXPERT-OBJECT provides a form-like screen for entering in rules.)

1.3 EX-C Approach and its Advantages:

The main purpose of EX-C is to allow rule bases to be defined and executed within the context of the C programming language. These rule bases can be called from within the application to handle complex tasks. Furthermore, rule bases can be created that exist first as stand-alone applications that can later be combined with conventional code and possibly other rule bases to form more extensive and complete applications. The environment allows tasks suited to rule-based programming techniques to be implemented as rule bases. Conversely, tasks suited to conventional programming techniques can be implemented conventionally. Figure 1.6 shows the three programming levels that are possible within EX-C.

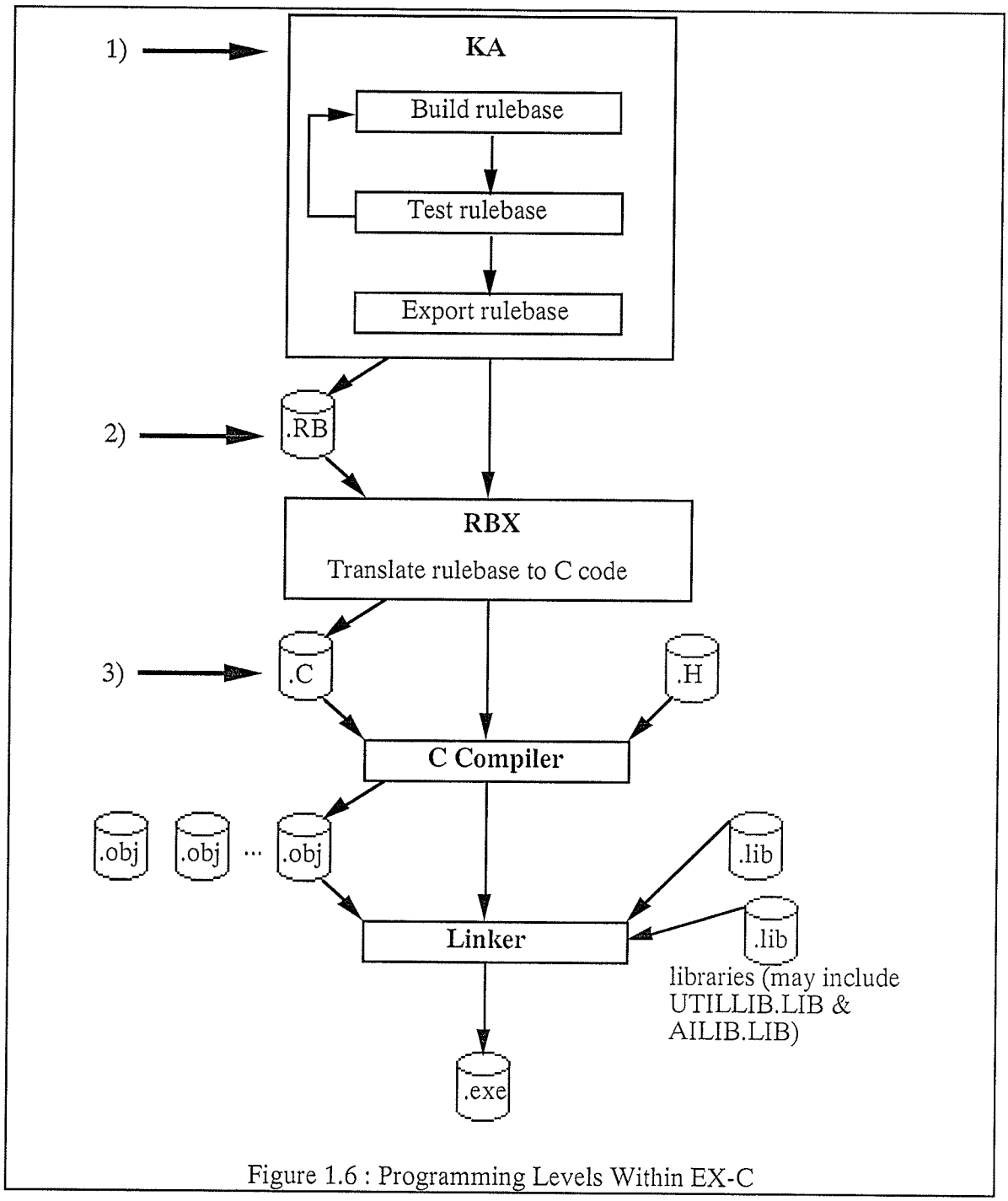

Figure 1.6 : Programming Levels Within EX-C

Another major consideration of EX-C is to allow custom explanations to be built based on rule base results, the data used, and the context of the explanations. The desire for a low cost tool was also a consideration.

The rules within an EX-C rule base are tightly coupled to C data structures. This is performed by allowing C expressions and statements to comprise the bulk of the premise clauses and actions within the rules themselves. Thus, parameters to a rule base can be examined and modified directly by the rules of a rule base. This allows the overhead of data transfer between a rule base and conventional code to be no more than that of a procedure call.

A shell program called KA supports the environment by providing an easy to use rule editor and automated facilities for quickly building rule bases that can later be embedded within applications. This shell allows the programmer to write rules that reference working memory variables that are independent of the application the rule base is later to be integrated with. When the time comes for this integration to occur, the working memory variables may be mapped to application data structures or routines that get and set data items not easily accessible to the rule base or the code that calls it.

EX-C is designed to provide a slightly different approach to integrated conventional and rule-based programming. The tight coupling of data between the rule base and the conventional code allows rule bases to execute quickly with little start up and shut down overhead. The ease with which rule bases may be embedded allows EX-C to become a part of the programmer's tool set. Consequently, tasks amenable to rule-based programming techniques may be represented as such in a conventional programming environment with little extra effort.

Chapter Two discusses the foundation of EX-C. This foundation consists of the internal representation of rule bases in the C programming language. Chapter Three details the various tools that are included within EX-C and its rule base language extensions to C.

Chapter Four discusses three separate examples of rule base development with EX-C. Each example highlights some of the facilities provided. Finally, Chapter Five outlines the major conclusions of this thesis.

Chapter Two: Rule Bases in C

The backbone of all rule base execution tools is the inference engine they use. Some tools, like EX-C, provide multiple inference engines. The inference engines used by EX-C are called rule base processors because they process a set of C structures and routines that define a rule base and return a set of results that describes that processing. There are currently two such rule base processors defined in EX-C, a forward chainer and a backward chainer. The purpose of this chapter is to present the input to and the output from these two processors, thus detailing how rule bases are internally represented in EX-C.

2.1 Rule Base Definition:

A rule base is a complex object. This implies that any formal method of describing such an object will most likely be complex as well. This is the case with rule bases in EX-C. Therefore, let us now discuss how a rule base is defined to the rule base processors in EX-C.

A rule base usually requires data to both reference and modify. This data is provided to the rule base via an object called a working memory structure. This structure is programmer-defined in that the programmer decides what data the rule base should operate on. The working memory structure in EX-C can be any valid C structure but is usually defined specifically for the particular rule base. There is one restriction to the definition of the working memory structure. It must contain a field called *rb_results* which is a pointer to a pre-defined structure of type struct RB_RESULTS. The reasons for this will be detailed later.

There are three C structures used to define rule bases to be used by the rule base processors. The first of these structures is the RB_RULE structure. An array of these is needed, where each element corresponds to one rule in the rule base. The second structure used is called RB_CLAUSE. It has three fields and an array of these is needed for each rule. One RB_CLAUSE array represents the clauses in the premise of one rule. The third and last structure used to define rule bases in EX-C is the RB_GOAL structure. As its name implies, this structure is used to define goals and sub-goals within rule bases. One array of these is needed for an entire rule base, where each element represents a goal or sub-goal that exists in that rule base. Goals and sub-goals are only needed if the backward chainer is to be used.

Let us first discuss the RB_RULE structure. Each RB_RULE structure is used to define only one rule in a rule base. It consists of six fields (see Figure 2.1). The first field, *name*, is a pointer to a character string. This gives a name to the rule. The second field, *n_clauses*, is an integer which provides the number of clauses in the premise of the rule. The third field, *short_circuit*, is used by the processor when evaluating the clauses of the rule's premise to determine if the rule should or should not be fired. It may be that the processor can determine this before it evaluates all of the clauses. This field allows the programmer to force the processor to always test all premise clauses of a rule. The default value for *short_circuit* is true. In other words this field tells the processor whether or not it should use short circuit *and*'s and *or*'s. The fourth field, *clauses*, is a pointer to an array of RB_CLAUSE structures. This array represents the premise of the rule. The fifth field, *actions*, is a pointer to a function that returns void (i.e. a procedure). This procedure represents the action part of the rule or the code to execute if the rule is fired. The sixth and last field in the RB_RULE structure, called *reason*, is another pointer to a character string. This string should contain a high level explanation of the conditions necessary for a rule to execute and what the action of that rule performs.

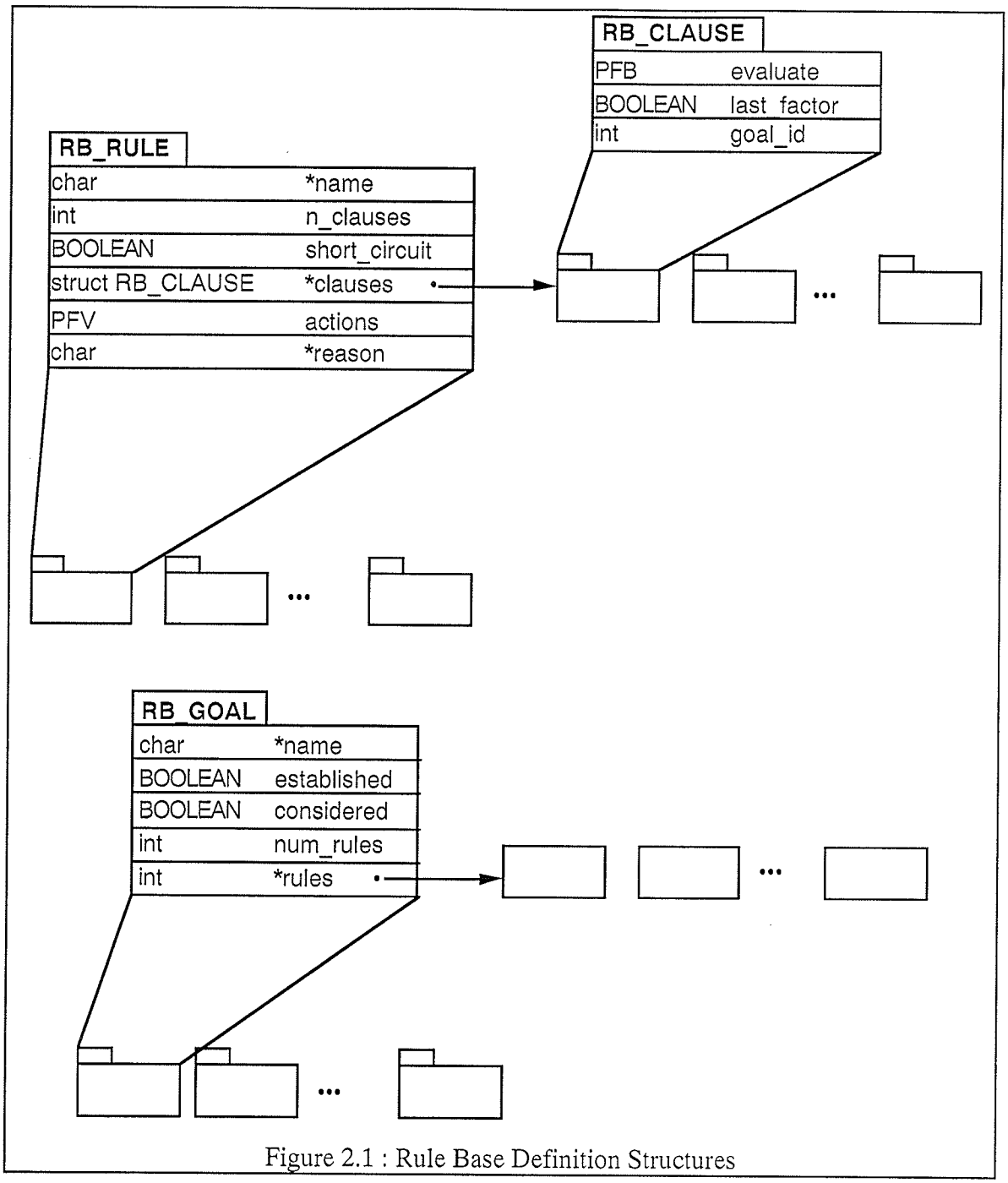

Figure 2.1 : Rule Base Definition Structures

Let us now discuss the RB_CLAUSE structure. This structure, as mentioned above, represents one clause in the premise of a rule. There are three fields in this structure which are also pictured in Figure 2.1. The first field, called *evaluate*, is a pointer to a routine that returns a boolean result. This routine contains C code that when executed tests a condition

representing the (user-defined) clause. The rule base processors use these boolean results to determine if the rule should be fired or not. The second field, called *last_factor*, is a boolean field. It is used by the rule base processors to determine how the clauses in the premise should be *and*'ed and *or*'ed together. This is discussed in section two of this chapter. The third and final field in the RB_CLAUSE structure is an integer called *goal_id*. It denotes whether a sub-goal is being tested in this clause. This field either contains the value *NULL_GOAL* or the index of a valid goal in the rule base.

The last structure that is required to define a rule base in EX-C is the RB_GOAL structure (see Figure 2.1). An array of these structures is actually needed, where each element represents a valid goal or sub-goal in the rule base. There are five fields in the RB_GOAL structure. They consist of a pointer to a character string, the *name* of the goal, a boolean field *established* that indicates whether or not the goal has already been established, a boolean field *considered* that is set to true if the goal has been considered at least once by the rule base processor, a pointer to an array of integers, *rules*, that contains the rule indexes of all rules that conclude the goal, and an integer field *num_rules* that gives the size of the *rules* array.

In the discussion of the structures RB_RULE and RB_CLAUSE it was mentioned that there were subroutines that were pointed to by these structures. These subroutines obviously must also be defined. Each clause in the premise of each rule should have a corresponding function that returns a boolean result. As well, there should be one procedure for the action part of each rule in the rule base. These routines all should accept as their only parameter, a pointer to the working memory structure that is defined for the rule base. Example 2.1 shows an example of a rule and the corresponding subroutines needed to define that rule in C.

```
RULE Test
IF      A == 0
 AND    B >= 2
THEN    C = 2;
        D = E;

static BOOLEAN rb0_r0_c0(wm)
  struct WM *wm;
{return(wm->A == 0);}

static BOOLEAN rb0_r0_c1(wm)
  struct WM *wm;
{return(wm->B >= 2);}

static void rb0_r0_a(wm)
  struct WM * wm;
{
  wm->C = 2;
  wm->D = wm->E;
}
```

Example 2.1 : Sample Rule and its Corresponding Subroutines

The following, Example 2.2, shows a driver routine for a particular rule base. Notice that the RB_CLAUSE arrays, RB_RULE array, and RB_GOAL array are statically defined within this routine.

```
static void backward_test()
{
#define SHOWERED 0
#define FULL 1
#define DRESSED 2
#define READY 3
#define UNDRESSED 4

static int goal_concluder0[1] = {4};
                .
                .
                .
static int goal_concluder4[1] = {3};
static struct RB_GOAL goals[] = {
        {"SHOWERED",        FALSE,FALSE,1,goal_concluder0},
                                        .
                                        .
                                        .
        {"UNDRESSED",       FALSE,FALSE,1,goal_concluder4},
        };

static struct RB_CLAUSE r0[] = {{rb1_r0_c0, TRUE,  NULL_GOAL}};
                                        .
                                        .
                                        .
static struct RB_CLAUSE r5[] = {{rb1_r5_c0, TRUE,  FULL}};
static struct RB_RULE rules[] = {
       {"setup",1,TRUE, r0, rb1_r0_a, "Set up variables ..."},
       {"ready_for_day",3,TRUE, r1, rb1_r1_a, "If the person has ..."},
       {"get_dressed",2,TRUE, r2, rb1_r2_a, "If undressed, then ..."},
       {"prepare_for_shower",1,TRUE, r3, rb1_r3_a, "If clothed, ..."},
       {"shower",2,TRUE, r4, rb1_r4_a, "If ready for shower, ..."},
      {"eat",1,TRUE, r5, rb1_r5_a, "If hungry then eat  breakfast."}};
static struct RB_RESULTS *res;
static struct WM wm = {NULL,NULL};
struct WM working_memory;

working_memory = wm;
res = rb_setup_results(RB_BACKWARD, 5, goals, &working_memory);
if (res != NULL) {
  working_memory->rb_results = res;
  res = rb_process(6,rules,READY);
  rb_cleanup(res);
  }
}
```

Example 2.2: A Rule Base Driver Subroutine

As presented earlier, the definition of a rule base in C is very complex. Thus, to attempt to code such a definition by hand would be cumbersome, tedious, and error-prone. Therefore, C was extended to include a set of keywords and language constructs that allow

a rule base to be defined in a more natural fashion. To support this, a translator called RBX was written. It produces the structures and routines discussed above. RBX and the extensions to C will be discussed in Chapter Three.

2.2 Rule Base Execution:

Now that the definition of a rule base in C has been described, let us now describe how such a definition is used by the two current rule base processors to execute the rules of a rule base. For this discussion, Example 2.2 will be used.

Before a rule base can be executed, it must have a valid set of data with which to work. A rule base gets its data from its working memory structure. Therefore, the rule base's working memory structure must be defined and initialized prior to execution. Remember, fields of a working memory structure may be any valid C data type, which includes pointers. Therefore, a working memory structure may contain addresses of structures that already contain initialized data. In this case, the pointers in the working memory structure must be initialized. Working memory structures may also contain fields that are local to the rule base only. In this case, the values of these fields need to be initialized.

Next, the pre-defined subroutine *rb_setup_results* must be called. (See Figure 2.2 for a description of the three built-in routines that are used to set up, invoke, and clean up a call to a rule base.) This subroutine accepts four parameters, the mode in which the rule base is to be run (constants *RB_FORWARD* and *RB_BACKWARD* are defined for this), the number of goals in the rule base, the address of an initialized goal array of the specified size (this is ignored if the number of goals is zero), and the address of the working memory structure to be used by the rule base. The routine returns the address of an allocated and partially initialized results structure. The address of the results structure must be assigned to the *rb_results* field of the working memory structure. This cannot be performed by

rb_setup_results because it does not know how the fields of the working memory are organized. (It only knows the working memory's location.)

Next, the routine *rb_process* is called. It accepts three parameters. The first, *nrules*, specifies the number of rules present in the rule base. The second, *rules*, is an array of *RB_RULE* structures that correspond to the rule base to be processed. The third, *solve*, indicates the goal to be solved. In the case of the forward chainer, *solve* will be ignored. However, in the case of the backward chainer, *solve* contains the index of the goal the chainer is to pursue.

A stack of results structures, internal to the *rb_proc.c* module (the module that contains all the subroutines for setting up, invoking, and cleaning up rule base executions), is maintained by the *rb_setup_results* and *rb_cleanup* routines. The *rb_process* routine always uses the results structure that is on the top of this stack. It determines whether to use the forward or backward chainer from the *processing_mode* field of this results structure. The reason this stack is maintained is to allow one rule base to call another.

rb_cleanup	:	frees up all storage allocated for the rule base results
rb_process	:	invokes the applicable rule base processor
rb_setup_results	:	allocates a results structure and partially initializes it

Figure 2.2 : Built-in Routines for Invoking Rule Bases

After a rule base has completed executing, the results structure may be used in whatever way the programmer decides as long as it is not modified. When the results structure is no longer needed, the subroutine *rb_cleanup* should be called. This routine accepts one argument, a pointer to the results structure. The *rb_cleanup* subroutine frees up any memory that was dynamically allocated by the rule base processor and the *rb_setup_results* routine (including the results structure itself).

2.2.1 Forward Chainer:

A forward chainer is one type of inference engine used by knowledge-based systems to utilize their knowledge. Forward chaining is often described as event or data directed because events or data are used as starting points rather than defined goals. [Liebowitz, 1988]. Forward chaining has been used in previous expert systems for such tasks as data analysis, design, diagnosis, and concept formation [Liebowitz, 1988].

Let us now discuss the inner workings of the forward chainer in EX-C. It uses a simple conflict resolution strategy. Each rule may only execute once. Rules are considered in the order in which they are defined. When a rule is found to be applicable, it is fired. The chainer then begins at the top of the array of rules once again and repeats the process. The process will halt in two cases: no more rules can be found to be fired or the last rule to be fired executed a call to the *rb_exit* built-in subroutine.

To invoke the chainer, the rule base driver routine calls the subroutine *rb_process*. In this case, the *processing_mode* field of the results structure would have the value *RB_FORWARD*. The *rb_process* routine then initializes the remaining results structure fields and calls the forward processing rule chainer.

The forward chainer starts at the top of the array of rules. It skips any rules that have already been fired (initially none would have previously fired). When it finds a rule that has not previously fired, the forward chainer tests that rule's premise. This is done through the use of the *clauses* array that is a field of the RB_RULE structure of the corresponding rule.

The test of the premise is performed differently if the *short_circuit* flag of the rule is set to true as opposed to if it is set to false. Basically, if the *short_circuit* field is false, all clauses of the premise must be evaluated. If it is true, clauses only need to be evaluated until it can be determined that the rule is definitely applicable or definitely not applicable.

Clauses of the premise are evaluated using the *evaluate* function that is a field of each element in the *clauses* array. This C function returns a true or false value that indicates the value of the conditional expression for the corresponding clause. Clauses are *and*'ed and *or*'ed together by using the *last_factor* field of each clause element. If a clause is a *last_factor*, then it is the last clause in a group of clauses that should be *and*'ed together. The result of *and*'ing the next group of clauses is *or*'ed with the result of *and*'ing the first group of clauses. This is repeated for all clauses.

Once the forward chainer finds a rule whose premise evaluates to true, the results structure is modified and the action of that rule is fired. The chainer then proceeds to the next rule in the array of rules and continues as outlined above. Thus, no conflict set is established when the forward chainer processes the rules; when a rule is found to be applicable, it is fired immediately. However, the design of EX-C allows for other chaining mechanisms to be added. These chainers may use other algorithms to decide which rule in the conflict set should be fired. Figure 2.3 shows a simplified flow chart of this algorithm and Figure 2.4 shows the structure of the subroutines that are called.

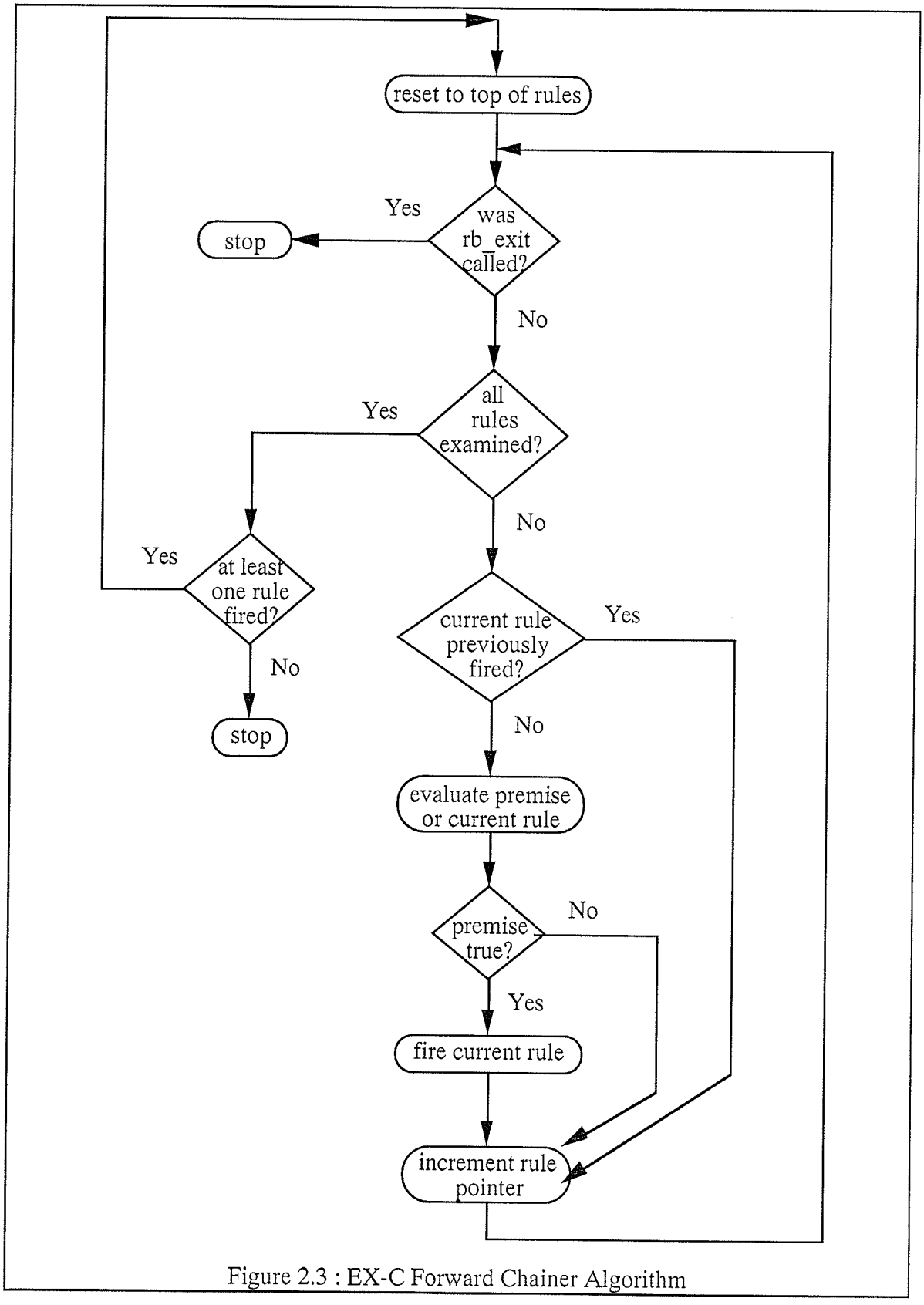

Figure 2.3 : EX-C Forward Chainer Algorithm

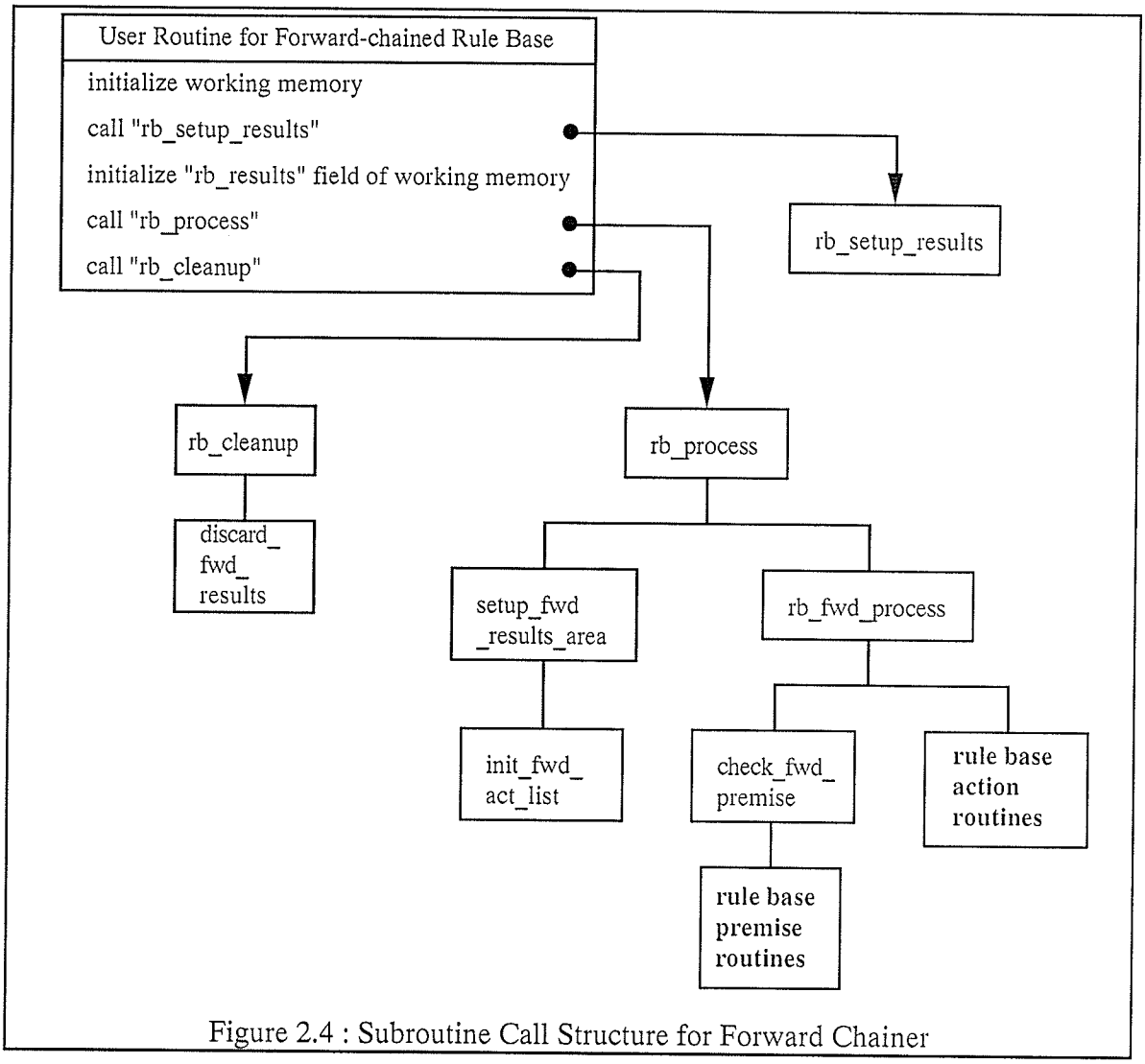

Figure 2.4 : Subroutine Call Structure for Forward Chainer

2.2.2 Backward Chainer:

As one would expect, a backward chainer works quite differently than a forward chaining inference engine. Backward chainers use what is called goal-driven reasoning because they start with a goal as a starting point and work backwards in an attempt to solve that goal [Liebowitz, 1988]. Backward chaining has been used in the past for tasks that involve diagnosis and planning [Liebowitz, 1988].

The backward chainer in EX-C is thus much different from the forward chainer presented above. It uses a recursive approach rather than a looping mechanism to satisfy goals that are specified in the rules of the rule base. The *solve* argument to *rb_process* must be a valid goal index (i.e. an index into the RB_GOAL array). It is used as the index of the goal that the rule base is supposed to satisfy. The *processing_mode* flag of the results structure will hold the value *RB_BACKWARD*. The *rb_process* routine again initializes the remaining fields in the results structure. These fields are intitialized differently, however, than for the forward chainer. The backward processing rule chainer routine, *rb_bwd_process*, is then called by *rb_process*.

The backward chainer always concentrates on the goal to be solved. Thus, it originally calls a routine called *rb_deduce_goal* to solve the specified goal. This routine searches for rules that can be fired to satisfy the goal passed to it. For each goal, an array of rule numbers is stored. These rule numbers are the indexes of the rules that can be used to solve the goal. To satisfy the goal, *rb_deduce_goal* searches this array of rule numbers for a rule that can be fired. If it succeeds, the goal is satisfied.

Again to fire a rule, its premise must evaluate to true. With regards to the *short_circuit* flag, clauses of a premise are handled in the same way as in the forward chainer. However, a clause in the premise of a rule may contain a reference to a particular goal through its *established* field. In this case, the *goal_id* field of the clause structure must be set to the index of that goal. The forward chainer treats this type of premise clause no differently than any other. The backward chainer, however treats this type of clause in a special way. First, it calls the clause's *evaluate* function. If the clause evaluates to true it proceeds normally. If the clause does not evaluate to true, the backward chainer attempts to satisfy the sub-goal if it has not already been satisfied by recursively calling

rb_deduce_goal. It then recalls the clause's *evaluate* routine and uses the value returned as the value of the clause. This is done because other conditions within the clause may cause it to fail.

If the rule that the chainer examines cannot be fired, the next rule in the array of rule indexes (for the current goal to be satisfied) is examined. If one cannot be found then the goal is unsolvable with the current contents of the working memory.

If the backward chainer finds a rule that can be fired, it fires that rule by calling the action routine that is associated with it. This will cause the particular goal (and possibly other goals) to be solved. As well, other actions may also be performed. If the chainer, at that moment, happens to be solving the original goal, its job would then be complete. The chainer would then simply return to *rb_process*. If however, the chainer was attempting to solve another goal (i.e. it was recursively called previously), it would now be able to return back to the point at which the chainer was recursively called and continues from there. Figure 2.5 shows a flow-chart-like picture of the backward chainer's algorithm. Note the recursion that may occur when attempting to evaluate a premise clause. Figure 2.6 shows the structure of the subroutines that are called.

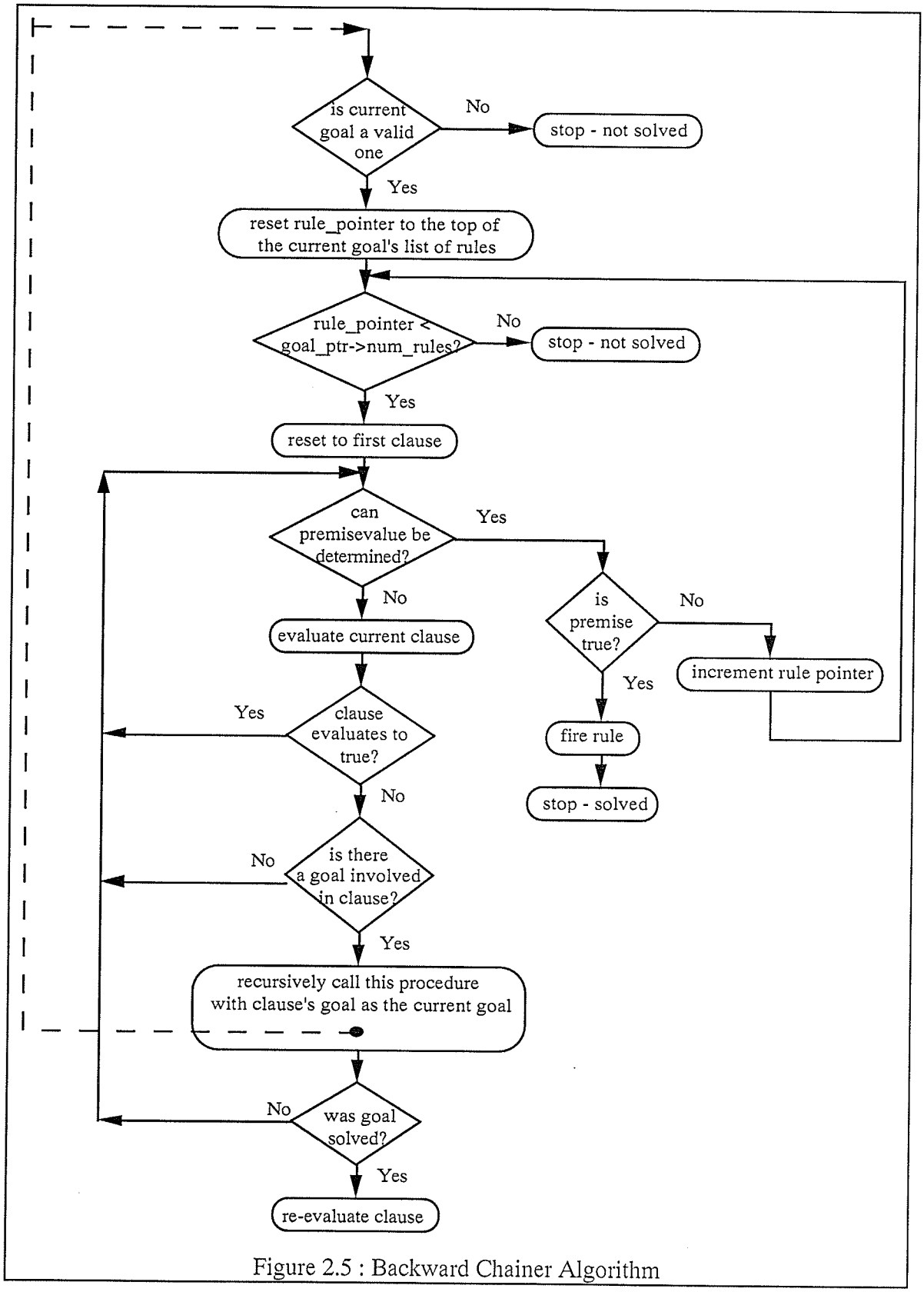

Figure 2.5 : Backward Chainer Algorithm

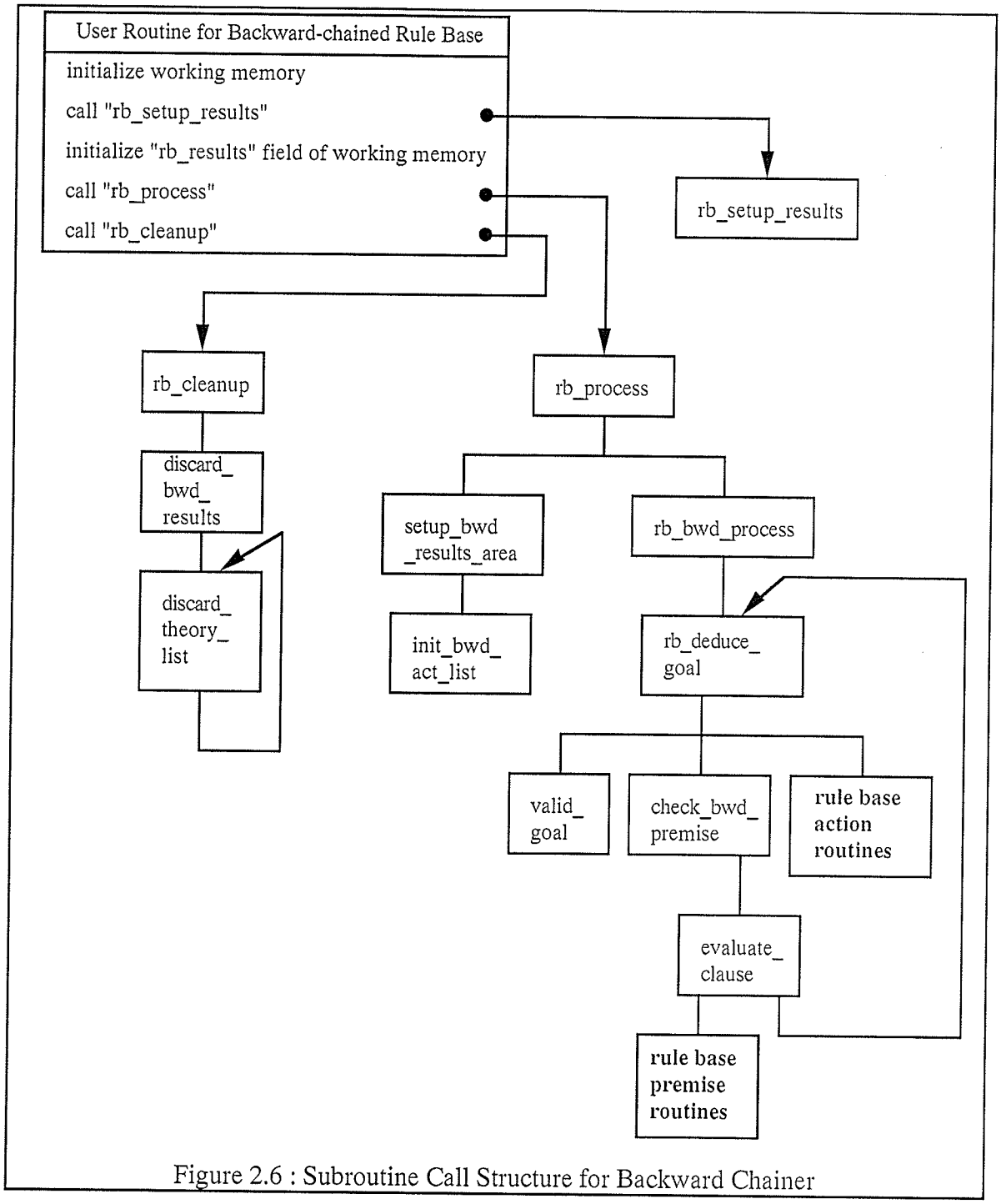

Figure 2.6 : Subroutine Call Structure for Backward Chainer

2.3 Rule Base Results:

The rule base processors construct an extensive structure of results based upon the execution of a rule base. The base of this results structure is constructed before the rule

base actually begins to execute. This base contains various information about the rule base that includes all the structures discussed earlier in this chapter (the RB_GOAL array and the RB_RULE array combined with the RB_CLAUSE arrays). Beyond the base of the results structure is a set of information that describes how the rules in the rule base were examined and executed by the particular rule base processor. Since the forward and backward chainers are vastly different in the way they process a set of rules, the information that describes this processing is also vastly different. Therefore, the results from executing a rule base are structured differently depending upon the chainer that was used. Let us first discuss the information that is common to the results returned by both rule base processors.

2.3.1 Common Results:

Figure 2.7 and Figure 2.9 illustrate the results structures constructed by the forward chainer and the backward chainer respectively. Notice that the top halves of the two diagrams are identical. These parts of both diagrams represent the information that is common to the results structures constructed by both chainers.

Let us begin with the RB_RESULTS structure. The *rb_process* routine returns a pointer to a structure of this type to its caller when the rule base processor has finished. This structure contains ten fields.

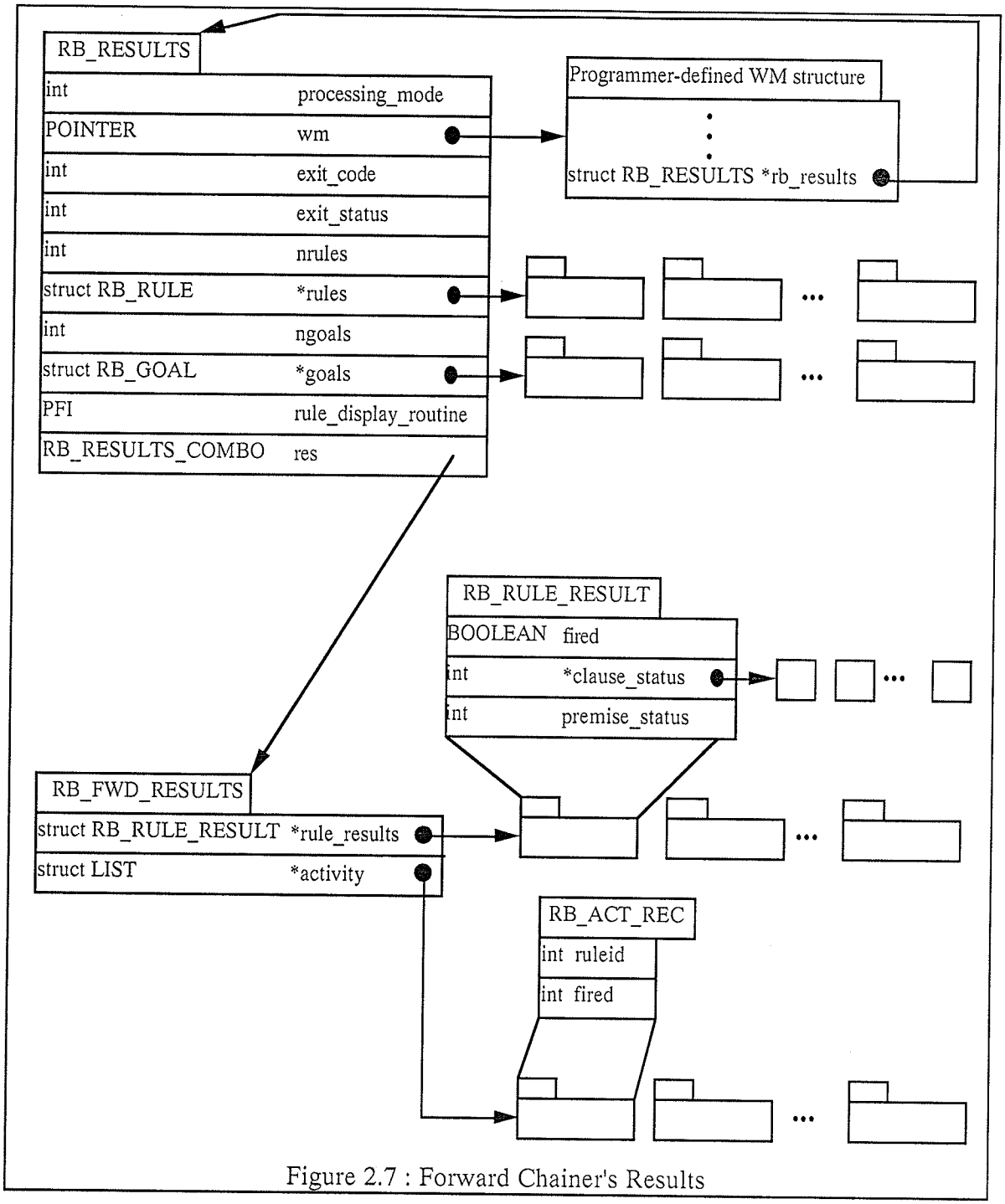

Figure 2.7 : Forward Chainer's Results

The first field, *processing_mode*, indicates which chainer was used to process the rule base. This field is discussed earlier in this chapter.

The second field, *wm*, is a generic C pointer. It will contain the address of the working memory structure that was used when executing the rule base. A more specific pointer type is not associated with this field because the structure of the working memory is not known to the rule base processors. All references to fields in the working memory structure are made by the premise clause and action routines of the rules themselves.

The third field, *exit_code*, is defaulted to zero. It is only changed if the built in routine *rb_exit* is called from within a rule. This routine causes rule base processing to halt before the next rule is examined. A integer value may be passed to *rb_exit*. The *exit_code* of the results structure is set to that value if *rb_exit* is called.

The fourth field, *exit_status* can take on four different values, *RB_EXIT_SET*, *RB_EXIT_ALL*, *RB_EXIT_NOTALL*, and *RB_UNKNOWN* (see Figure 2.8 for a description of these values). *RB_EXIT_SET* indicates that rule base processing was halted by a call to the *rb_exit* routine. *RB_EXIT_ALL* applies only to the forward chaining processor. It indicates that all the rules in the rule base were successfully fired. *RB_EXIT_NOTALL* also applies only to the forward chainer. It indicates that not all the rules in the rule base were fired, but no more rules could be found to fire. The *exit_status* field can also contain the value *RB_UNKNOWN* if the backward chainer was used and no call was made to *rb_exit*.

RB_EXIT_ALL	: the rule base halted execution because all rules were fired
RB_EXIT_NOTALL	: the rule base halted execution because a pass of all rules that had not fired found none that were eligible to be fired
RB_EXIT_SET	: the rule base halted execution because the last rule that fired called the *rb_exit* built-in subroutine
RB_UNKNOWN	: the initial value of *exit_status*

Figure 2.8 : The Possible Values of *exit_status*

The next two fields, *rules* and *nrules* make up the RB_RULE array and its size. An element of the RB_RULE array is not exploded in the diagrams because of space limitations on the paper and because its structure is shown in Figure 2.1. This is the actual array of rules that is used by the rule base processors.

The next two fields, *goals* and *ngoals* make up the RB_GOAL array and its size. Like the RB_RULE array, an element of the RB_GOAL array in these diagrams is also not exploded because it too is shown in Figure 2.1. This too is the actual array that is used by the chainers.

The ninth field, *rule_display_routine* is related to some of the support routines that are provided with the rule base processors. Specifically, it can be used in special circumstances by the rule base activity display routines to show the contents of a rule. This field and the routines related to it will be discussed in Chapter Three.

The last field in the RB_RESULTS structure, is the field that contains different information depending upon which rule base processor is used. Let us now discuss the information that is constructed by the forward chaining processor.

2.3.2 Forward Chainer Results:

The data built into the results structure specifically by the forward chainer have two sections. The first section, *rule_results*, contains information about the status of each rule when the rule base has completed executing. This information includes whether or not the rule was ever fired, the status of each clause in the premise (*RB_UNKNOWN, TRUE, FALSE*), and the status of the premise (*RB_UNKNOWN, TRUE, FALSE*). The status of the premise and clauses can be unknown if they were never evaluated by the forward chainer.

The second section of information pertains to the activity of the forward chainer. It will be referred to as the *activity list.* Its data are kept in a generic linked list data structure, but the structure of the data is that of RB_ACT_REC. Each time the forward chainer examines a rule, it inserts a node into the activity list. The two fields for RB_ACT_REC are *ruleid* and a *fired* flag that indicates whether or not the chainer was successful, at that point in time, in firing the rule.

2.3.3 Backward Chainer Results:

The backward chainer provides an activity list and a field that indicates whether or not it was successful in satisfying the goal. Its activity list is much more extensive than the forward chainer's because the backward chainer executes rules in a more complicated way than the forward chainer does.

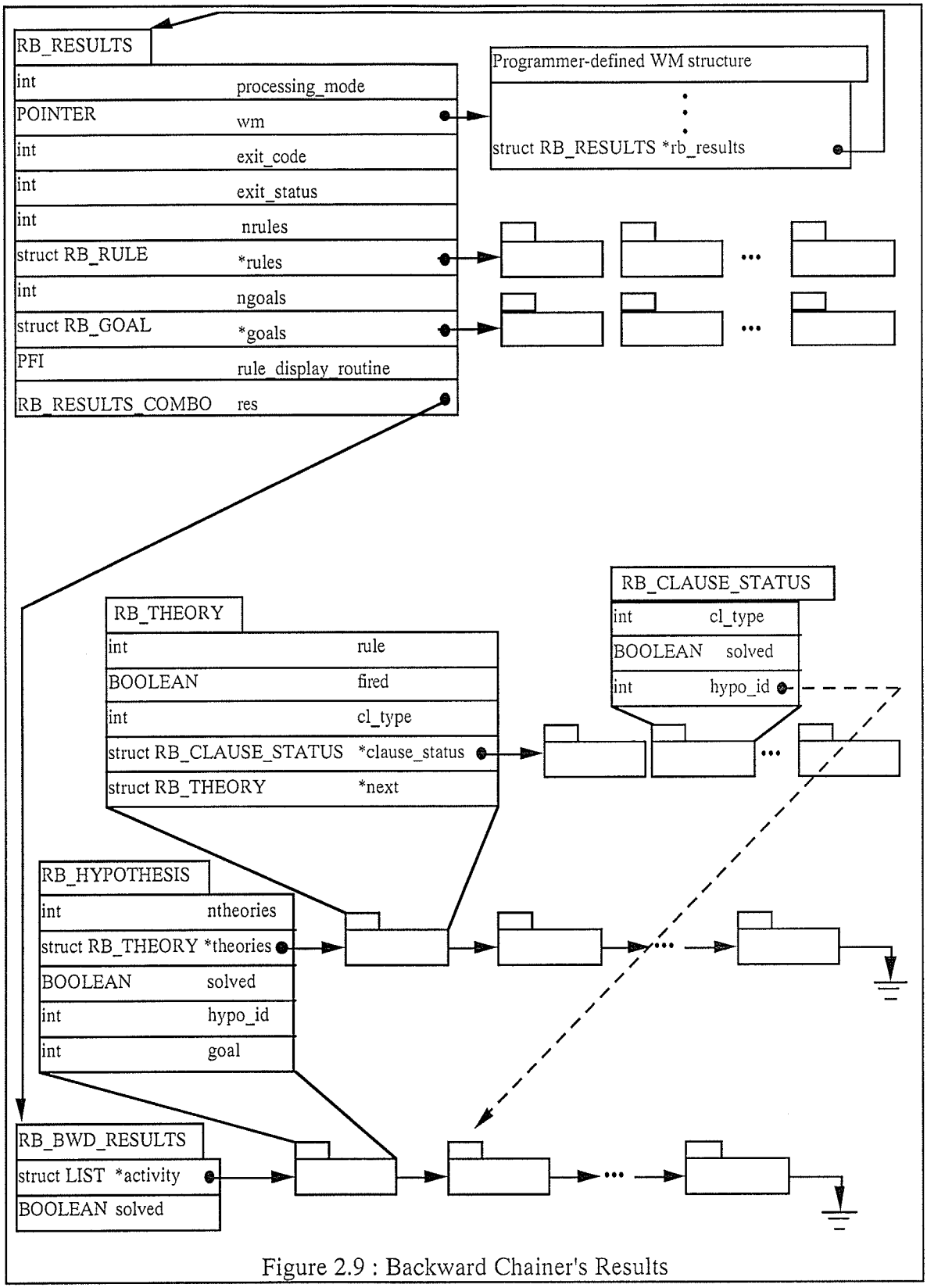

Figure 2.9 : Backward Chainer's Results

The highest level of the activity list is a linked list of RB_HYPOTHESIS's. These are simply the goals that the backward chainer attempted to solve starting with the original goal that was passed to the chainer (let us call this the *main goal*). For each goal that the chainer attempts to solve, a list of theories is stored that basically represents which rules were examined that might possibly solve that goal. Lastly, for each theory (rule) that the chainer examines, an array of status structures is allocated and initialized. Each element of these arrays corresponds to one clause in the premise of the examined rule. The *hypo_id* field of the RB_CLAUSE_STATUS structure is used when a clause is a sub-goal clause that needs to be recursively solved. Thus, the recursive nature of the backward chainer is reflected in the structure of the results it returns.

The above is the representation EX-C uses to define and execute rule bases in C. However, such a representation is tedious to code by hand. A higher level facility is needed to augment this representation to make rule bases simpler to define and thus more efficiently utilized. The approach taken in EX-C is to provide a set of tools that perform this augmentation. These tools are discussed in the next chapter.

Chapter Three: EX-C Tools

The foundation of EX-C consists of the ability to represent a rule base-like object in C. This representation is presented in Chapter Two. However, having the representation is not enough (e.g. having a description of how to represent C source code in object code format is not enough). This chapter presents the additional tools in EX-C that make it a complete extension for rule base programming within the context of a C programming environment. These tools include support routines for on-screen viewing of a results structure, a translator (RBX) that translates an extended version of C (with rule base language constructs) into C, and an integrated rule base shell called KA that ties the whole package together in an easy-to-use fashion that automates most of the related tasks required to develop rule bases. Figure 3.1 shows the different levels (and the required tools) at which users of EX-C can program.

3.1 Support Routines:

There are two support routines that can be used by the programmer to help determine whether a rule base is executing correctly. These routines are called *rb_dump_result* and *rb_display_activity*. The *rb_dump_results* routine is only applicable to results that have come from the forward chainer. The *rb_display_activity* routine has two modes, one for forward chaining results and one for backward chaining results.

The output produced by *rb_dump_results* is a dump of the *rule_results* array that exists in the forward chaining results structure. This output can be used to determine the status of the rules and their premises when the forward chainer halted execution.

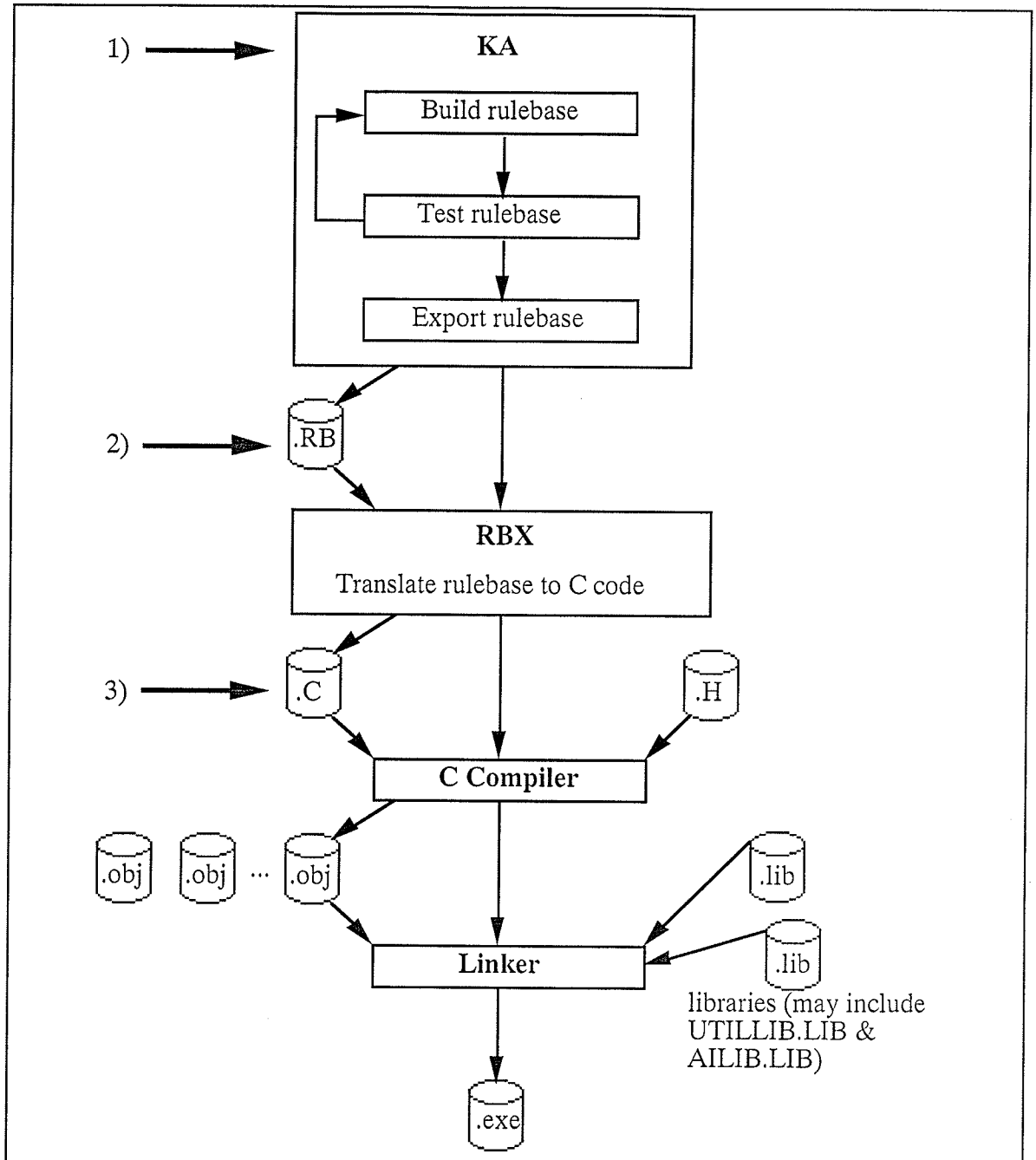

Figure 3.1 : EX-C Development and the Levels at Which it Can Occur

Rb_display_activity produces a different set of output depending on the type of results structure passed to it. For forward chaining results, it can display all rules that were examined by the processor, it can display only the rules that fired, or it can prompt the user

for one of the above. In either mode, the forward chaining activity list display shows the index of the rule that was examined, its name, and whether it was fired or not. Figure 3.2 shows an example of the forward chainer's activity list display. The user can select any row by pressing the enter key. This causes a box to be displayed that contains the *BECAUSE* text from the associated rule.

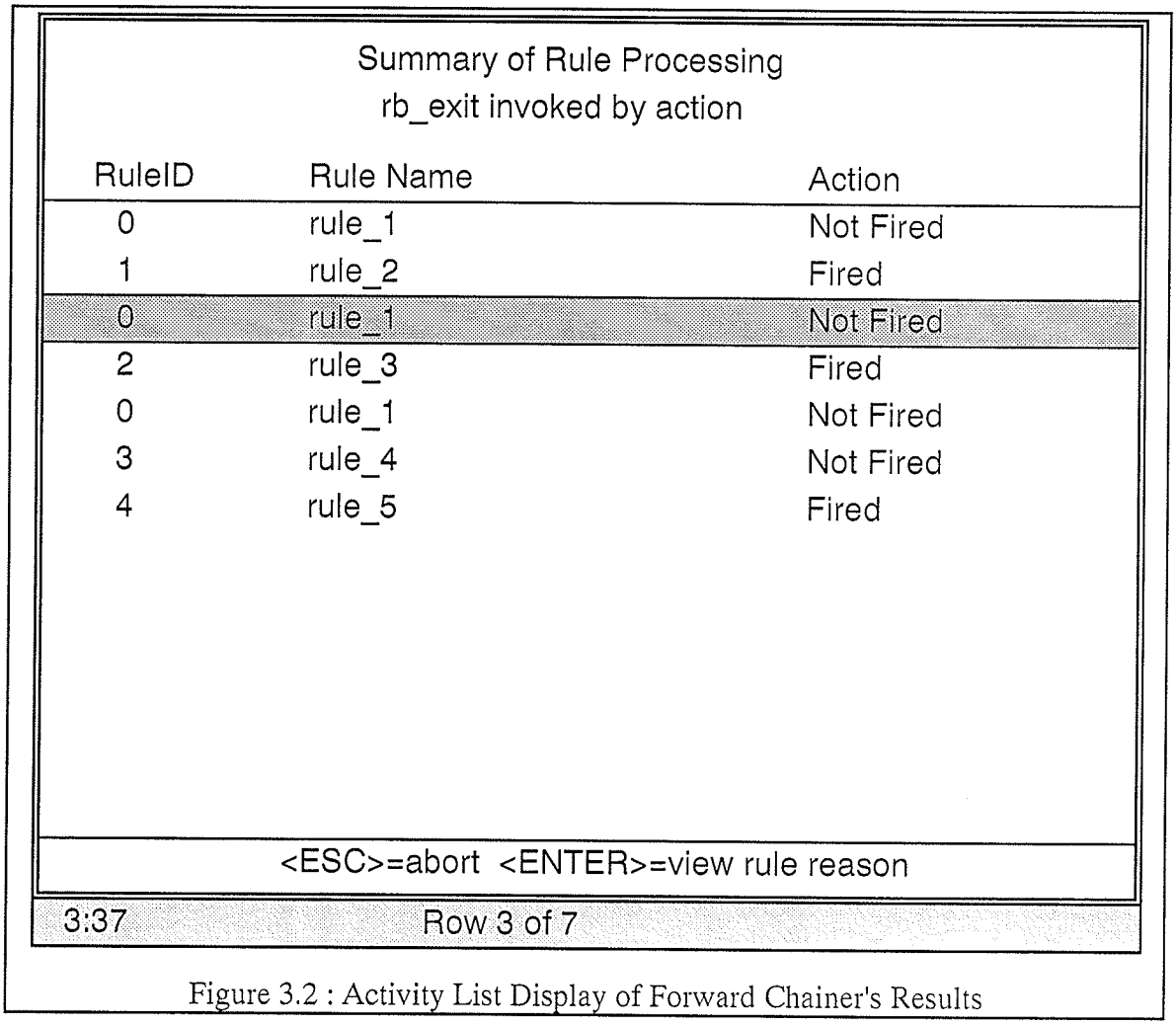

Figure 3.2 : Activity List Display of Forward Chainer's Results

The backward chaining activity display, as one would expect, is quite different than the display for the forward chaining results. This stems from the fact that the two chainers process rules in a different fashion. The activity list display for backward chaining results shows which rules were processed while attempting to solve the goal. For each rule, the

method used in determining a value for each clause in the premise is shown. This may involve examining other rules to solve a sub-goal. Figure 3.3 shows an example of the backward chainer's activity list display. It should be noted that the displays available through *rb_display_activity* are generic to all rule bases. However, since the results used to generate these displays are available to the programmer's own code, customized displays can be constructed if needed.

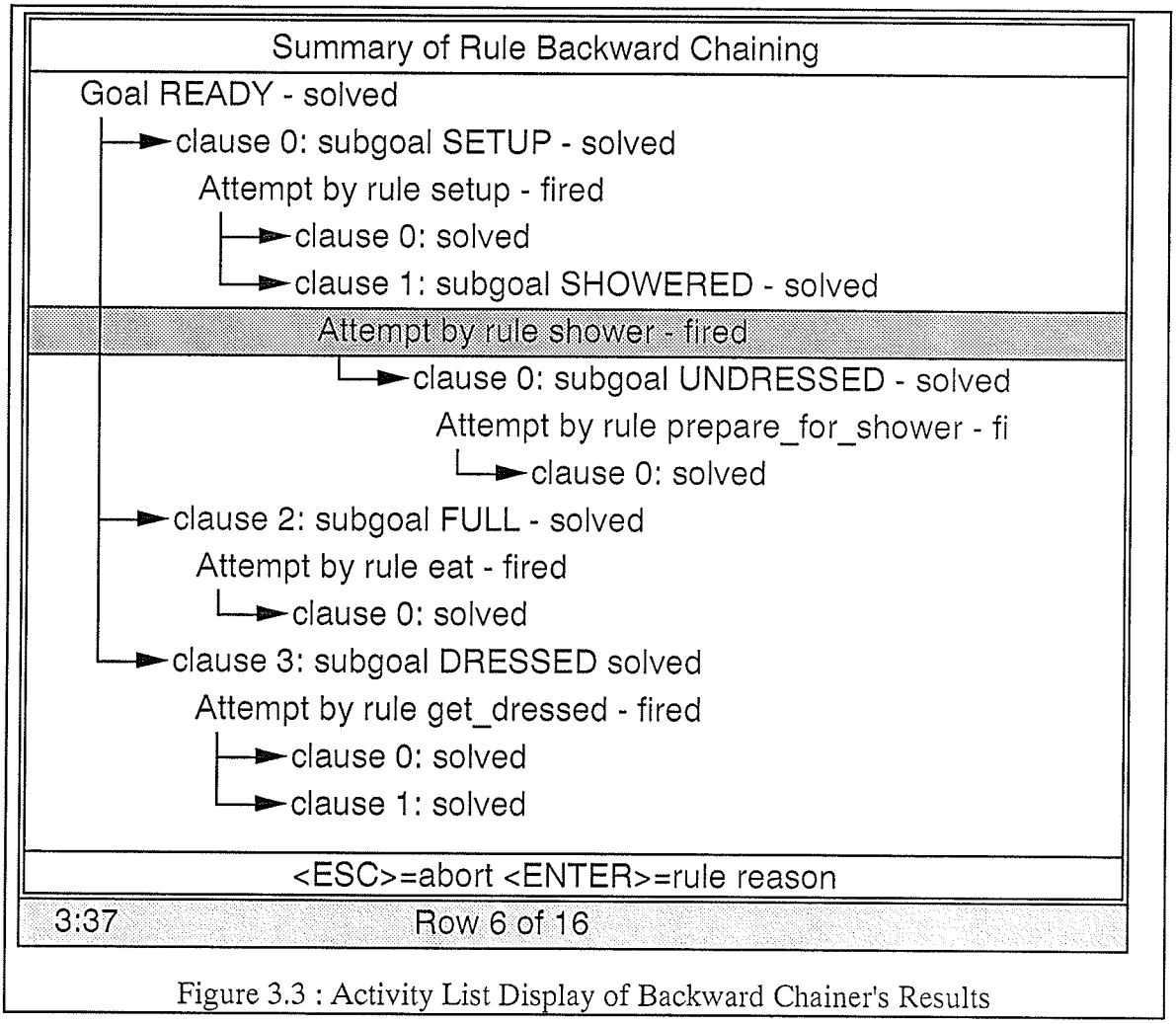

Figure 3.3 : Activity List Display of Backward Chainer's Results

3.2 Rule Base Language Extension of C:

To represent a rule base in C, as presented in Chapter Two, is not an easy task. It is complex enough to warrant a higher-level method of representation. Therefore, EX-C

utilizes a set of rule base language constructs that can be intermixed with regular C language constructs. There are three such rule base language constructs that are used within EX-C. They are, the *INCLUDE* statement, the *DECLARE* block, and the *RULEBASE* block. A translator, RBX, has been developed that parses intermixed rule base and C code and produces C code with the rule base code represented as discussed in Chapter Two. This translator and the details of what the rule base code translates to will be discussed later in this chapter.

Let us begin with the *INCLUDE* statement since it is the simplest of the three to translate. See Figure 3.4 for its format. The purpose of this statement is to allow working memory structures to be shared by more than one rule base but to be declared in only one place. (This mirrors the way a C data structure may be declared in one file but shared among many.) The <*wmfilename*> is the base name of a ".WM" file that may contain other *DECLARE* blocks and *INCLUDE* statements. This file must be translated by RBX to produce a ".H" file.

INCLUDE <wmfilename>

where <wmfilename> is the base name of a .WM file that contains other DECLARE blocks and INCLUDE statements.

example:	INCLUDE family
translates to:	#include <family.h>

Figure 3.4 : *INCLUDE* Statement

The *DECLARE* block is used to declare a working memory structure. Every rule base in EX-C requires a working memory structure on which to operate. The format of this code block is shown in Figure 3.5. As one can see from its format, the *DECLARE* block mirrors the C *struct* statement. However, RBX, when translating a rule base needs to know the names of the data items in its working memory. Because RBX will have

previously translated the *DECLARE* block for the working memory to be used by a rule base, it will already know the names of the data items in that working memory. Thus it will have the information it needs to properly translate the actual rule base.

```
DECLARE <wmname>
        <c_declarations>
DECLARE_END

 where:
        <wmname> is a unique C structure name
        <c_declarations> is a set of valid C variable declarations

example:        DECLARE TEST_WM
                        struct DATE     date;
                        double          total;
                        int             num_people;
                DECLARE_END

translates to:  struct MEMBER_WM {
                        struct DATE     date;
                        double          total;
                        int             num_people;
                        };
```

Figure 3.5: *DECLARE* Block

The final rule base declaration construct that EX-C uses is the *RULEBASE* block. Figure 3.6 illustrates its format. The *RULEBASE* block is the language construct that is used to describe the rules in a rule base. The *storage class* of a rule base indicates whether the rule base driver routine that is created by RBX should be callable outside the module it is defined in.

The *RULEBASE* block requires a name to be given to the rule base. This name must follow the *RULEBASE* keyword. The translator uses this name as a basis for some of the subroutines it must create. Also, a rule base must either use a previously defined working memory structure or define its own. The *USING* clause of the *RULEBASE* block is used for this purpose.

The language extensions allow a table whose rows can be any valid C data type to be associated with a rule base. The *table constructor* clause of the *RULEBASE* block is used to define the data type of the rows and to indicate that a table will be associated with the rule base. Each row of the table is assigned to one rule in the rule base. The purpose of this table is to allow each rule and its corresponding results to be linked to other data structures in an application. This could allow for the creation of better explanations based on the results of a rule base execution and the corresponding application data structures. An alternative way to accomplish the same result is to build required explanations within the actions of the rules themselves.

A rule base may contain any number of rules (except for zero). The *rules* section of the *RULEBASE* block is used for defining rules within a rule base. The premise of a rule may contain valid C expressions and *sub-goal* commands separated by *AND* and *OR*. These expressions usually contain references to data elements within the rule base's working memory. However, references to globally-defined variables are also allowed. A rule may have any number of clauses in its premise. Actions within a rule come after the *THEN* token that follows the rule's premise. A rule may have zero or more actions that consist of valid C statements. Like the expressions in the premise of rules, action statements usually reference data elements within the working memory, but they may also reference global variables. Each rule may have a string constant that describes what it represents. This string constant is located in the *BECAUSE* clause of a rule descriptor. Finally, if a table is defined for the rule base (using the *table constructor*), an entry may be specified by the rule using the *TABLE_ENTRY* clause of the rule descriptor.

```
[<storage class>] RULEBASE <rbname>
        USING {<wmname> | <declare block>}
        [<table constructor>]
        <rules>
        RB_END
where:
    <storage class>          : EXTERNAL | LOCAL
    <rbname>                 : valid C language identifier
    <wmname>                 : name of a working memory structure
    <declare block>          : (defined in Figure 3.5)
    <table constructor>      : [<storage class>]
                               RB_TABLE <table name>
                               USING (<type spec>)
                               [DEFAULT_ENTRY (<entry>)]
          where:  <storage class>  : (defined above)
                  <table name>     : valid C language identifier
                  <type spec>      : a valid C type specification
                  <entry>          : a valid C static data initializer for the given
                                     type specification

    <rules> : One or more rules of the following format:

    RULE <rulename> [SHORT_CIRCUIT | NO_SHORT_CIRCUIT]
    IF      <premise>
    THEN    <actions>
    [BECAUSE <str const>]
    [TABLE_ENTRY (<entry>)]
    where:      <rulename>    : a name to reference the rule by
                <premise>     :<clauses>
                <clauses>     :<clause> | <clause> {AND | OR} <clauses>
                <clause>      : valid C conditional expression which
                                may include at most one use of
                                <subgoal>.
                <subgoal>     : SUBGOAL(<goalname>);
                <goalname>    : a valid C identifier
                <actions>     : zero or more C language statements
                                or <goal cmd>'s
                <goal cmd>    : GOAL(<goalname>);
                <str const>   : a valid C string constant
                <entry>       : (defined above)

example: see Example 3.1 for an example and its translation
```

Figure 3.6 : *RULEBASE* Block

3.3 Automated Translation to C (RBX):

RBX is a batch program written in C that translates combined C and rule base code to strictly C code. RBX can process files with two extensions, ".RB" or ".WM". ".RB"

files are translated to ".C" files and ".WM" files are translated to ".H" files. Let us call a file with a ".WM" extension a *working memory file*, a file with a ".H" extension a *C header file*, a file with a ".RB" extension a *rule base file*, and a file with a ".C" extension a *C source file*.

Rule Base files may contain *INCLUDE* statements, *DECLARE* blocks, *RULEBASE* blocks, and C code. Working memory files may contain the same information as rule base files except they may not contain *RULEBASE* blocks.

RBX can accept as many command line arguments as DOS will permit. It processes each argument in the order it appears. A command line argument may be the full filename of a rule base file, the full filename of a working memory file, or only the base name of a rule base file or working memory file. If a base filename is only provided, RBX attempts to processes both a rule base file with that base name and a working memory file with that base name. As well, RBX only processes files that actually exist.

3.3.1 Translation of Rule Base (.RB) Files:

Let us first discuss the method RBX uses to process a rule base file. Initially, RBX assumes that it is processing normal C language statements. It simply echoes these to the output file until it encounters either an *INCLUDE* statement, a *DECLARE* block, or a *RULEBASE* block. *INCLUDE* statements are translated as indicated in Figure 3.4. As well, when RBX finds an *INCLUDE* statement, it adds the specified file (which is assumed to be the base name of a working memory file) to a list of files that it may later use to search for required working memory definitions that are referenced by *RULEBASE* blocks but have not been found in the rule base file. *DECLARE* blocks are translated as indicated in Figure 3.4. RBX also remembers the names of all working memory

definitions it encounters and the names of the fields in those working memory definitions. This information is kept in the *working memory list*.

When a rule base block is encountered, RBX initially parses the header of that block (up to the first rule). If the working memory is not defined in the header, it searches the working memory list for the required definition. If the working memory definition does not exist in the list, working memory files found in the the list of working memory files (that have not yet been parsed) are parsed until the required working memory definition is found. If it is not found after the last working memory file has been parsed, an error condition results. Let us assume that RBX now has the definition for the required working memory.

If a table is defined for the rule base, various information about the table including its name, its type, and its default entry is kept. A temporary file, to which the table's static definition and initialization is echoed, is also opened (as rules are processed and echoed to the C source file). RBX then proceeds to translate the rules in the rule base until it finds the *RB_END* token, finds an *INCLUDE*, *DECLARE*, or another *RULEBASE* block, or it reaches the end of the current rule base file.

The translation of each rule is quite complex; so let us start with the premise of a rule. A local (static) function is created for each clause in the premise. It is named "rb<rule base #>_r<rule #>_c<clause #>". This function accepts only one parameter, the address of the working memory structure being used and it returns a boolean result. Each clause function contains only one C statement, a *return* statement that contains the text found in the corresponding clause. The only exceptions being that all variable names that are found to be fields in the working memory structure are preceded with "wm->" and any *sub-goal* statements are translated to "wm->rb_results->goals[<goal index>].established" (where <goal index> is the index of the specified goal).

In addition to the routines that are created for the premise of a rule, an array of RB_CLAUSE structures is also created for the premise of each rule. See example 3.1 below to see an actual example of the translation of a rule's premise.

Let us now proceed with how a rule's set of actions is translated. One static procedure is created for the rule's entire set of actions. The rule's action text comprises the body of that procedure again with all variables names that exist in the rule base's working memory being preceded by "wm->".

Finally, an element in an array of RB_CLAUSE structures is created for each rule in the rule base. This structure is used to tie all the pieces of the rule's translation together. For a more detailed description of this structure and others presented here, refer back to Chapter Two.

```
RULE plan_investment_mix_yes SHORT_CIRCUIT
IF      is->class_income[INCLASS_INVESTMENT].gross > 0.0
AND     subgoal(INVESTMENTS_LIQUID)
THEN    agenda->tasks[agenda_retrieve(agenda,PLAN_INV_INC,
                memberid)].applicable = TRUE;
        plan_flags[PLAN_INV_INC] = TRUE;
        buffer1  = txt_print("Planning with the investment allocation mix is applicable"
                " because the individual is receiving investment income (%).",
                txt_cvt_double(is->class_income[INCLASS_INVESTMENT].gross ,
                "$,.2"));
      explain_plan(PLAN_INV_INC , WHY , buffer1);
BECAUSE "Since we have investment income, we may execute the investment mix"
                "individual plan."
TABLE_ENTRY({1,20.0,"plan_investment_mix_yes"})
```

(Assume that this rule is the first rule in the first rule base, that the working memory's name is *MEMBER_WM*, and the *INVESTMENTS_LIQUID* goal has index value 0.)

Translation:

```
static BOOLEAN rb0_r0_c0(wm)
struct MEMBER_WM *wm;
{return(wm->is->class_income[INCLASS_INVESTMENT].gross > 0.0);}

static BOOLEAN rb0_r0_c1(wm)
struct MEMBER_WM *wm;
{return(wm->rb_results->goals[0].established);}

static void rb0_r0_a(wm)
struct MEMBER_WM *wm;
{
wm->agenda->tasks[agenda_retrieve(wm->agenda,PLAN_INV_INC,
        wm->memberid)].applicable=TRUE;
wm->plan_flags[PLAN_INV_INC]=TRUE;
wm->buffer1=txt_print("Planning with the investment allocation mix is applicable"
        " because the individual is receiving investment income (%).",
        txt_cvt_double(wm->is->class_income[INCLASS_INVESTMENT].gross,
        "$,.2"));
explain_plan(PLAN_INV_INC,WHY,wm->buffer1);
}

static struct RB_CLAUSE r0[] = {
        {rb0_r0_c0, FALSE, NULL_GOAL},
        {rb0_r0_c1, TRUE,  INVESTMENTS_LIQUID}};

static struct RB_RULE rules[] = {{"plan_investment_mix_yes",2,TRUE, r0, rb0_r0_a, "Since we have
        investment income, we may execute the investment mix individual plan."},
        .
        .
        .
        };
}
```

Example 3.1 : A Rule and its Translation

There are several objects created by RBX when producing the translated version of a rule base that apply to the entire rule base. One of these objects is the array of RB_RULE structures that has already been discussed. Others include the goal arrays, the rule base table array, the rule base set up routine, and the rule base executor routine.

Let us begin with the goal arrays. For each goal that is concluded by at least one rule, an array of integers called "gconc<goal#>" (where <goal#> is the index number of the particular goal) is defined. These arrays are called *goal concluder arrays*. The elements of each goal concluder array are the rule indexes of rules that conclude the related goal. As well, each goal defined in the rule base has a corresponding element in an array of RB_GOAL's called *goals*. All of these arrays are defined statically within the rule base set up routine (which is described below), so there is no need for them to have names particular to the rule base they are defined for. If a particular goal has a "gconc<goal#>" array defined for it, that goal's RB_GOAL element will have its *rules* field initialized to point to that goal's concluder array. Otherwise, the *rules* field will be initialized to a null pointer. Thus, the concluder arrays may be referenced through the RB_GOAL array. See Example 3.2 for a sample set of goal arrays.

The rule base table array is created if a table is defined in the rule base header using the *RB_TABLE* keyword. This *clause* of the rule base header allows the programmer to specify a C data type that is to be the type of each row in the table. An array of this type will be statically defined and given the name "rb<rule base #>_table". As well, an array pointer of this type is also statically defined and given the name that is provided in the *RB_TABLE* clause. The array pointer is initialized by the rule base set up routine to point to the area where the table resides. This table is defined in this way to allow it to be defined after the actual rule routines, yet still allowing the rule routines to reference the table. This saves RBX from having to make two passes of the rule base itself.

The rule base set up routine is a function that is given the name of the rule base postfixed by "_s". It needs two parameters, the address of the working memory structure being used for the rule base and the type of processing to be performed on the rule base. Its purpose is to set up the rule base for execution. After calling this routine and before the rule base is executed, the value of a goal may be initialized to a value other than its default. The set up routine calls *rb_setup_results* which allocates a results structure, places it on *rb_proc.c*'s internal stack of results structures, initializes some of its fields, and returns its address. The set up routine initializes the working memory structure's *rb_results* field to the address returned by *rb_setup_results*. Another function of this routine is to initialize the rule base table array pointer. True is returned by the rule base set up routine if the results structure was successfully allocated by *rb_setup_results*. Otherwise false is returned.

Finally, the rule base driver routine is defined by RBX and is given the name of the rule base postfixed by "_r". It is a function that accepts one parameter, a goal index, and returns a pointer to a rule base results structure. All RB_CLAUSE arrays and the RB_RULE array are defined statically within this function. The rule base driver function calls *rb_process* with its required parameters and returns the pointer value that *rb_process* returns to it.

```
static struct TABLETYPE rb0_table[]= {
        {0,100.0, "plan_div_yes"},
        {1, 20.0, "plan_investment_mix_yes"},
        {1,  2.0, "hello"},
                .
                .
                .
        {1,  2.0, "hello"}
        };

static BOOLEAN check_member_plans_s(wm, processing_mode)
struct MEMBER_WM *wm;
enum PROCESSING_MODE processing_mode;
{
#define INVESTMENTS_LIQUID 0

static int gconc0[1] = {2};
static struct RB_GOAL goals[] = {
        {"INVESTMENTS_LIQUID",FALSE,FALSE,1,gconc0}
        };

wm->rb_results = rb_setup_results(processing_mode, 1, goals, wm);
table = rb0_table;
return(wm->rb_results != NULL);
}

static struct RB_RESULTS *check_member_plans_r(goal)
int goal;
{
static struct RB_CLAUSE r0[] = {  {rb0_r0_c0, FALSE, NULL_GOAL},
                                  {rb0_r0_c1, TRUE,  NULL_GOAL}};
static struct RB_CLAUSE r1[] = {  {rb0_r1_c0, FALSE, NULL_GOAL},
                                  {rb0_r1_c1, TRUE,  INVESTMENTS_LIQUID}};
                .
                .
                .

static struct RB_RULE rules[] = {
        {"plan_div_yes", 2,TRUE, r0, rb0_r0_a, "Since we have both corporate salary and dividends, we
                map perform the dividend - salary mix member plan."},
        {"plan_investment_mix_yes",2,TRUE, r1, rb0_r1_a, "Since we have investment income, we may
                execute the investment mix individual plan."},
        .
        .
        .       }
        ;
return(rb_process(10,rules,goal));
}
```

Example 3.2 : Example Table Structure and Set Up and Run Routines Produced by RBX

3.3.2 Translation of Working Memory (.WM) Files:

Working memory files are those that have an extension ".WM". They are used to share working memory structures between more than one rule base. At first this seemed like a

feature that would be frequently used. Even though this has not been the case, EX-C supports this anyway. Thus, let us discuss how RBX translates such files.

RBX treats a working memory file in much the same way as a rule base file. The only differences being that rule bases cannot be defined in working memory files and that working memory files are translated into ".H" files instead of ".C" files. Thus, any C code that may normally be found in an include file, *INCLUDE* statements, and *DECLARE* blocks are likely code to be found in working memory files. The generated ".H" file will later be included by various ".C" files and used by the C compiler.

3.4 Integrated Shell (KA):

To this point, EX-C has been presented as a programming language extension to C. Thus, to program rule bases using EX-C, one must know the details of the language extensions and how they work together. This is where the integrated shell, KA, comes in. It is a menu-driven tool for entering and maintaining rule base code that allows rapid prototyping. It provides a rule editor for the user to enter rules with (see Figure 3.7). This editor removes the user from some of the syntactic details of the EX-C rule base language and provides an easy to use mechanism to manage a rule base. The editor also provides a facility to allow explanations to be built based on the failure of the individual clauses of the premise in each rule. Figure 3.8 shows the main activities of the user when using KA.

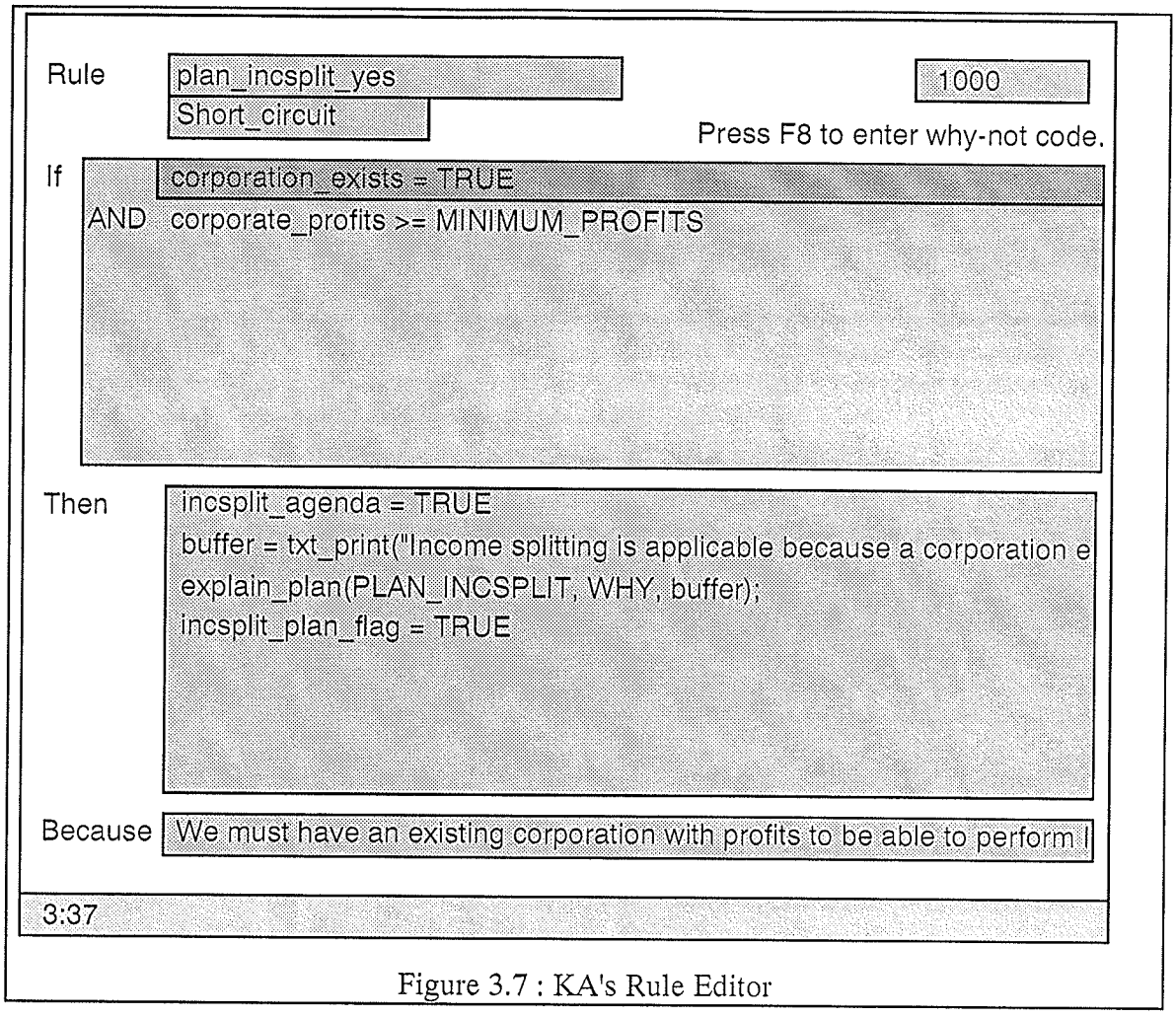

Figure 3.7 : KA's Rule Editor

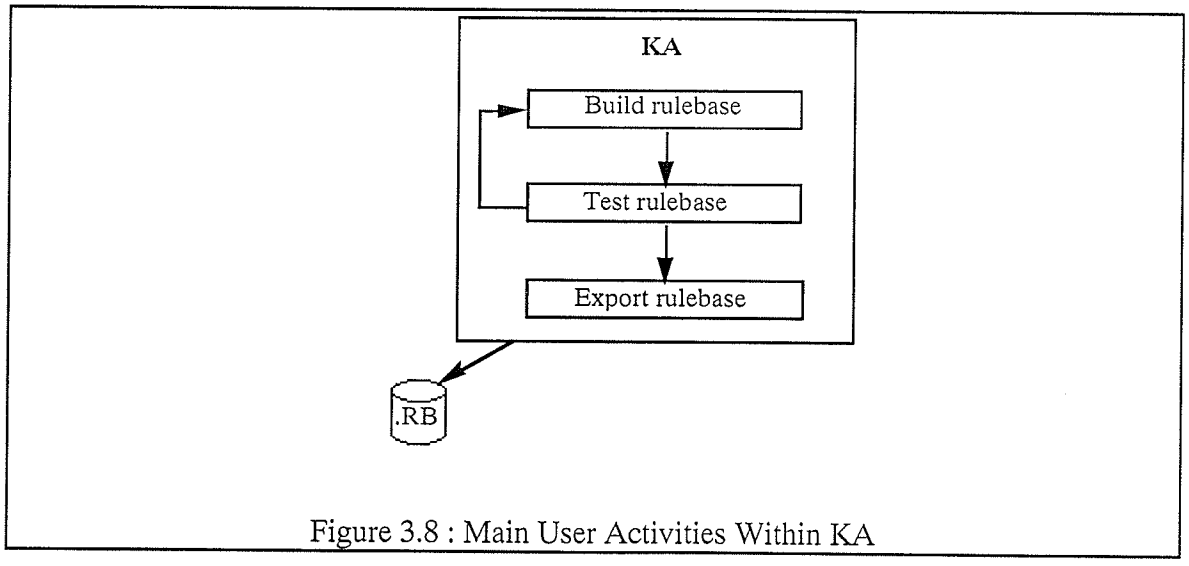

Figure 3.8 : Main User Activities Within KA

There are many other automated facilities in KA that aid the programmer in quickly developing rule bases. These include a section to manage constants that may be used in rules, a facility to automatically build the working memory structure to be used by a rule base, a set of built in data entry routines that prompt the user for input of variables with unknown values during execution of the rule base, a facility that builds an executable file from the current rule base to allow the user to test that rule base, an export facility that allows one to easily map the variable names used within the rule base to actual data elements that exist within an outside application, and a report facility for reproducing a rule base on hard copy.

3.4.1 Constant Manager:

Let us begin with the constant manager. It manages a list of zero or more constants that the user may define to be used within a rule base. There are three pieces of information that can be entered for every constant. These are its name, its value, and a boolean flag that indicates whether this constant is only used by the rule base or whether, upon exporting to an application, the constant will already be defined. This flag is called the *local* flag and it defaults to true. See Figure 3.9 for a sample constant manager screen.

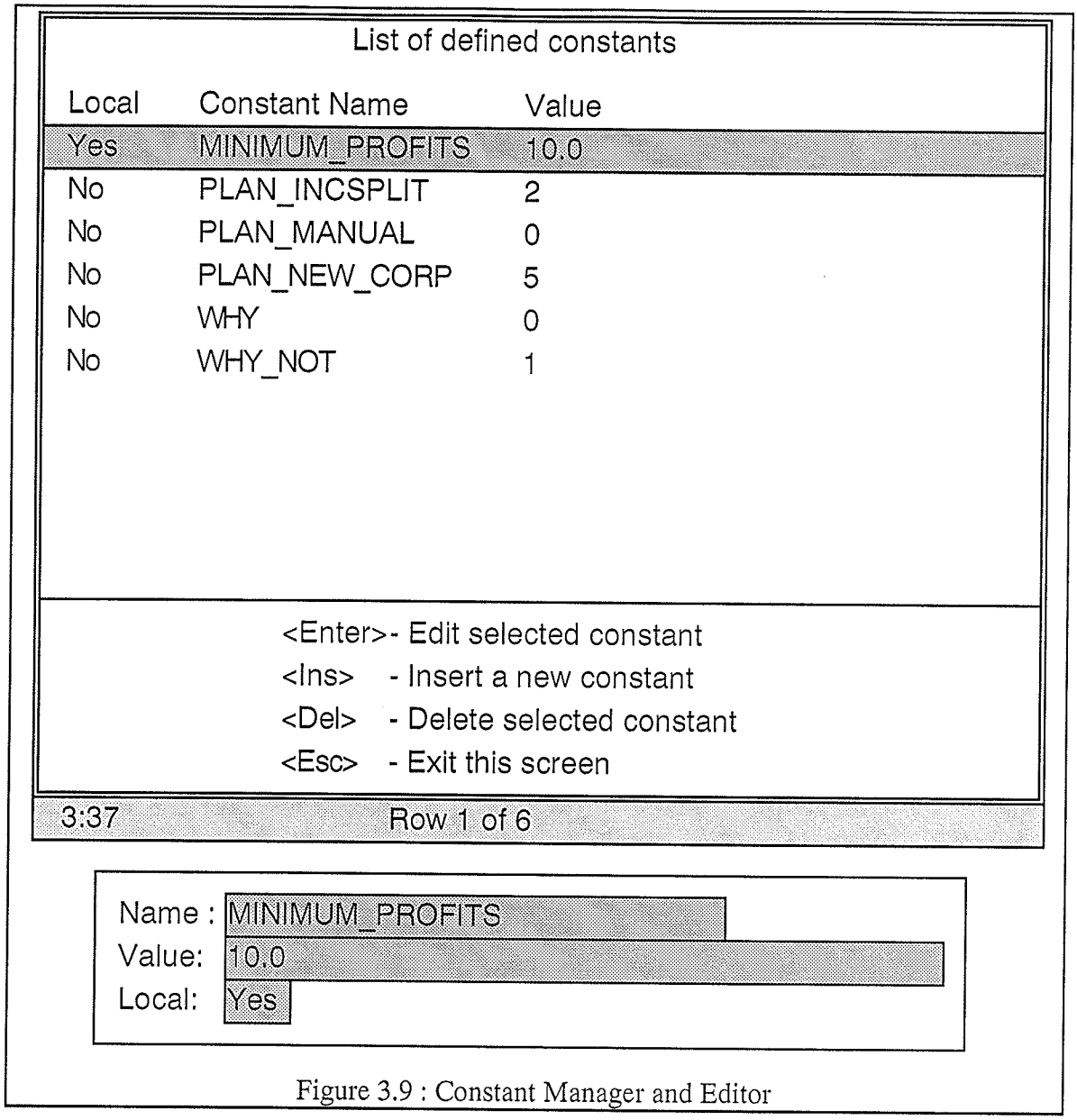

Figure 3.9 : Constant Manager and Editor

3.4.2 Automatic Working Memory Construction:

Another important feature of KA is its ability to construct a working memory structure for the rule base to use. This is accomplished by a section that parses the rules and extracts the names of variables that are used within those rules. Pre-defined constants, although they look just like variable identifiers are not included as variables within the working memory. Rather, their values and the values of data literals that the variable identifiers are assigned

and compared to are used to decide the types of the variable identifiers. The ability of KA to automatically build the working memory structure for a rule base alleviates the user from having to perform this task by hand. As well it eliminates the chance for error on the part of the user performing this task. A display screen that shows all variables within the working memory structure allows the user to set various attributes of the working memory variables (see Figure 3.10). Some of these attributes depend on the actual types of the variables but most types allow a default value, and user-defined prompt string for the default get routine. Get routines will be discussed below.

3.4.3 Working Memory Variables (Objects):

Variables that exist within the working memory of a rule base in KA are given special types. These types are similar to standard C data types but actually act more like data objects. There are six such types allowed. They include an integer type, a floating point type called *double*, a special string type that remembers all string literals that are assigned and compared to it, a boolean type, a set of string type, and a set of integer type. The integer, double, boolean, and string types have built-in *get* and *set* routines associated with them (see Figure 3.11). These routines are defined so that the user, upon initial development of a rule base, does not have to worry about writing routines necessary to allow him or her to enter values for the variables used by the rule base. This allows the user to quickly prototype and test a rule base and worry about data entry later. The set types can be ordered and have routines associated with them that allow one to maintain them. These routines include initialization, an *IN* function, a function to retrieve the rating of a particular set element, an *ADD* function, a *REMOVE* function, a *CLEAR* function, functions to set the current element pointer to the first, last, next, and previous set elements, and a routine to retrieve the current set element (see Appendix C). Thus the data objects are designed to allow the user to do as little work as possible before getting a rule base up and running.

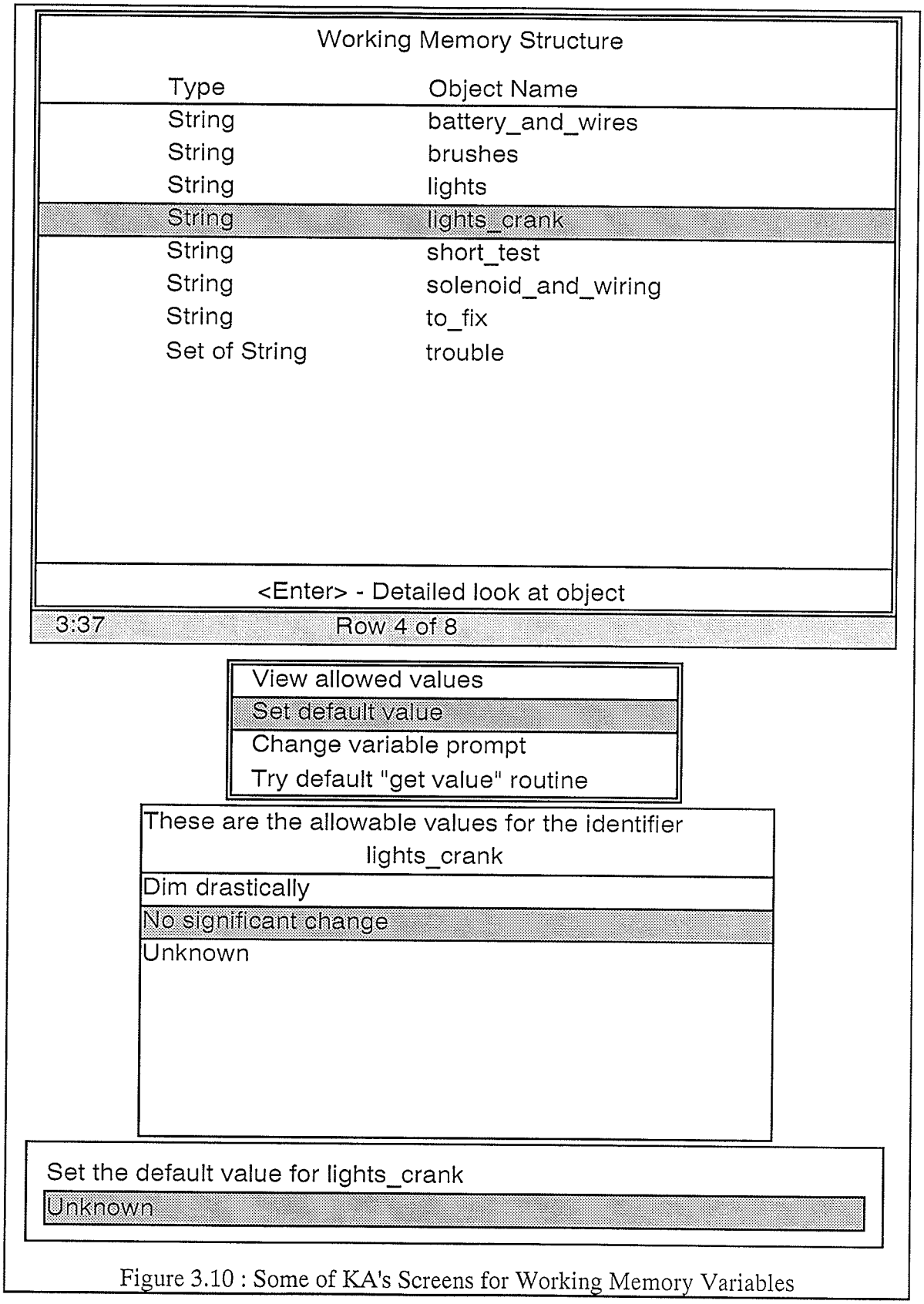

Figure 3.10 : Some of KA's Screens for Working Memory Variables

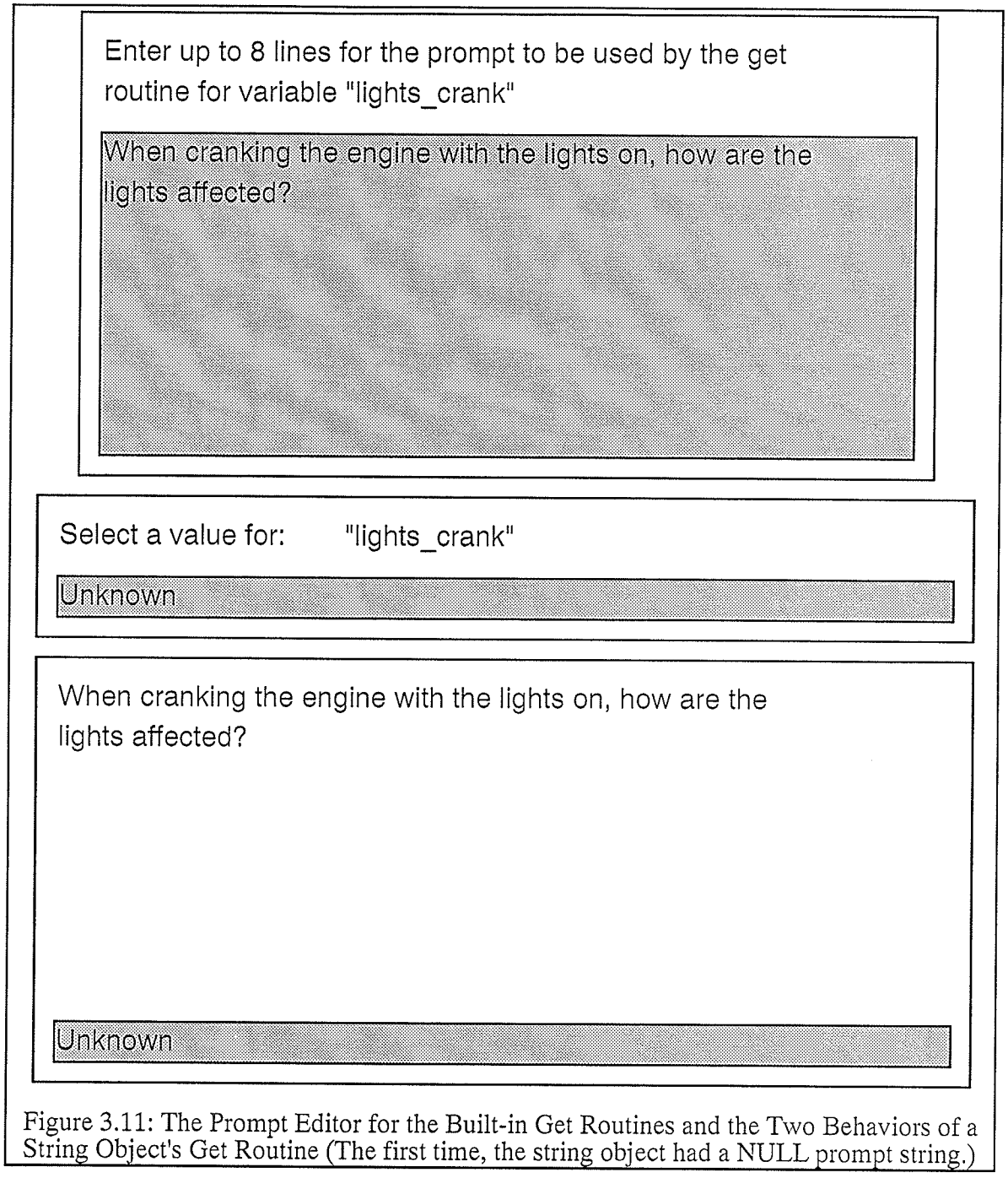

Figure 3.11: The Prompt Editor for the Built-in Get Routines and the Two Behaviors of a String Object's Get Routine (The first time, the string object had a NULL prompt string.)

3.4.4 Test Function:

KA provides a facility that builds an executable file from the user's rule base. This executable file contains code that will set up and execute a rule base, later allowing the user

to view the results of that execution. This file is built in the following fashion. Before the executable file is built, if goals exist in the rule base, the user is prompted whether the forward or backward chainer should be used. Next, KA outputs a combination of rule base and C code to a file called "KATMP.RB". If this is successful, RBX is called with "KATMP.RB" as its input. If errors result, they are placed in a file called "KATMP.RBX", the process is aborted, and the contents of that file will be displayed for the user. If no errors result, a file called "KATMP.C" will result and the C compiler will be called. This may also produce errors, and if so, they will be placed in a file called "KATMP.LIS" along with the C code that was processed, the process will be aborted, and "KATMP.LIS" will be displayed for the user. If no errors resulted from the compile, the linker is called. It links the file "KATMP.OBJ" with various needed run-time libraries. This too may produce errors, in which case they will be placed in a file called "KATMP.LL", the process will aborted, and "KATMP.LL" will be displayed for the user. If the linker produces no errors, a valid executable file called, "KATMP.EXE" will result. KA then runs this program for the user. Let us call this program the *test program*.

The test program first runs the rule base. This may cause the user to be prompted for various data values that the rule base requires in its processing. The values of all variables of type string, integer, double, and boolean are retrieved only through the built in get routines. If the value of a variable is unknown when its associated get routine is called, that get routine will prompt the user to input a value for that variable. Once the rule base has completed, the user may examine various parts of the rule base results and the working memory to verify that it has run correctly. See Figure 3.12 for an example of the main screen for a test program. The user may view: a display of the contents of all working memory variables at the point when the rule base completed execution, the rule base activity list (created by *rb_display_activity*), or may view an explanation that is constructed from

the *BECAUSE* clauses of all rules that were fired (forward chainer only). The user may also choose to re-run the rule base or to return back to KA's main menu.

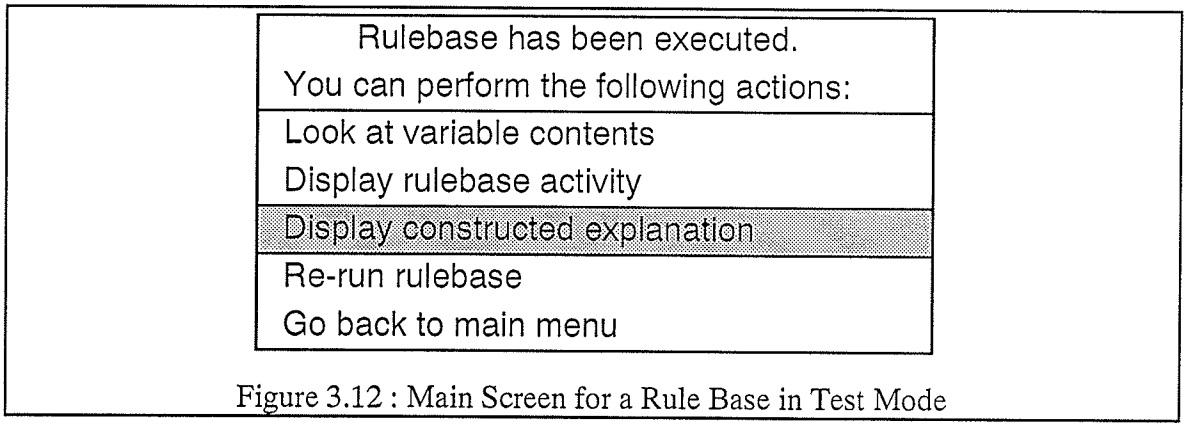

Figure 3.12 : Main Screen for a Rule Base in Test Mode

3.4.5 Stand-alone Program:

KA provides a feature to allow the user to create a stand-alone program from the current rule base. The base name of the file used to store the rule base in binary format is used as the default base for the files produced. First a rule base text file is created. It is then translated by RBX, and if successful, compiled and linked into an executable file. This feature is almost identical to the *Test* feature of KA. The only differences are that the last option of the main menu of the program reads "Quit Program" instead of "Go back to main menu" and that the names of the files produced are different.

3.4.6 Preface Code:

Sometimes it is necessary to add specialized C code to the beginning of a rule base file. This C code may include local routines that are to be called by rules, structure definitions, or simply C include statements that specify header files that need to be included. KA provides a facility to do this with its preface code section. Each line of code that is entered has an associated attribute that indicates which rule base files it should be added to (Export Only, Test and Stand-alone only, or both.) An example of how this facility is used is presented in Chapter Four.

3.4.7 Exporting Rule Bases From KA:

An important function of KA is its ability to export rule bases to be embedded within an existing or evolving application. This is done through a mechanism that allows the user to map working memory variable names to other variable names that may be fields in complex C data types or may be names of functions or procedures that return or set the required values.

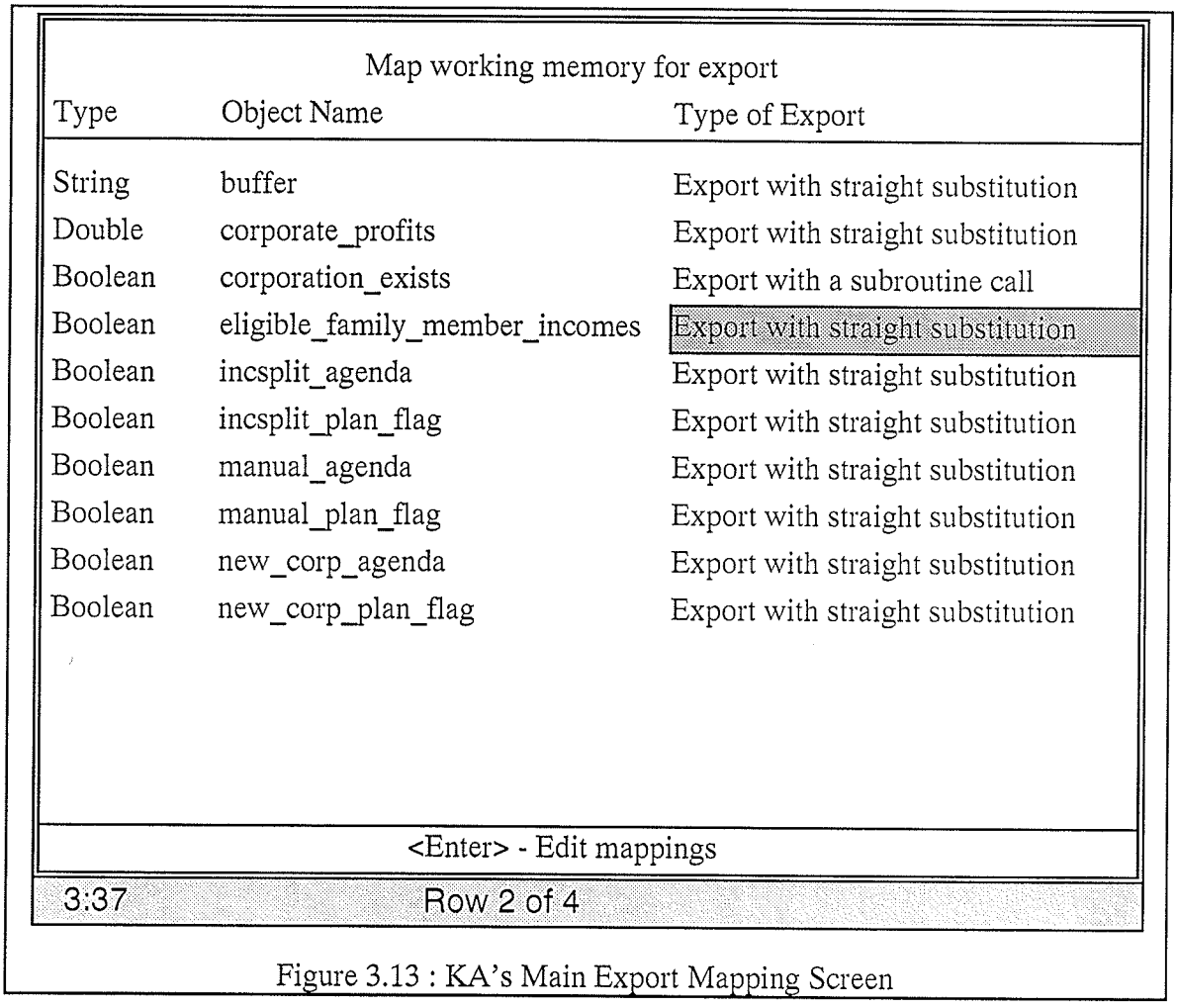

Figure 3.13 : KA's Main Export Mapping Screen

The main export screen (see Figure 3.13) presents the user with a list of the working memory variables with an attribute that indicates how they are being mapped. This attribute can have three different values, "Variable is local to the rule base", "Export with straight

substitution", and "Export with a subroutine call". If the variable is local, no mapping is allowed.

Exporting with straight substitution allows the user to map the variable to any string the user enters (see Figure 3.14). Whenever that variable appears in a rule, the substitution string the user entered is substituted for the variable name. A set routine name and an associated argument string may optionally be entered. This provides for the case if a routine is needed to set the value of the mapped variable in some special way. (For example, suppose a particular data structure exists that has an associated set of subroutines defined for it that perform all possible operations on it. Thus, assignments may not be possible to such a data structure except through one of these subroutines.) The set routine will only be used (if one is entered) in cases where the original working memory variable is being assigned a value.

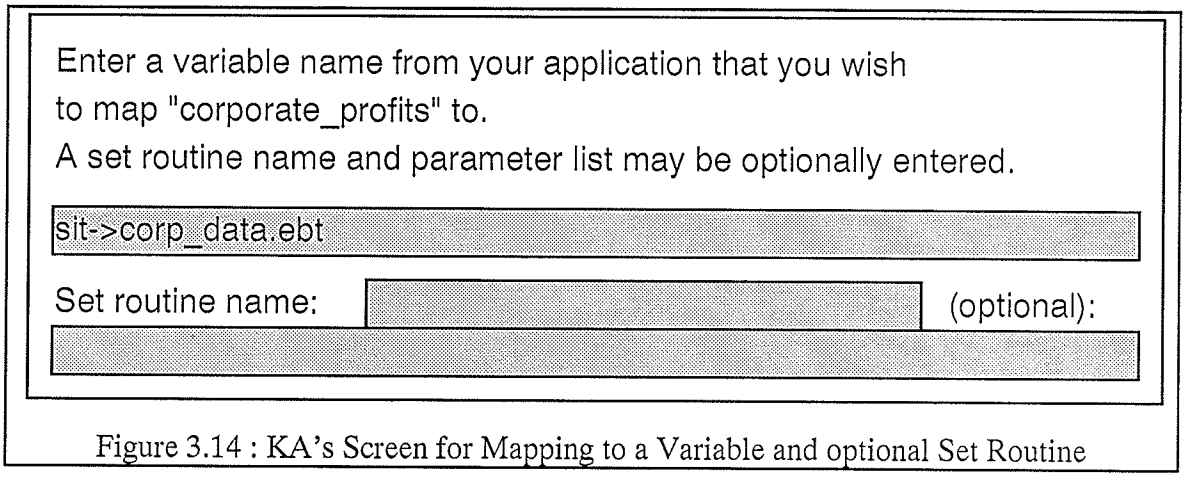

Figure 3.14 : KA's Screen for Mapping to a Variable and optional Set Routine

Exporting with a subroutine call (see Figure 3.15) can be used when the value of the variable needed is not directly accessible or must be computed. The subroutine name that is entered must be a function that returns a result of the proper type (*int*, *double*, *boolean*, *character pointer*, etc.). An argument list string can also be entered (to allow the get routine

to accept parameters). An optional set routine may also be entered. It operates in the same fashion as the set routine for exporting with straight substitution.

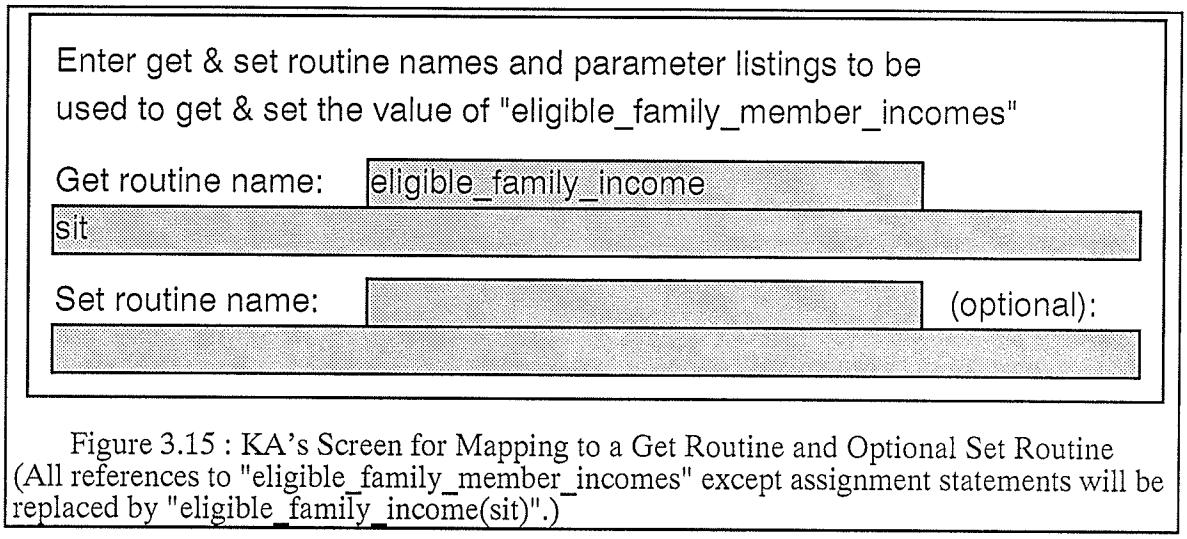

Figure 3.15 : KA's Screen for Mapping to a Get Routine and Optional Set Routine (All references to "eligible_family_member_incomes" except assignment statements will be replaced by "eligible_family_income(sit)".)

The user is also allowed to add additional outside elements to the working memory that may be of any valid C data type (see Figure 3.16). Fields within these additional elements are likely what various working memory variables will be mapped to when straight substitution is used. An associated export function outputs a .RB file that contains the rule base code and an external C routine that sets up, calls, and cleans up a call to the actual rule base. The user may specify which added working memory elements should be parameters of this external routine. The export facilities described here give the ability to produce an initial prototype of a rule base without regard to the specifics of the names and locations of an application's data that will later be needed to embed the rule base into that application. Later, when the prototype is complete, the user can proceed with mapping working memory variables as described above to allow the rule base, when exported, to work with data items that exist within that application.

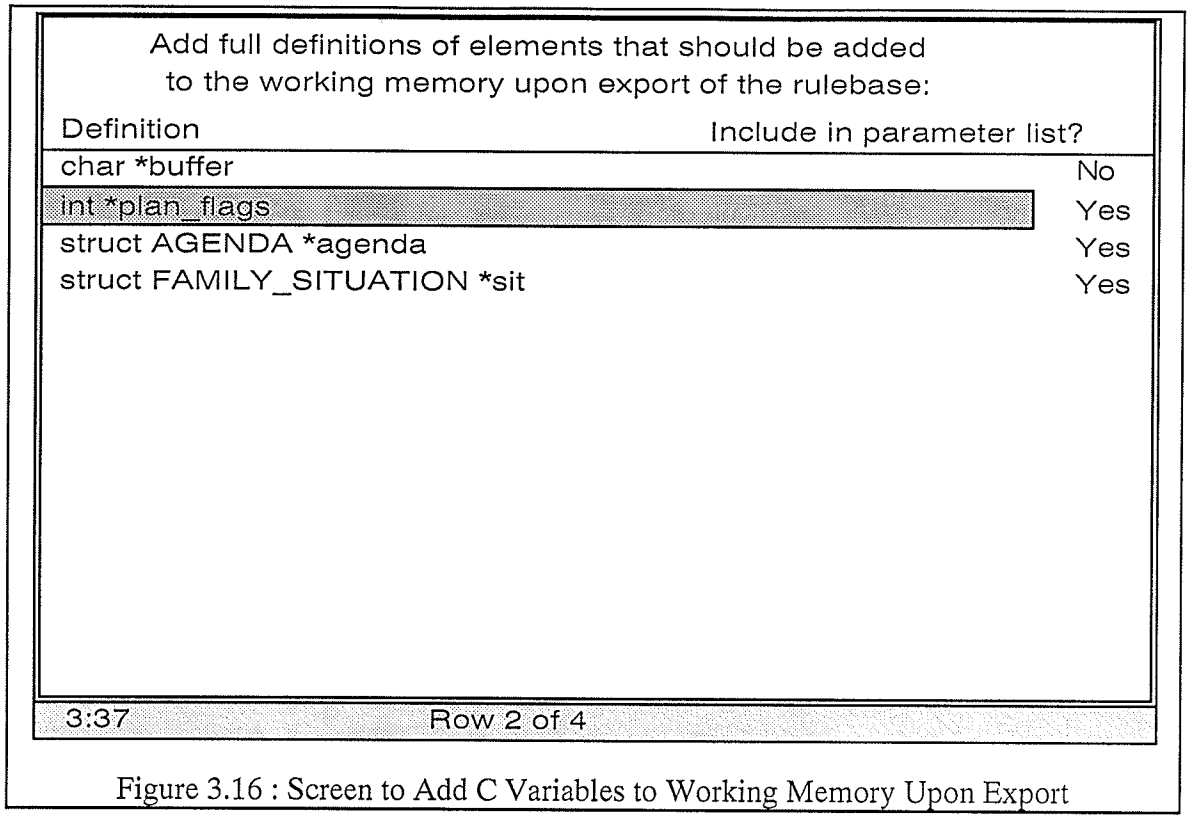

Figure 3.16 : Screen to Add C Variables to Working Memory Upon Export

3.4.8 Rule Base Report:

Lastly, KA provides a mechanism to produce a hard copy report of a rule base. It includes the name of the rule base, a list of the goals (if any) that exist in the rule base, a list of the working memory variables and how they are exported, a list of the additional working memory variables that are added when exporting, a printout of the text of all the rules, and a subroutine prototype of the procedure used to invoke the rule base from an external module. See Appendix D for an example of such a report.

Now that all the tools and support routines of EX-C have been discussed, it seems appropriate to note which of them I was directly involved with designing and implementing and which I was not. I was the main designer and implementor of the RBX translator and the backward chainer. As well, I designed the language extensions and run-time routines related to the backward chainer. Lastly, I designed and implemented all aspects of the KA

tool. The other aspects of EX-C, (the forward chainer, the rule base language, and the original design of the rule base representation data structures), were created by another graduate student.

One can see that the package of integrated tools presented in this chapter is extensive. However it is difficult to convey the feel of such a such a package through words alone. Users of EX-C have been pleased with the way it works and with the ease in which rule bases can be developed and embedded within applications. There are limitations to these tools that have yet to be addressed. These will be examined in the next chapter that discusses examples of how the tools are used.

Chapter Four: Using the Tools (Examples):

In the preceding chapter, we examined the various tools that comprise the EX-C development package. The goal of this chapter is to show how those tools were used to develop various rule bases. Three examples will be discussed, two of which have been embedded into actual applications. The first example is a small rule base that diagnoses automobile problems related to the starter and ignition systems. The second example is a rule base that exists as one component of a fertilizer advisor expert system. The rule base adjusts the toxicities of particular fertilizers and produces relevant explanations. The third example involves an application called *TAXPLAN*, a financial planning expert system. Three EX-C rule bases are currently embedded in *TAXPLAN*. Two of them, *SPLIT_RB* and *FAM_PLAN* will be presented. *SPLIT_RB* is used to build an ordered set of tasks that need to be executed to perform an optimum distribution of a family-owned corporation's income to the various members of that family. FAM_PLAN is used by the part of the system that determines which of the available family-oriented plans are applicable the current situation of the current family.

4.1 *STARTER* Example:

Let us begin with the *STARTER* rule base. It is not embedded into any application but is presented here to show how one might start an application with a rule base created with EX-C possibly with the idea of later building around it. The knowledge for this rule base comes from Peter Jackson's *Introduction To Expert Systems* [Jackson, 1990]. The purpose of the rule base is to diagnose problems that can occur with an automobile's starting system. The system asks various questions from the user and, when done, produces an explanation of what the problem is and what should be done to fix it.

```
Rule   Battery
IF       lights = "Dim or not on"
THEN   set_add(trouble, "Battery wires", 1)
         to_fix = "Replace battery and/or connecting wires."
         rb_exit(0)
BECAUSE Since the lights are dim or don't work at all, the problem is probably the battery or its
connecting wires.

Rule   Battery ok
IF       lights == "Normal or bright"
THEN   battery_and_wires = "Good"
BECAUSE Since the lights are ok, we assume that the battery and wires are in good shape.

Rule   Starter 1
IF       battery_and_wires == "Good"
AND    lights_crank == "Dim drastically"
THEN   set_add(trouble, "Starter", 1)
         set_add(trouble, "Solenoid and wiring", 1)
         to_fix = "Have the starter tested or install a rebuilt unit."
         rb_exit(0)
BECAUSE Since the battery and its connecting wires are ok and the lights ...

Rule   Solenoid and wiring 1
IF       short_test == "Starter works"
THEN   set_add(trouble, "Solenoid and wiring", 1)
         to_fix = "Check the the solenoid and the ignition switch wiring.  Replace
         either if necessary."
         rb_exit(0)
BECAUSE Since the starter works when the solenoid and ignition switch ...

Rule   Brushes
IF       set_in(trouble, "Starter")
AND    short_test == "Starter does not work"
AND    solenoid_and_wiring == "Needs replacing" || TRUE
AND    brushes == "Bad"
THEN   set_add(trouble, "Brushes", 1)
BECAUSE The starter still does not work even when the solenoid is shorted.

Rule   Starter 2
IF       brushes == "Good"
AND    solenoid_and_wiring == "Good"
THEN   set_add(trouble, "Starter", 1)
         to_fix = "Have the starter tested or install a rebuilt unit."
BECAUSE Since everything else seems ok, so the starter must be the problem.

Rule   Solenoid and wiring 2
IF       solenoid_and_wiring == "Needs Replacing"
THEN   set_add(trouble, "Solenoid and wiring", 1)
         to_fix = "Check the solenoid and the ignition switch wiring.  Replace
         either if necessary."
         rb_exit(0)
BECAUSE Since the solenoid and ignition switch wiring look bad, the ...
```

Figure 4.1 : *STARTER* Rules

The rules were entered using the rule editor of the KA tool. See Figure 4.1 for a listing of

these rules. The editor allows the user to associate a sequence number with each rule. This sequence number is used to order the rules within the rule base. Since this rule base uses the forward chainer, the order in which they appear is important. The contents of the rules caused KA to build the working memory shown in Figure 4.2. Each variable that KA finds within the rule base is given a type by examining the context in which it appears. For example, the variable "lights" is compared to the string "Dim or not on" in the first clause of the premise of rule *Battery*. This causes the working memory builder function of KA to determine that *lights* should have the type *String*. Certain conflicts may arise within particular rule bases. These are flagged with warning messages. As well, the working memory builder may not be able to determine the type of some variables. In this case, the user is asked once to provide a type. After that, the builder will use that type unless, (after rules are changed), the builder can infer its own type.

Type	Variable	Allowable Values for Strings
String	battery_and_wires	"Good", "Other than Good", "Unknown"
String	brushes	"Bad", "Good", "Unknown"
String	lights	"Dim or not on", "Normal or Bright", "Unknown"
String	lights_crank	"Dim drastically", "Other than Dim drastically", "Unknown"
String	short_test	"Starter works", "Starter does not work", "Unknown"
String	solenoid_and_wiring	"Needs replacing", "Good", "Unknown"
String	to_fix	(values too long to show)
Set of String	trouble	N/A

Figure 4.2: Working Memory for the *STARTER* Rule Base

There are three subroutines that are called from within the rules of the *STARTER* rule base. These are *set_in*, *set_add*, and *rb_exit*. The first two are built-in routines that operate on variables of the two set types, *Set of String* and *Set of Integer*. These routines are discussed in Appendix C. The *rb_exit* routine allows a rule to halt the chaining process

even if all applicable rules have not yet been fired. The integer passed as a parameter is stored in the *exit_code* field of the results structure.

The *STARTER* rule base can be tested using the *Test Rule Base* function of the KA tool. This function creates a .RB file with a special name that contains the rules in the *STARTER* rule base and code that allows the user to run the rule base, view the contents of the working memory (Figure 4.3), view an explanation of the rule base's reasoning (constructed from the *BECAUSE* strings of each rule fired) (Figure 4.4), and see the activity list created by the rule base processor (an example of such a display is shown in Figure 3.2). One can see that the consultation that produced Figure 4.3 indicated that the trouble was in the solenoid and ignition switch wiring. The prescribed actions were to check and replace them if necessary. The code also allows the rule base to be run as many times as needed. However, simply having a .RB file is not sufficient. The RBX translator is then called to create the appropriate C code. This C code is then compiled and finally linked with the required libraries to produce an executable file. This executable file is then called to let the user test the rule base.

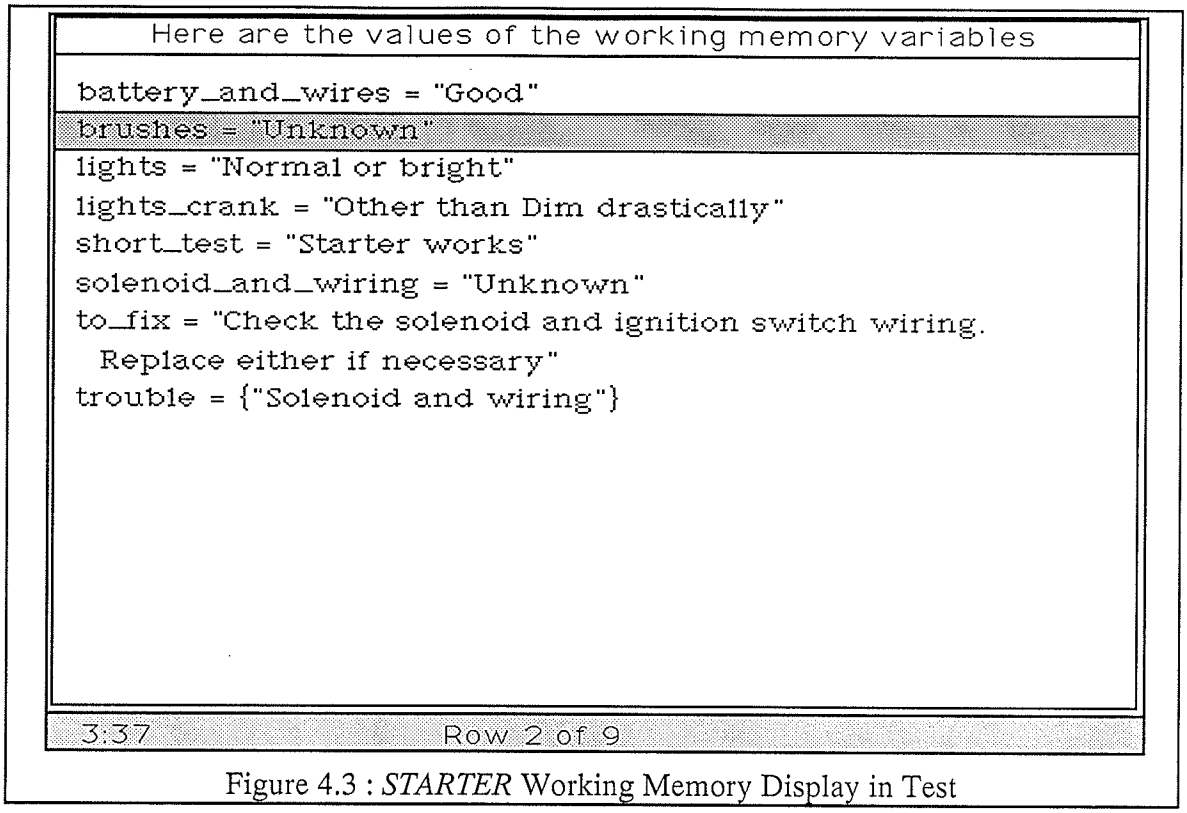

Figure 4.3 : *STARTER* Working Memory Display in Test

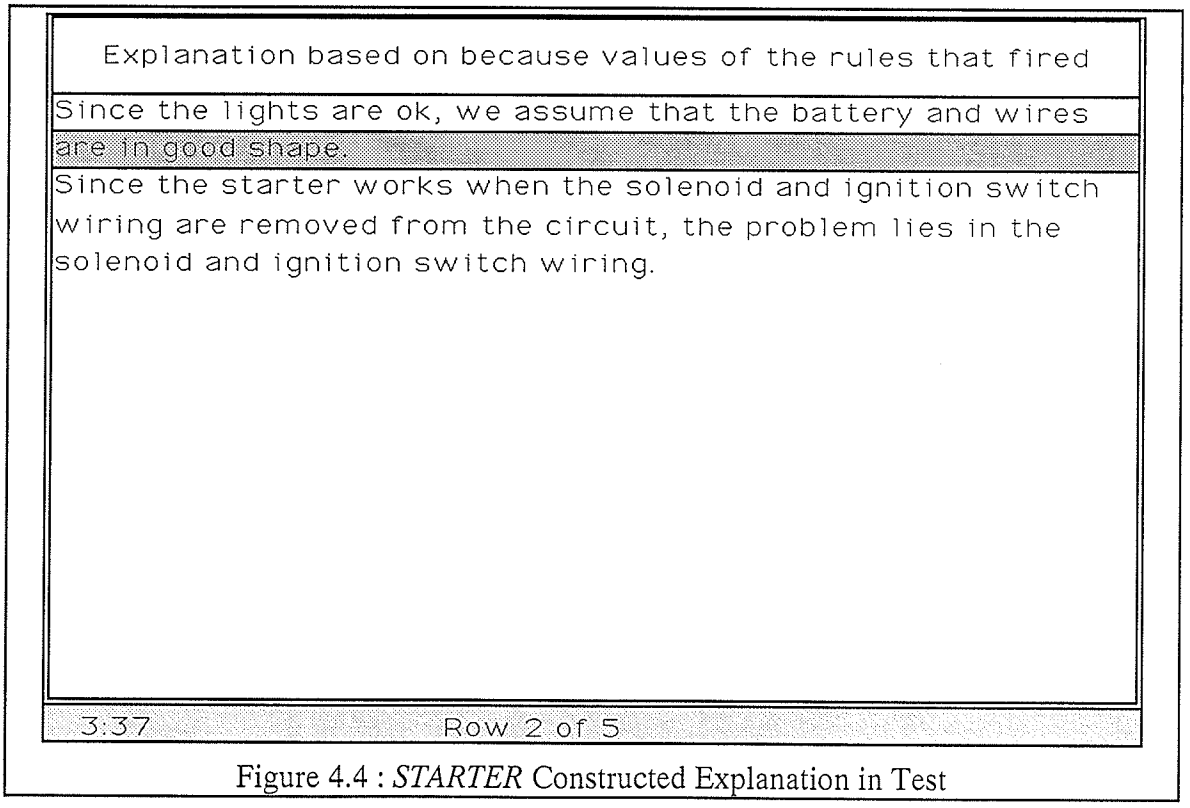

Figure 4.4 : *STARTER* Constructed Explanation in Test

Finally, the *STARTER* application was created using the *Build Standalone* function of the KA tool. This function is similar to the test function, but it gives the files created a base name that the user supplies, some of the code is slightly different, and the final executable file is not called.

4.2 *FERT_TOX* Example:

The Fertilizer Advisor system (*FA*) is an expert system that maintains information about various crops, fertilizers, crop yield tables and the information about particular fields of clients. Based on this information and their associated costs, fertilizer recommendations are generated that should produce the optimum yield for those costs. The *FA* is "designed to provide assistance to farmers in making decisions about rates, sources, timing, and methods of applying fertilizers" [Li, 1991].

The toxicity of a fertilizer indicates how much of the fertilizer may be applied with the seeds, in the seed row, when they are being planted [Li, 1991]. Applying the fertilizer in this fashion is usually very cost-efficient and thus is important in building the best recommendation [Li, 1991]. The field condition and the type of implement used by the farmer may have an affect on this toxicity for some fertilizers. The rule base presented here customizes the toxicity of various fertilizers in the system based on the current client's field conditions and the type of implement being used. Figure 4.5 shows how the *FERT_TOX* rule base is used within *FA*. *FERT_TOX* takes information about the client to make changes to the toxicities of certain fertilizers. This operation is performed before any other as the toxicities of the fertilizers must remain constant throughout the rest of the processing required to generate a valid recommendation.

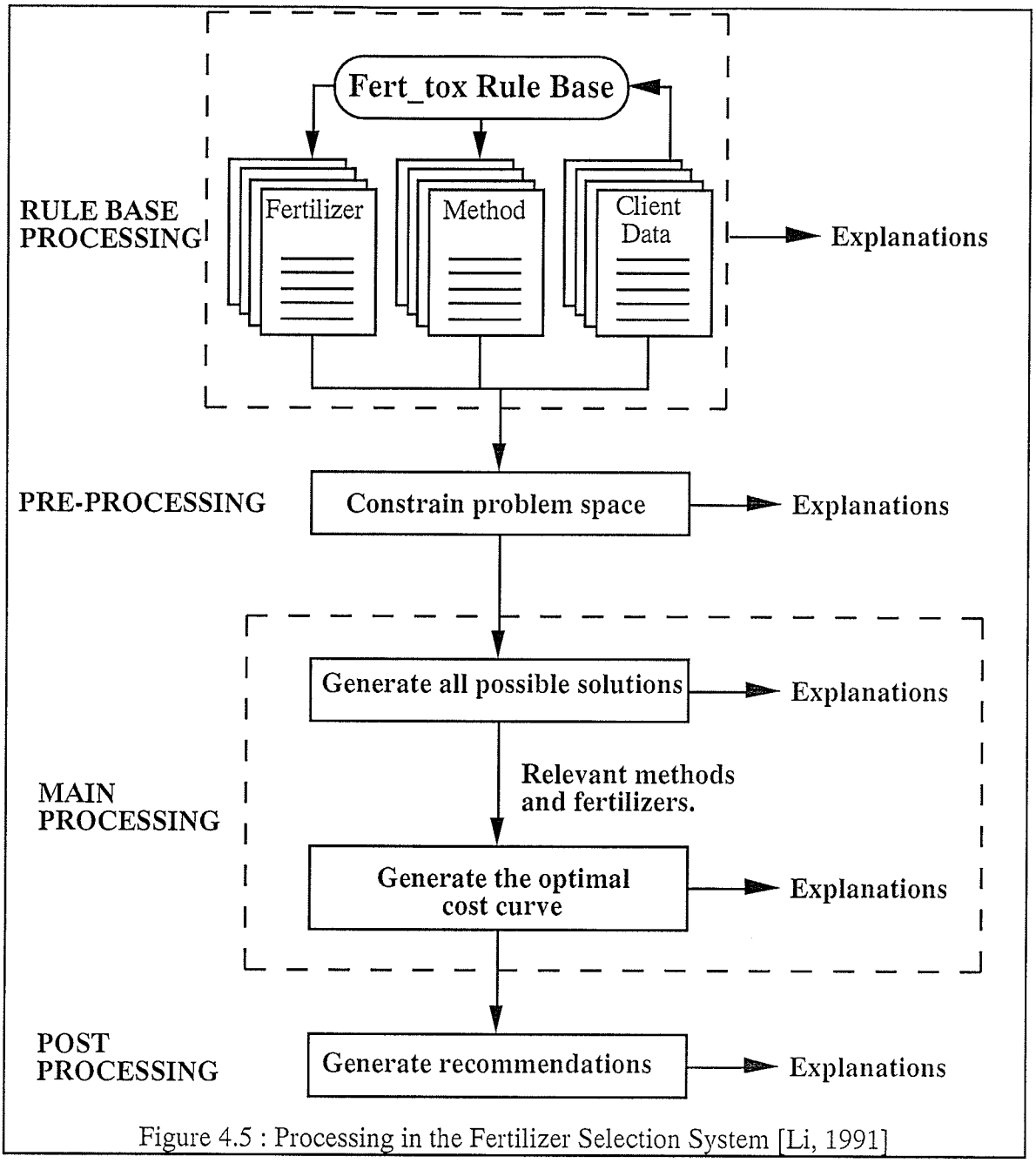

Figure 4.5 : Processing in the Fertilizer Selection System [Li, 1991]

A partial listing of the rules is shown in Figure 4.6. They were entered, like the *STARTER* rule base, with KA's rule editor. The working memory created by KA for this rule base is shown in Figure 4.7.

```
Rule    Set Up
IF      TRUE
THEN    adjustments_made = FALSE
        explain = " "

Rule    Coarse
IF      soil_texture ==  TEXTURE_COARSE
THEN    urea_fert_safe = "NO"
        explain = txt_join(explain, "\xf9 Soil texture is Coarse. Therefore Urea-based fertilizers", NULL)
        explain = txt_join(explain, " should not be applied in the seedrow.\n\n", NULL)

Rule    Not Moist and Not Coarse
IF      moisture_condition !=  MOIST_COND_DRY
AND     soil_texture !=  TEXTURE_COARSE
THEN    urea_fert_safe = "YES"

Rule    Discer Implement
IF      seeding_implement == IMPLS_DISCER
THEN    toxicity_all = "LOW"
        explain = txt_join(explain, "\xf9 Since you are using a DISCER and you growing", NULL)
        explain = txt_join(explain, " conditions are suitable, you can safely apply more Nitrogen fertilizer
                in the seedrow.\n\n", NULL)

Rule    Scattered Airseeder
IF      seeding_implement ==  IMPLS_AIRSEEDER
AND     seeding_type == IMPLS_TYPE_SCATTERED
THEN    toxicity_all = "LOW"
        explain = txt_join(explain, "\xf9 Since you are using an AIRSEEDER and scattering the", NULL)
        explain = txt_join(explain, " seed and fertilizer, and your growing conditions are suitable, you can
                safely apply additional nitrogen fertilizer in the seedrow.\n\n", NULL)

Rule    Urea Safe but Low Toxicity
IF      urea_fert_safe == "YES"
AND     toxicity_all == "LOW"
THEN    adjust_toxicity(urea_toxicity, 40.0)
        adjust_toxicity(uan_toxicity, 60.0)
        adjust_toxicity(am_nitrate_toxicity, 90.0)
        adjustments_made = TRUE
        explain =txt_join(explain, "\xf9 The seedrow toxicity limits for UREA, UAN Liquid, and",NULL)
        explain =txt_join(explain, " Amm. Nitrate have been doubled due to conditions described
                above.\n\n", NULL)

Rule    Urea Unsafe and Low Toxicity
IF      urea_fert_safe == "NO"
AND     toxicity_all == "LOW"
THEN    adjust_toxicity(urea_toxicity, 0.0)
        adjust_toxicity(uan_toxicity, 0.0)
        adjustments_made = TRUE
        explain = txt_join(explain, "\xf9 The seedrow toxicity limits for UREA and UAN Liquid ",NULL)
        explain = txt_join(explain, "have been set to zero because these Nitrogen fertilizers contain Urea.",
                NULL)
        explain = txt_join(explain, "The seedrow toxicity limit for Am.Nitrate has been doubled", NULL)
        explain = txt_join(explain, " because of the conditions described above (Note that Am. Nitrate does
                not contain Urea).\n\n",  NULL)
```

Figure 4.6 : Some of the *FERT_TOX* Rules

Type	Variable	Exported As
Boolean	adjustments_made	client->adjustments_made
Double	am_nitrate_toxicity	&fert[AM_NITRATE]
String	explain	*client_expl
String	moisture_condition	client->moist_cond
String	seeding_implement	client->seed_impl
String	seeding_type	client->seed_impl_qual
String	soil_texture	client->texture
String	toxicity_all	(local variable)
Double	uan_toxicity	&fert[UAN]
String	urea_fert_safe	(local variable)
Double	urea_toxicity	&fert[UREA]

Working memory variables added when exporting:
parameter: char **client_expl
parameter: struct CLIENT_DATA *client
parameter: struct FERTILIZER *fert

Exported rule base subroutine prototype:
```
extern void analyse_client_data(char **client_expl, struct
        CLIENT_DATA *client, struct FERTILIZER *fert);
```

Figure 4.7 : Working Memory for *FERT_TOX* Rule Base

In addition to the basic working memory information, Figure 4.6 also shows how the working memory variables are mapped when the rule base is exported. This mapping information was entered by the user of KA. To export the rule base, KA uses this information in the following way. First of all, the extra variables, *client_expl*, *client*, and *fert* are added to the working memory. Their types are also provided. (Note that although all the extra variables for this rule base are parameters, others could be added that are considered to be only local variables. Local variables are not considered to be arguments to the external function that is created.) Next, all non-local variables of the original working memory are mapped in some way to the extra variables. For example, *adjustments_made* is mapped to the boolean field *adjustments_made* of the *client* structure. Lastly, Figure 4.7 shows the external prototype for the routine created by KA to set up, execute, and clean up a call to the rule base processor for the *FERT_TOX* rule base. This subroutine has three arguments, the three additional working memory variables. This routine is called from within the fertilizer selection system so that the rules may modify the passed data

structures. The changes in those data structures are used elsewhere in the program.

The following, Figure 4.8, shows a sample client data entry screen from *FA*. Notice the values for the fields labeled *Soil Texture* and *Seeding Implement*. They cause three rules (*Coarse*, *Discer Implement*, and *Urea Unsafe and Low Toxicity*) from the *FERT_TOX* rule base to be fired. These rules adjust values in the fertilizer data structures (how this is done is explained below) and produce the explanations shown in Figure 4.9.

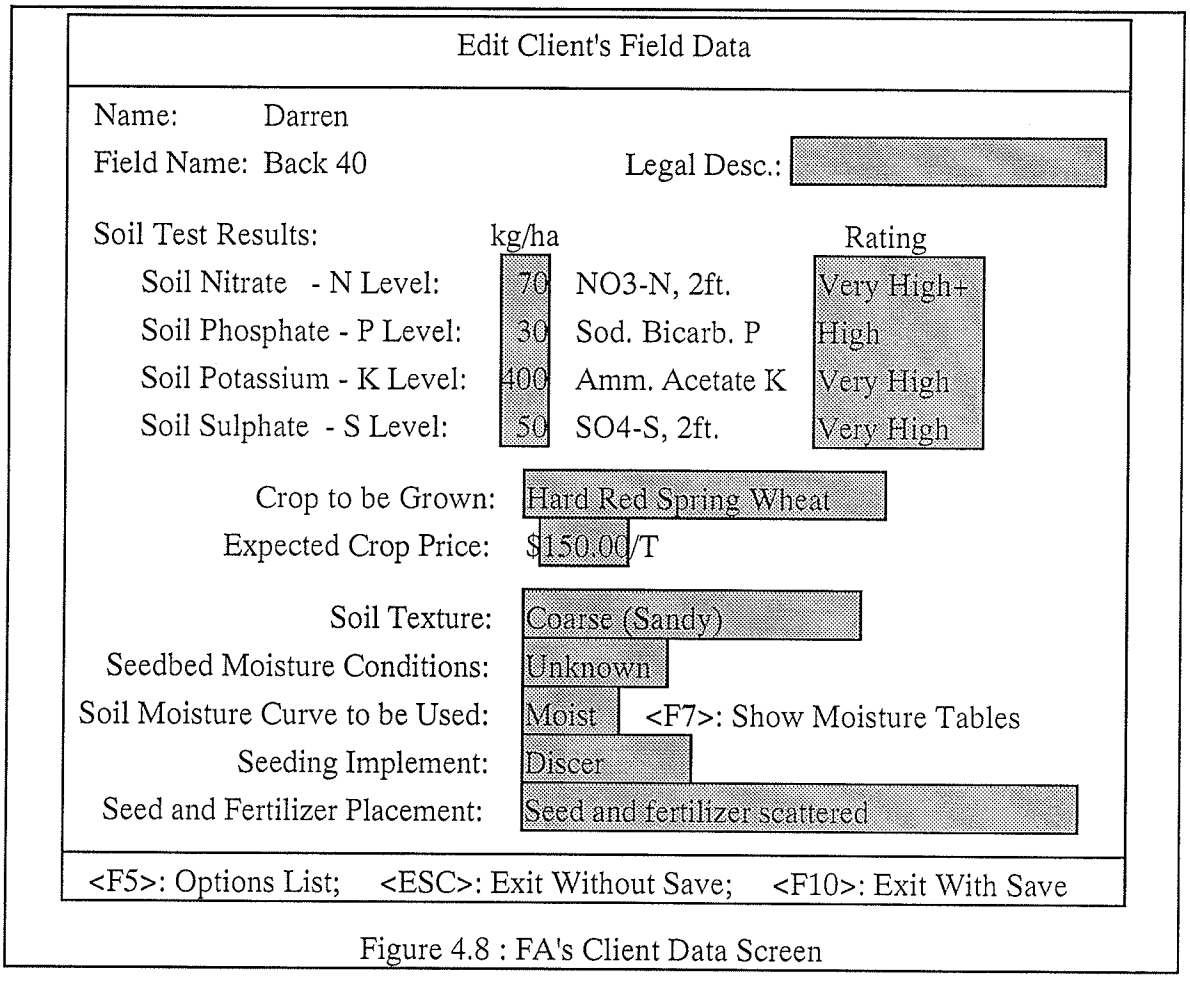

Figure 4.8 : FA's Client Data Screen

Adjustments made to the fertilizer toxicity limits
• Soil texture is Coarse. Therefore Urea-based fertilizers should not be applied in the seedrow. • Since you are using a DISCER and you growing conditions are suitable, you can safely apply more Nitrogen fertilizer in the seedrow. • The seedrow toxicity limits for UREA and UAN Liquid have been set to zero because these Nitrogen fertilizers contain Urea.The seedrow toxicity limit for Am.Nitrate has been doubled because of the conditions described above (Note that Am. Nitrate does not contain Urea).

Figure 4.9 : Explanations produced by FERT_TOX rule base

Notice that subroutine *adjust_toxicity* is called by many of the rules within the *FERT_TOX* rule base. This routine is defined locally within the KA tool with the *preface code* module (see Figure 4.10). This module of KA allows the programmer to define extra C code that should be placed at the beginning of the .RB file. Each line can be flagged to exist in the test version only (this includes the stand-alone version), the export version only, or both versions. *adjust_toxicity* was defined twice in the preface module, once for the test version and once for the exported version. The purpose of the routine is to modify two of the fields of a FERTILIZER structure. Since variables of this type only exist in the exported version (elements of the array *fert*), this routine is really only needed for that version. Therefore, a stub of the routine is created for the test version in order to give the rules a routine to call.

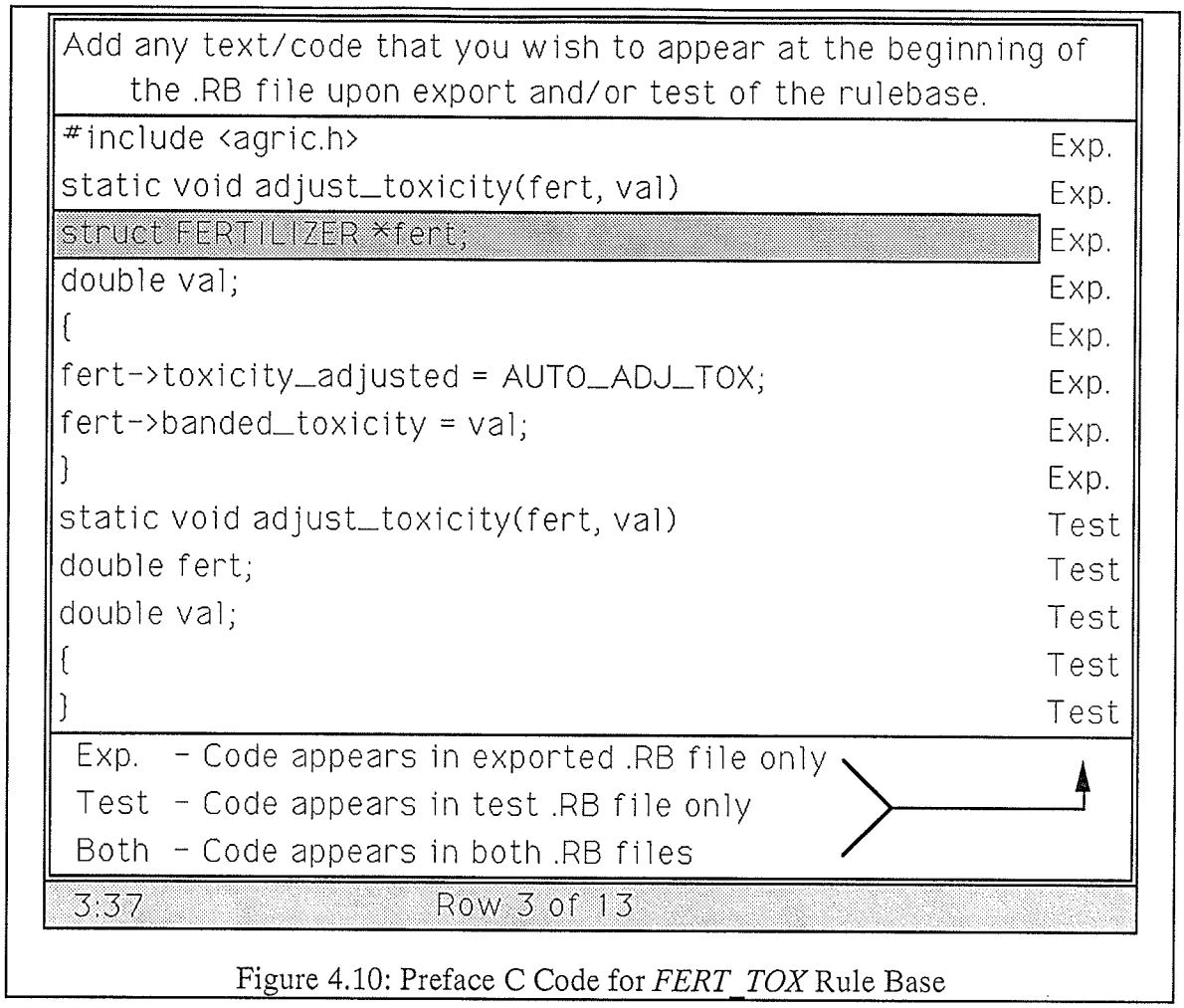

Figure 4.10: Preface C Code for *FERT_TOX* Rule Base

Even though the *FERT_TOX* rule base has been embedded within the fertilizer selection system, it still may be tested within the KA program. This is because no changes to the code exported by KA were needed to be made to successfully embed the rule base. Therefore, if the rule base ever needs to be modified, the KA tool can be used and no embedding changes have to be repeated.

4.3 *TAXPLAN* Example:

The third and final example consists of the two rule bases that are used by a financial planning expert system called *TAXPLAN*. The operation of *TAXPLAN* relies on two distinct databases: the tax database and the client database. The tax database contains rates and thresholds used in calculating federal and provincial tax: current tax rates for each income bracket and for each type of income (capital gains, dividends, interest, etc.). In addition, there are thresholds and tax credits set up to be modified whenever changes to the Canadian tax laws occur. The tax database contains a set of constants for each tax year. The client database stores data about families and their financial situations (assets, liabilities, incomes, and so on). Once a client situation has been loaded from this database, the financial planners typically follow the same iterative action path to optimize a client's situation. This path consists of an analysis of the current situation, selection of a plan or plans to explore, implementation of the plans resulting in new situations, an analysis of the new situations, and a possible save of the new situations to the client database. This process (shown in Figure 4.11) may be repeated until the planner is satisfied that the client's situation has been optimized [Evans, et al., 1990].

TAXPLAN contains three major components. These include: the databases, knowledge-bases, and conventional processing routines (see Figure 4.12).

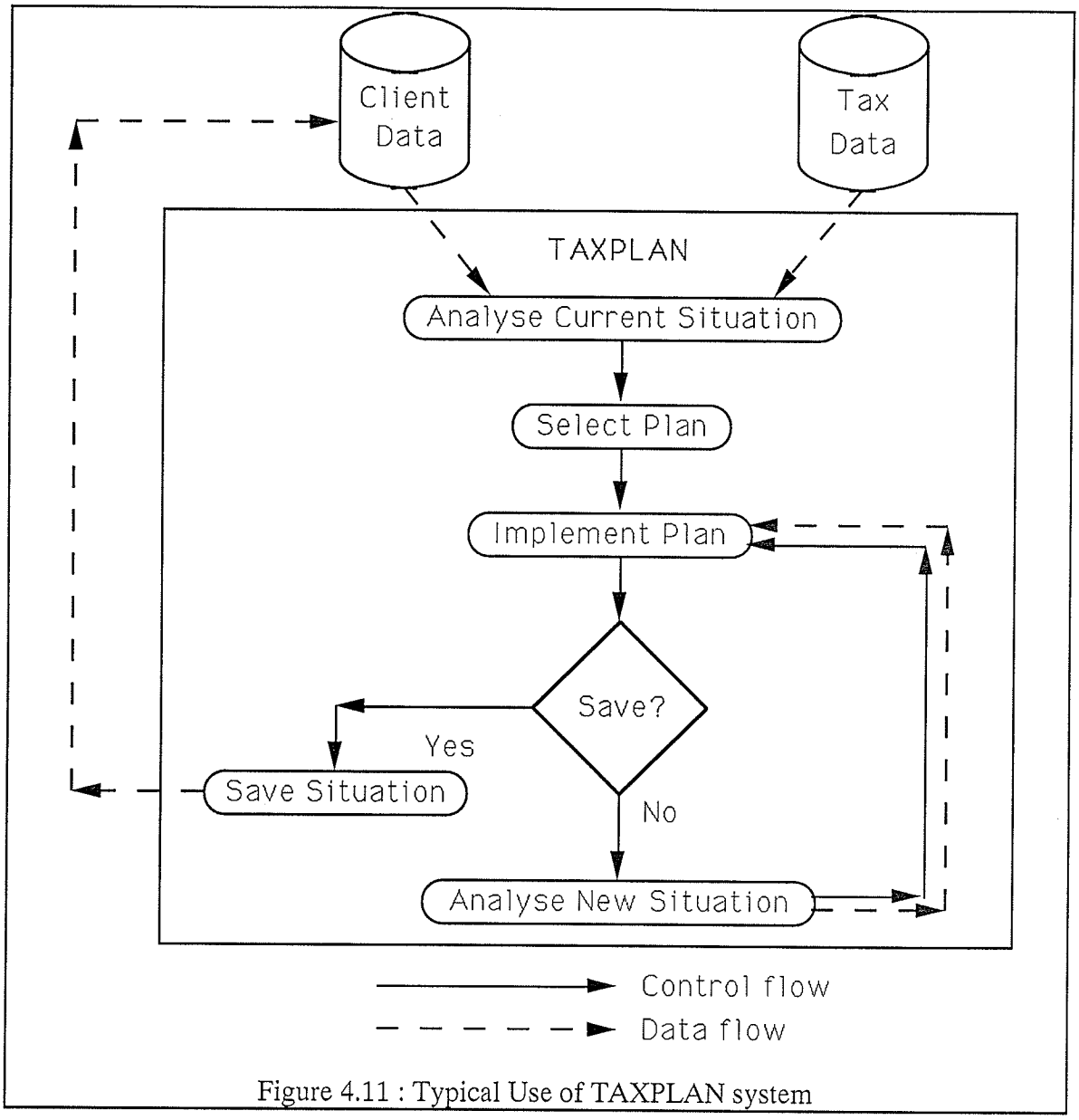

Figure 4.11 : Typical Use of TAXPLAN system

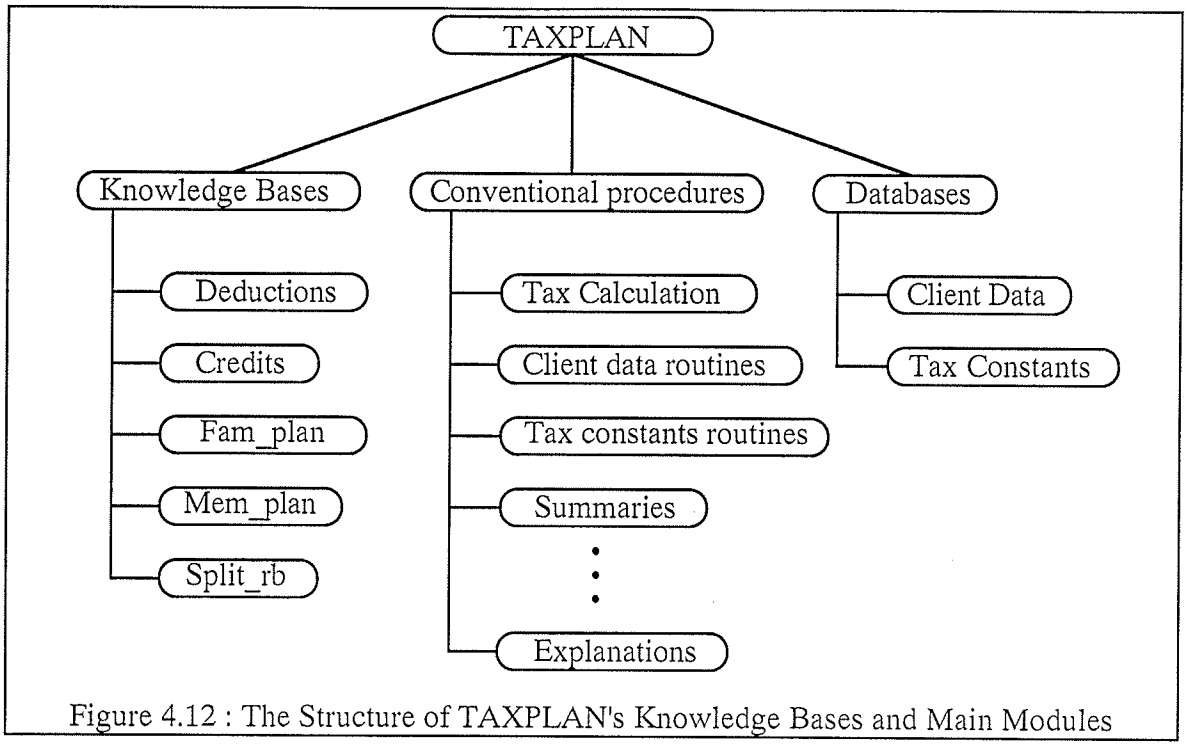

Figure 4.12 : The Structure of TAXPLAN's Knowledge Bases and Main Modules

The client database contains information about the entire family and information particular to individual members of the family. Examples of family specific information include information about a corporation that may be owned by the family, the desired income level of the family, expected retirement ages for the family heads, and whether the current standard of living should be maintained, increased, or decreased.

As mentioned above, the client database also stores information specific to each family member. This information includes personal information about each member of a family such as their name, age, their relationship to the rest of the family (husband, wife, child, etc.), the number of months they attend post secondary school, and whether they are disabled. In addition to personal information, TAXPLAN requires financial information about each family member, which includes various income and expense sources; amounts, frequencies, annual change, and duration. Detailed asset information is kept for the head(s) of the family, typically the husband and wife. The data is detailed enough to allow accurate

after-tax projections of asset values, the income they produce, and whether the income is to be reinvested or brought into cashflow. Other data specific to each family member are maintained, including: capital losses from previous years, capital gains deductions used in previous years, cumulative net investment loss (CNIL) from previous years, and alternative minimum tax (AMT) carry over from previous years.

Data in the tax database is grouped into four sections: marginal tax rate tables, tax credit constants, tax deduction constants, other constants. These data elements are grouped by year and used when calculating individual income taxes for a particular year.

Marginal tax rate is a term used to describe the rate at which the next dollar earned will be taxed. By identifying the income thresholds up to which different tax and surtax rates (eg. basic federal tax, federal surtax, provincial tax) are applied, one may derive a number of *tax brackets*. To further complicate matters, the tax rates are applied to different types of income differently. Thus, income types are grouped together into four groups, each with its own set of tax brackets and corresponding marginal tax rates. Rates such as these are useful in projecting future values of income generating instruments where taxes payable are removed before reinvestment. These are stored in tabular form, one table per year, in the marginal tax rates section of the tax database.

There are many constants involved when calculating the numerous tax credits that may be applied to individual income taxes. Some constants represent maximum allowable values for particular credits, some are actual credit amounts allowed for certain types of individuals, while others represent income thresholds for particular credits. These constants are stored in the tax credit group of the tax database.

In the Canadian income tax system, individuals are allowed certain deductions to be applied

to their total income, the result being their net income. These deductions, like credits, have various constants associated with them. These constants are stored in the tax deductions portion of the tax database.

Finally, there are many other constants associated with income tax calculation. Some of these constants represent tax rates and income thresholds for those rates. Others involve various provincial tax reductions.

Many components of TAXPLAN do not involve enough complexity to warrant the use of AI programming techniques. Thus, TAXPLAN contains a considerable amount of conventional computer code. Take an input screen tailored to a specific set of data for example: a fairly straight-forward (although sometimes tedious) task to create. Tasks of this sort are easily implemented with third generation programming tools. Therefore, the cost and effort required to apply AI techniques to such tasks is not worthwhile. We use an extensive set of general purpose utility routines to simplify a number of tasks including the creation of input screens. All code that handles the storage and retrieval of values from the two databases and calculation of taxes is implemented in this fashion. All data input and summary screens were also created conventionally.

Despite the large amount of conventional processing, some important sections of TAXPLAN involve the use of AI techniques. Due to their complexity, generation and maintenance of these sections is not practical using conventional techniques. These sections include the calculation of tax deductions and tax credits, automated income splitting, and plan selection.

4.3.1 *SPLIT_RB*:

Let us first discuss the *SPLIT_RB* rule base. It is used by a plan that performs family

corporate income splitting. This plan distributes income from a family-owned corporation to the various members of that family in a way that is the most beneficial to the family from a tax perspective. This distribution can result in a tremendous savings in taxes for the family if performed properly. The purpose of the rule base is to create a set of tasks for the system to perform, ordered in such a way so as to produce the best distribution. Figure 4.13 shows the rules contained in *SPLIT_RB*.

```
Rule    general_split
IF      TRUE
THEN    set_add(tasks, PROLOGUE, 1)
        set_add(tasks, CLEAR_MEM_CORP_INCOME, 2)
        set_add(tasks, GET_PLAYER_INFO, 3)
        set_add(tasks, EXPLAIN_CORP, 4)
        set_add(tasks, DIST_FAMILY_SALARIES, 10)
        set_add(tasks, DIST_DIVIDENDS, 11)
        set_add(tasks, DISP_SPLIT_RESULTS, 12)
        set_add(tasks, PLAN_SELECTION, 13)
BECAUSE Setup the income splitting tasks that always need to be done.

Rule    make_eligible_for_sbd
IF      profits > SMALL_BUSINESS_LIMIT
THEN    set_add(tasks, DIST_FAMILY_SALARIES_SBD, 7)
BECAUSE If corporate profits are greater than the small business deduction limit then try to reduce them
below the limit by distributing salaries to family members.

Rule    display_already_eligible_sbd
IF      profits <= SMALL_BUSINESS_LIMIT
THEN    set_add(tasks, EXPLAIN_ALREADY_ELIGIBLE_SBD, 7)
BECAUSE If corporate profits are under the small business deduction limit then put that in the explanation.

Rule    members_with_cnil
IF      members_with_cnil == TRUE
THEN    set_add(tasks, DIST_DIVIDENDS_TO_OFFSET_CNIL, 8)
BECAUSE If the family has members with CNIL, reduce that CNIL by distributing dividends.

Rule    rrsp_contributions_left
IF      avail_rrsp_contributions == TRUE
THEN    set_add(tasks, DIST_INCOME_FOR_RRSP_CONTRIBS, 9)
BECAUSE If any family members below the maximum RRSP contribution limit, distribute salary to them
so they can contribute up to the limit.
```

Figure 4.13 Rules in *SPLIT_RB* Rule Base:

SPLIT_RB was developed independently of *TAXPLAN*. Its rules were written using KA with *TAXPLAN* in mind, but the names of the data objects referenced within the rules bear

little resemblance to the actual data objects available within *TAXPLAN*. *SPLIT_RB* was tested with KA and when ready, its variables were mapped and the rule base was embedded into the *TAXPLAN* system.

Some of the variables within the working memory of *SPLIT_RB* can be mapped directly to actual data objects within *TAXPLAN*. For example *profits* which is the amount of earnings the corporation has before its taxes are paid can be mapped to the *ebt* (earnings before taxes) field of the *corp_data* structure which is a field of the family situation used within *TAXPLAN* (see Figure 4.14). Two of the variables, *avail_rrsp_contributions* and *members_with_cnil* do not represent data elements that actually exist within *TAXPLAN*. However, the required values of these variables can be obtained procedurally by accessing multiple pieces of *TAXPLAN's* data. Thus, two functions were written within the preface section of KA to obtain the required values for these variables (see Figure 4.15 to see the definition of *members_with_cnil*). The variables were then mapped to these functions.

Type	Variable	Exported As
double	avail_rrsp_contributions	avail_rrsp_contributions(sit)
double	members_with_cnil	members_with_cnil(sit)
double	profits	sit->corp_data.ebt
Set of string	tasks	agenda_tasks

Working memory variables added when exporting:
parameter: SET *agenda_tasks
parameter: struct FAMILY_SITUATION *sit

Exported rule base subroutine prototype:

```
extern void setup_income_split(SET *agenda_tasks, struct
        FAMILY_SITUATION *sit);
```

Figure 4.14 : *SPLIT_RB's* Working Memory and Export Data:

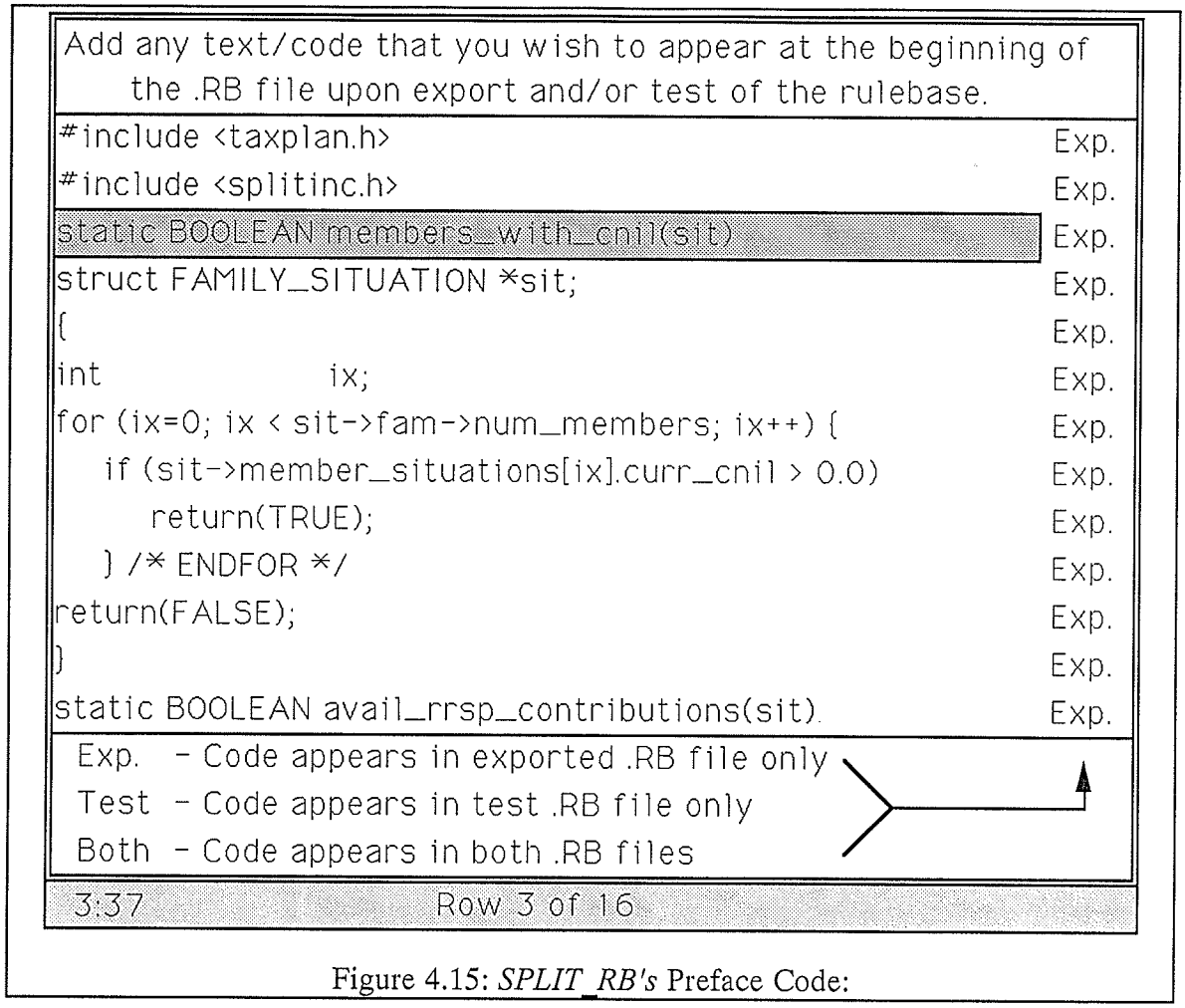

Figure 4.15: *SPLIT_RB's* Preface Code:

It is important to note that once embedded, this rule base does not prompt the user to enter any data. All the data needed by the rule base is already supplied by the application. When testing within KA, the rule base prompts the user for this data. Thus, the user is able to simulate the data that would be supplied by the rest of the application when testing a rule base within KA.

Let us consider that a family has three members, a husband, a wife, and a daughter. Let us then suppose that this family owns a corporation which is projected to make a profit of $100,000.00 in the coming year. This is a case where the income splitting is applicable. When SPLIT_RB was executed for a family with the above characteristics, the ordered set

of tasks shown in Figure 4.16 was produced. The set actually stored integer codes that represented the tasks but textual descriptions are shown here so that one may understand their purpose.

Display Income Splitting Prologue
Clear all family member's corporate income fields
Get information about the participants in income splitting
Explain the details of the corporation
Explain that the corporation is already eligible for the small business deduction
Distribute salaries to the family members
Distribute dividends to the family members
Display the results of the income splitting
Allow the user the choice of accepting the plan or not

Figure 4.16 : Ordered Set of Tasks Produced by *SPLIT_RB*

These tasks are executed by the system in a conventional fashion (using procedural code). The result is that the family situation data structure is modified to reflect the changes contained in the set of tasks. As well, an explanation is built that describes the process to the user. The explanation produced for the family situation described above is shown in Figure 4.17.

Family Income Splitting Explanation

Determining whether corporate profits are eligible for the small business tax deduction (Corporate profits exceeding $200,000.00 are taxed at 38.84 % while profits under $200,000.00 are taxed at 22.84 %) ...

Corporate Income:	$100,000.00
Corporate Expenses:	$0.00
Corporate Salaries:	$0.00
Corporate Profits:	$100,000.00

Since profits do not exceed $200,000.00, they are eligible for the small business tax deduction (tax rate is 22.84 %).

Next, we distribute salaries to eligible members to bring each member's taxable income to the end of the first tax bracket ...

Jack ...	$28,275.00
Jill ...	$28,275.00
Janie ...	$28,275.00

Finally, we distribute dividends to eligible members to bring each member's taxable income to the end of the first tax bracket ...

No dividends distributed.

Figure 4.17 : TAXPLAN Income Splitting Explanation

4.3.2 *FAM_PLAN*:

There are three family-oriented plans currently available in *TAXPLAN*. They are *Manual Planning* which is always applicable and allows the user to make manual modifications to the family's financial data, *Corporation Set Up* which sets up a corporation for a family from self-employed or professional fees income that one or more of the family members may be currently receiving, and *Income Splitting* which distributes income from a family-owned corporation to the members of that family in the most beneficial manner from a tax perspective.

The latter two plans are not always applicable. Thus the rule base *FAM_PLAN* was

created to set appropriate flags and explanations that indicated whether or not these plans are applicable and why they are, or are not, applicable. This rule base consists of only three rules, yet nine combinations of why and why not explanations may be built. How is this done?

First of all, the action of each rule includes code to produce the why explanation for the particular plan. The rules that pertain to *Corporation Set Up* and *Income Splitting* each have two *AND*'ed clause expressions in their premises. This implies that for each rule, three possibilities exist: the rule may be fired, clause one may fail, or clause two may fail. Thus, six possible why and why not explanation combinations should be produced. The rule editor of KA has a feature that allows the user to link C code to each clause of a rule's premise. This C code can access the working memory variables and is executed after the rule base has completed execution but only if the rule and the associated clause failed (i.e. the clause expression evaluated to false). The rule base results structure is used to determine which rules failed and which clauses of their premises caused them to fail. Figure 4.18 shows the three rules and the associated why not code.

```
RULE plan_manual SHORT_CIRCUIT
 IF   TRUE
 THEN
  manual_agenda = TRUE;
  manual_plan_flag = TRUE;
  explain_plan(PLAN_MANUAL , WHY , "This planning option is always
   applicable and allows you to make manual modifications to the client's situation
   while in planning mode.");
 BECAUSE "Manual planning is always applicable."
```

```
RULE plan_incsplit_yes SHORT_CIRCUIT
 IF      corporation_exists == TRUE
  AND corporate_profits >= MINIMUM_PROFITS
 THEN
  incplit_agenda = TRUE;
  buffer = txt_print("Income splitting is applicable because a corporation exists
   and profits are available for distribution (%)." ,
   txt_cvt_double(corporate_profits, "$,.2"));
  explain_plan(PLAN_INCSPLIT , WHY , buffer);
  incsplit_plan_flag = TRUE;
 BECAUSE "We must have an existing corporation with profits to be able to
   perform income splitting."
```

```
explain_plan(PLAN_INCSPLIT , WHY_NOT , "Income splitting is not applicable
because a corporation does not exist. Try using the ESTABLISH CORPORATION
plan to set up a corporation."); 1
```

```
buffer  = txt_print("Income splitting is not applicable because the corporate profits
(%) are too low (less than %)." ,txt_cvt_double(corporate_profits ,"$,.2"),
txt_cvt_double(MINIMUM_PROFITS , "$,.2"));
explain_plan(PLAN_INCSPLIT , WHY_NOT , buffer); 2
```

```
RULE plan_new_corp_yes SHORT_CIRCUIT
 IF      !corporation_exists
  AND elig_fam_mem_inc
 THEN
  new_corp_agenda  = TRUE;
  new_corp_plan_flag = TRUE;
  explain_plan(PLAN_NEW_CORP , WHY , "The client is eligible to establish a
   new corporation because at least one family member has income sources
   eligible for transfer to a corporation.");
 BECAUSE "Determine whether family is eligible to establish a corporation"
```

```
explain_plan(PLAN_NEW_CORP , WHY_NOT , "The system currently allows
only one corporation to be used at any given time and a corporation is currently in
use."); 1
```

```
explain_plan(PLAN_NEW_CORP , WHY_NOT , "A corporation cannot be
established because none of the family members have eligible income sources
(self employed or professional fee incomes)."); 2
```

Figure 4.18: *FAM_PLAN* Rules and Why Not Code

The same result can be accomplished by adding more rules. (Before this feature of KA was available, *FAM_PLAN* contained seven rules.) However, this would only slow the execution of the rule base and add to its complexity. (Obviously this would not be much of a problem for a rule base with three rules, but would greatly add to the complexity of a rule base with one thousand rules.) Figure 4.19 shows an example of the possible plans available to a family with one member. Notice that some plans are flagged as applicable, while others are not. The rules in *FAM_PLAN* are used to set the flags of the family-oriented plans and to generate explanations of the reasoning behind their values. Figure 4.20 shows an example of a *why* explanation for the *Establish a Corporation* plan and a *why not* explanation for the *Family Income Splitting* plan generated by *FAM_PLAN*.

Planning Agenda
for Miller (1990)

Plan	Member	Used	Applicable
Establish a Corporation	* FAMILY *	0	YES
Family Income Splitting	* FAMILY *	0	NO
Manual Planning	* FAMILY *	0	YES
Dividend/Salary Mix	Darren	0	NO
Investment Income Mix	Darren	0	NO
Leveraged Investment	Darren	0	YES

F7: Explain planning operation; F8: Manipulate Plan Agenda
F9: Utilities; F10: Apply plan; ESC: Exit

Figure 4.19 : Plan Agenda Screen From *TAXPLAN*

Family Income Splitting
Income splitting is not applicable because a corporation does not exist. Try using the ESTABLISH CORPORATION plan to set up a corporation

Establish a Corporation
The client is eligible to establish a new corporation because at least one family member has income sources eligible for transfer to a corporation

Figure 4.20 : One *why not* and one *why* explanation generated by *FAM_PLAN*

Like the *SPLIT_RB* rule base, *FAM_PLAN* is embedded into *TAXPLAN* through the use of KA's working memory variable mapping facility. Figure 4.21 illustrates how each field in *FAM_PLAN*'s working memory structure is mapped.

Type	Variable	Exported As
string	buffer	buffer
double	corporate_profits	sit->corp_data.ebt
boolean	elig_fam_mem_inc	eligible_family_income(sit)
boolean	incsplit_agenda	agenda->tasks[agenda_ret...
boolean	incsplit_plan_flag	plan_flags[PLAN_INCSPLIT]
boolean	manual_agenda	agenda->tasks[agenda_ret...
boolean	manual_plan_flag	plan_flags[PLAN_MANUAL]
boolean	new_corp_agenda	agenda->tasks[agenda_ret...
boolean	new_corp_plan_flag	plan_flags[PLAN_NEW_CORP]

Working memory variables added when exporting:
no parameter: char *buffer
parameter: int *plan_flags
parameter: struct AGENDA *agenda
parameter: struct FAMILY_SITUATION *sit

Exported rule base subroutine prototype:
extern void check_family_plans(int *plan_flags, struct AGENDA *agenda, struct FAMILY_SITUATION *sit);

Figure 4.21 : *FAM_PLAN's* Working Memory and Export Data

This chapter is intended to give the reader a feel for ease with which rule bases can be developed with EX-C. It is difficult, however, to provide such a feel without actually using the tool itself. The three sample applications used in this chapter are varied enough to show that EX-C is not designed with only one application or application type in mind.

Chapter Five: Conclusions

The previous chapters of this thesis have talked about various pieces and aspects of the EX-C tools being presented. Let us now finish up with a discussion of some of the conclusions that have been arrived at as a result of the work that is the basis for thesis.

EX-C was designed and constructed from a rule-based programmer's perspective. Thus, its target user is one who understands artificial intelligence and computer technology in general. Because of its relationship to C, EX-C was not designed to be used by non-programmers. Rather, it represents a set of programming tools to be used by the expert and knowledge-based systems developer.

EX-C was developed on top of a third-generation programming language, C, for a purpose. This purpose was to allow both conventional and rule-based programming to be performed within the same environment. Such an integrated environment allows procedural tasks to be coded conventionally. Likewise, more complex tasks may be coded with rules. This represents a more natural fashion in which to develop expert and knowledge-based systems which, most likely, will require both types of tasks. With the emergence of C++ as an object-oriented programming extension to C, it seems desirable that EX-C be integratable with C++. This certainly is a possibility since C++ encompasses all the functionality of C. However, such a project is beyond the scope of this thesis.

EX-C requires little overhead to start up and clean up a call to a rule base. The main reason for this is that rule bases are compiled in EX-C rather than interpreted as in other systems. This allows rule bases to be separated into smaller chunks. This, in turn, provides for easier manageability and quicker execution of rule bases.

One characteristic of most expert systems is their ability to explain themselves. This characteristic provides the user with a consistent knowledge reference and a tool whose results do not have to be accepted on faith. Because of its close relationship with a conventional programming language, EX-C allows more customized explanations to be constructed. Rather than simply showing a the text of a rule (knowledge as it is represented in the computer) as an explanation, EX-C allows the developer to build intelligent explanations that deal with knowledge as the domain expert might understand.

All run-time routines and all executable tools for EX-C were written in C. Having access to this source code allows for EX-C to be extended or modified if needed. This is an important feature of a development tool because the developer of that tool can usually not anticipate all the tasks that it might be applicable to. Thus, EX-C was designed and developed so that it may be extended some time in the future. For example different inference engines may be added to the system that provide more special purpose processing of rule bases.

Finally, EX-C, although it was intended to be and is being used for development, was also intended to be a research project. Its purpose was to examine rule-based programming within the context of C and to discover facilities not offered by other tools. It is not designed to replace other tools completely, but rather augment them and give other tool developers ideas that may be incorporated into their tools. The result of this will hopefully be better tools for the developers of expert and knowledge-based systems.

Appendix A: References

[Amir, 1989] Amir, S., Building Integrated Expert Systems, *AI EXPERT*, January, 1989, pp. 26-37 and March, 1989, pp. 42-52.

[Brown, 1990] Brown, D., Inference Engines for the Mainstream, *AI EXPERT*, February, 1990, pp. 32-37.

[Butler, et al., 1988] Butler, C., Hodil, E., and Richardson, G., Building Knowledge-Based Systems with Procedural Languages, *IEEE EXPERT*, Summer, 1988, pp. 47-59.

[Cohen, 1984] Cohen, B., Merging Expert Systems and Databases, *AI EXPERT*, February, 1989, pp. 22-31.

[Evans, et al., 1990] Evans, M., Miller, D., *Integrating Knowledge-Based Systems and Conventional Computing Environments: An Application Perspective*, In Proceedings of the IASTED International Symposium on Expert Systems Theory & Applications, Los Angeles, U.S.A., December, 1990.

[Firebaugh, 1988] Firebaugh, M., *ARTIFICIAL INTELLIGENCE A Knowledge-Based Approach*, Boyd & Fraser Publishing Company, Boston, 1988, 725 pp.

[Fox, 1990] Fox, M., AI and Expert System Myths, Legends, and Facts, *IEEE EXPERT*, February, 1990, pp. 8-20.

[Franke, 1990] Franke, D., Imbedding Rule Inferencing in Applications, *IEEE EXPERT*, December, 1990, pp. 8-14.

[Freundlich, 1990] Freundlich, Y., Transfer Pricing: Integrating Expert Systems in MIS Environments, *IEEE EXPERT*, February, 1990, pp. 54-62.

[Gervarter, 1987] Gervarter, W., The Nature and Evaluation of Commercial Expert System Building Tools, *Computer*, May, 1987, pp. 24-41.

[Holsapple, et al., 1986] Holsapple, C., Whinston, A., *Manager's Guide To Expert Systems Using GURU*, Dow Jones-Irwin, Homewood, Illinois, 1986, 306 pp.

[Hu, 1989] Hu, D., *C/C++ For Expert Systems*, Management Information Source, Inc., 1989, 556 pp.

[Jackson, 1990] Jackson, P., Introduction to Expert Systems, Addison-Wesley, Don Mills, Ontario, 1990, 515 pp.

[Jakobson, et al., 1986] Jakobson, G., Lafond, C., Nyberg, E., Piatetsky-Shapiro, G., An Intelligent Database Assistant, *IEEE* Expert, Summer, 1986, pp. 65-78.

[Leaman, 1989] Leaman, C., Rule-based Structural Design in C, AI EXPERT, May, 1989, pp. 28-34.

[Li, 1991] Li, C., *Examining the Use of Expert Systems in Managing Fertilizer Application Decisions*, Master's Thesis, University of Manitoba, March, 1991.

[Liebowitz, 1988] Liebowitz, J., *Introduction to Expert Systems*, Mitchell Publishing, Inc., Santa Cruz, California, 1988, 168 pp.

[Lyndon B. Johnson Space Center, 1989] *CLIPS Reference Manual*, Artificial Intelligence Section, Lyndon B. Johnson Space Center, May, 1989, 116 pp.

[Mettrey, 1991] Mettrey, W., A Comparative Evaluation of Expert System Tools, *COMPUTER*, February, 1991, pp. 19-31.

[Neuron Data, 1987] *Nexpert Object User Guide*, Neuron Data, Inc., 1988, Palo Alto, California, 1988.

[Payne, et al., 1990] Payne, E. and McArthur, R., *Developing Expert Systems: A Knowledge Engineer's Handbook For Rules & Objects*, John Wiley, Toronto, 1990, 400 pp.

[Rauch-Hindin, 1988] Rauch-Hinden, W., *A Guide To Commercial Artificial Intelligence*, Prentice Hall, Englewood Cliffs, New Jersey, 1988, 508 pp.

[Rich, 1984] Rich, E., Natural Language Interfaces, *COMPUTER*, September, 1984, pp. 39-47.

[Turban, 1988] Turban, E., *Decision Support and Expert Systems: Managerial Perspectives*, Macmillan Publishing Company, New York, 1988, 697 pp.

[Waterman, 1986] Waterman, D., A Guide to Expert Systems, Addison-Wesley, Don Mills, Ontario, 1986, 420 pp.

Appendix B: Reserved Words for Rule Base Language Constructs

BECAUSE
DECLARE
DECLARE_END
DEFAULT_ENTRY
EXTERNAL
GOAL
IF
INCLUDE
LOCAL
NO_SHORT_CIRCUIT
RB_TABLE
RULE
RULEBASE
SHORT_CIRCUIT
SUBGOAL
TABLE_ENTRY
THEN
USING

Appendix C: Support Routines

ka_display_rule

int ka_display_rule(int mode, ...)

Include	RB_DRULE.H
mode	RB_DRULE_MODE_INIT, RB_DRULE_MODE_DISPLAY, or RB_DRULE_MODE_CLEANUP
...	char *filename for RB_DRULE_MODE_INIT int rule_index for RB_DRULE_MODE_DISPLAY nothing for RB_DRULE_MODE_CLEANUP
Returns	0 if successful, -1 if not.

This routine is used by KA to allow the user to view the entire text of a rule from within the activity display routine.

rb_choose_goal

int rb_choose_goal(char *title)

Include	RULEBASE.H
title	title string
Returns	selected goal id

Allows user to select from a list of the rule base's goals.

rb_cleanup

void rb_cleanup(struct RB_RESULTS *res)

Include	RULEBASE.H
res	forward or backward chainer results structure
Returns	nothing

Free up all dynamic storage allocated for the rule base results.

rb_display_activity

void rb_display_activity(struct RB_RESULTS *res, int mode)

Include	RULEBASE.H
res	forward or backward chainer results structure
mode	one of (RB_PROMPT, RB_ALL, RB_FIRED_ONLY)
Returns	nothing

Builds a display of the activity the rule base processor performed when executing the rule base.

rb_dump_results

void rb_dump_results(struct RB_RESULTS *res)

Include	RULEBASE.H
res	a forward chainer results structure
Returns	nothing

Dumps the elements of the "rule_results" array to the screen using printf's. This routine is meant to be a debugging tool.

rb_exit

void rb_exit(int code)

Include	RULEBASE.H
code	any valid integer
Returns	nothing

Sets the "exit_code" field of the current rule base results structure to "code" and instructs the rule base processor to halt execution.

rb_init_act_rec_ptrs

void rb_init_act_rec_ptrs(void *ptrs[], struct RB_ACT_REC *act_rec)

Include	RULEBASE.H
ptrs	array of generic pointers. Array length is 2.
act_rec	address of RB_ACT_REC structure
Returns	nothing

Initializes ptrs to the addresses of the fields in act_rec for use with generic linked list routines.

rb_process

struct RB_RESULTS *rb_process(int nrules, struct RB_RULE rules[], int solve)

Include	RULEBASE.H
nrules	the number of rules in the rule base
rules	the array of rules in the rule base
solve	id of the goal to solve (ignored for forward chainer)
Returns	the address of the results structure

This routine invokes the applicable rule base processor. A results structure must have been previously set up with "rb_setup_results".

rb_set_goal

void rb_set_goal(int goal_id, int value)

Include	RULEBASE.H
goal_id	the id of the goal to be changed
value	the value to set the goal
Returns	nothing

Sets the passed goal with the passed value.

rb_setup_results

struct RB_RESULTS *rb_setup_results(enum PROCESSING_MODE processing_mode, int ngoals, struct RB_GOAL *goals, void *wm)

Include	RULEBASE.H
processing_mode	RB_FORWARD, RB_BACKWARD
ngoals	the number of goals in the rule base
goals	the array of goals for the rule base
wm	address of working memory structure
Returns	a partially initialized results structure

Allocates a results structure and partially initializes it for rule base execution.

rb_setup_rule_display

void rb_setup_rule_display(int (*routine)(), char *filename)

Include	RULEBASE.H
routine	address of a routine that can be used to print the entire contents of a rule given the ruleid.
filename	Name of a file that contains information about the rule base.
Returns	nothing

This routine changes the way the "rb_display_activity" works. Normally only the BECAUSE clauses of the rules are available for display since the premise and action code is compiled. This routine tells "rb_display_activity" that an entire rule can be displayed using the passed routine. (KA uses this routine when testing rule bases since it keeps a temporary file with all rule information in it - see "ka_display_rule.)

set_add

BOOLEAN set_add(SET *set, <value type of the set> element, int rating)

Include	SET.H
set	an initialized set variable
element	a value with the the type that the set contains
rating	an integer rating to associate with the element to order it within the set
Returns	TRUE if add was successful, FALSE otherwise.

This routine adds an element to the passed set. A rating is also supplied by the calling routine which is used to order the elements of the set. This routine may also be used to modify the rating of an element that is already a member of the set.

set_clear

void set_clear(SET *set)

Include	SET.H
set	an initialized set variable
Returns	nothing

This routine removes all elements from the set. After calling this routine, the set may be reinitialized to a different type or simply reused with its current type.

set_first

BOOLEAN set_first(SET *set)

Include	SET.H
set	an initialized set variable
Returns	TRUE if set has at least 1 element, FALSE otherwise

A current element pointer is maintained for all sets. This routine resets this pointer to point to the first element of the ordered set.

set_in

BOOLEAN set_in(SET *set, <value type of the set> element)

Include	SET.H
set	an initialized set variable
element	a value with the the type that the set contains
Returns	TRUE if element exists in the set, FALSE otherwise.

Used to determine if an element exists in a set.

set_init

void set_init(SET *set, enum TYPE_OF_SET type)

Include	SET.H
set	an uninitialized set variable
type	one of SET_STRING, SET_INT
Returns	nothing

This routine initializes a set to be of a certain set type. No return code is specified as this routine cannot fail. Make sure not to initialize a set that has been previously used already without first clearing that set with "set_clear".

set_last

BOOLEAN set_last(SET *set)

Include	SET.H
set	an initialized set variable
Returns	TRUE if set has at least 1 element, FALSE otherwise

This routine sets the current element pointer of the set to point to the last element in the ordered set.

set_next

BOOLEAN set_next(SET *set)

Include	SET.H
set	an initialized set variable
Returns	TRUE if successful, FALSE if set is empty.

Sets the current element pointer of the set to be the next one after the old current element.

set_prev

BOOLEAN set_prev(SET *set)

Include	SET.H
set	an initialized set variable
Returns	TRUE if successful, FALSE if set is empty.

Sets the current element pointer of the set to be the previous one before the old current element.

set_remove

BOOLEAN set_remove(SET *set,<value type of the set> element)

Include	SET.H
set	an initialized set variable
element	the value you wish to retrieve the rating of
Returns	TRUE if element existed and was removed, FALSE otherwise

This routine may be used to remove a particular element from the passed set.

set_retrieve

BOOLEAN set_retrieve(SET *set,<ptr to variable with type of the set> element)

Include	SET.H
set	an initialized set variable
element	pointer to a variable that can hold a set element
Returns	TRUE if successful, FALSE if set is empty

This routine retrieves the current element of the set and puts it where "element" points to.

set_retrieve_rating

int set_retrieve_rating(SET *set,<value type of the set> element)

Include	SET.H
set	an initialized set variable
element	the value you wish to retrieve the rating of
Returns	rating of element or SET_NOT_FOUND if element not found

Used to retrieve the rating of an element in a set.

Appendix D: Example Rule Base Print Out From KA Tool

Rulebase: check_member_plans (local)

WORKING MEMORY: MEMBER_WM

Variable	Exported As
actual_income	is->actual_income
buffer1	buffer1
buffer2	buffer2
corporate_dividends	is->income_details[INC_CORP_DIV].gross
corporate_salary	is->income_details[INC_CORP_SALARY].gross
corporation_exists	sit->corp_data.corp_exists
div_plan_agenda	agenda->tasks[agenda_retrieve(agenda, PLAN_DIV, memberid)].applicable
div_plan_flag	plan_flags[PLAN_DIV]
investment_income	is->class_income[INCLASS_INVESTMENT].gross
investment_mix_agenda	agenda->tasks[agenda_retrieve(agenda, PLAN_INV_INC, memberid)].applicable
investment_mix_plan_flag	plan_flags[PLAN_INV_INC]
leverage_agenda	agenda->tasks[agenda_retrieve(agenda, PLAN_LEVERAGE, memberid)].applicable
leverage_plan_flag	plan_flags[PLAN_LEVERAGE]

Working memory variables added when exporting:

```
no parm:    char *buffer1
no parm:    char *buffer2
parm:       int  *plan_flags
parm:       int  memberid
parm:       struct AGENDA *agenda
parm:       struct FAMILY_SITUATION *sit
parm:       struct INDIVIDUAL_SITUATION *is
```

```
Rule      plan_div_yes
IF        corporate_dividends != 0.0
  AND     corporate_salary != 0.0
THEN      div_plan_agenda = TRUE
          div_plan_flag   = TRUE
          buffer1 = txt_print("Planning with corporate income allocation is applicable
              because the individual receives both dividends (%)",
              txt_cvt_double(corporate_dividends, "$,.2"));
          buffer2 = txt_print("% and salary (%).", buffer1,
              txt_cvt_double(corporate_salary, "$,.2"));
          explain_plan(PLAN_DIV, WHY, buffer2);
BECAUSE Since we have both corporate salary and dividends, we map perform the
          dividend - salary mix member plan.
```

```
Rule        plan_investment_mix_yes
IF          investment_income > 0.0
THEN        investment_mix_agenda = TRUE
            investment_mix_plan_flag = TRUE
            buffer1 = txt_print("Planning with the investment allocation mix is applicable
                because the individual is receiving investment income (%).",
                txt_cvt_double(investment_income, "$,.2"));
            explain_plan(PLAN_INV_INC, WHY, buffer1);
BECAUSE Since we have investment income, we may execute the investment mix
                individual plan.

Rule        plan_leverage_yes
IF          actual_income > 0.0
THEN        leverage_agenda = TRUE
            leverage_plan_flag = TRUE
            buffer1 = txt_print("The leverage planning option is applicable because the
                individual's actual income (%) is > 0.",txt_cvt_double(actual_income,
                "$,.2"));
            explain_plan(PLAN_LEVERAGE, WHY, buffer1);
BECAUSE We can only perform the leveraged investment plan when a family member has
                actual income.

Rule        plan_div_no_no_divs_and_no_sal
IF          corporation_exists == TRUE
  AND       corporate_salary == 0.0
  AND       corporate_dividends == 0.0
THEN        explain_plan(PLAN_DIV, WHY_NOT, "Planning with corporate income
                allocation is not applicable because the individual does not receive corporate
                salary and dividends");
BECAUSE Corporate exists but the individual does not receive salary OR dividends from it

Rule        plan_div_no_no_corp
IF          corporation_exists == FALSE
THEN        explain_plan(PLAN_DIV, WHY_NOT, "Planning with corporate income
                allocations is not applicable because a corporation has not been established.
                Try using the ESTABLISH CORPORATION plan.");
BECAUSE Corporation does not exist

Rule        plan_div_no_divs
IF          corporation_exists == TRUE
  AND       corporate_dividends == 0.0
  AND       corporate_salary > 0.0
THEN        buffer1 = txt_print("Planning with corporate income allocation requires that the
                individual receives both salary and dividends. In this case the individual
                does not receive dividends, therefore");
            buffer2 = txt_print("% the plan is not applicable");
            explain_plan(PLAN_DIV, WHY_NOT, buffer2);
BECAUSE Individual only receives corporate salary not corporate dividends
```

```
Rule        plan_div_no_sal
IF          corporation_exists == TRUE
  AND       corporate_salary == 0.0
  AND       corporate_dividends > 0.0
THEN        buffer1 = txt_print("Planning with corporate income allocation requires that the
                individual receives both salary and
                dividends. In this case the individual does not receive salary, therefore");
            buffer2 = txt_print("% the plan is not applicable");
            explain_plan(PLAN_DIV, WHY_NOT, buffer2);
BECAUSE Individual only receives corporate salary not corporate dividends

Rule        plan_leverage_no_no_income
IF          actual_income == 0.0
THEN        explain_plan(PLAN_LEVERAGE, WHY_NOT, "Leveraged planning is not
                applicable because the individual's actual income is 0.");
BECAUSE Actual income is 0

Rule        plan_investment_no_no_income
IF          investment_income == 0.0
THEN        explain_plan(PLAN_INV_INC, WHY_NOT, "Planning with the investment
                allocation mix is not applicable because the individual is not receiving any
                investment income.");
BECAUSE investment income is 0
```

Exported rulebase subroutine prototype:

```
extern void check_member_plans(int   *plan_flags, int   memberid, struct AGENDA
    *agenda, struct FAMILY_SITUATION *sit, struct INDIVIDUAL_SITUATION *is);
```