TEST FRAME GENERATION FROM Z SPECIFICATIONS

By

TEGUH RAHARDJO

Bachelor of Engineering

Bandung Institute of Technology

Bandung, Indonesia

1983

Submitted to the Faculty of the
Graduate College of the
Oklahoma State university
in partial fulfillment for
the Requirement for
the Degree of
MASTER OF SCIENCE
July 1995

TEST FRAME GENERATION FROM **Z** SPECIFICATIONS

Thesis Approved:

Mansur Samadzadeh

Thesis Adviser

Blayne E. Mayfield

H. Lu

Thomas C. Collins

Dean of the Graduate College

ACKNOWLEDGMENT

Many thanks to my thesis adviser Dr. Mansur H. Samadzadeh for his advice, guidance, and assistance toward finding my thesis topic finding and the completion of my thesis. My sincere thank to Drs. Blayne E. Mayfield and Huizhu Lu for serving on my graduate committee and to Mr. H. Sinaga for reading an early version of this thesis. I would also like to acknowledge the financial support of the Government of Indonesia through the Science and Technology for Industrial Development Program of the Research and Technology Ministry and through the Computing Center of Nusantara Aircraft Industry Ltd., during my graduate studies.

TABLE OF CONTENTS

LIST OF FIGURES

Figure Page

CHAPTER I

INTRODUCTION

Software testing is the main method generally used to validate the correctness of a program. The testing process accounts for 28% to 50% of the total software development cost [Ramamoorthy75] [Sommerville92]. Studies on test data selection and generation have been conducted to improve the effectiveness and efficiency of the testing process as well as the overall software development process.

Research on the requirement specification and design processes has been conducted for improving the software development process. Informal specification languages were popular for software requirement engineering in the 1970's [Boehm76]. Formal specifications were introduced in the early 1980's and have since become more popular [Basili91]. One of the established formal specification languages is Z. Z was developed at Oxford University [Diller90] [Spivey88].

Studies on specification language based testing techniques are conducted as the specification languages become more established. Studies on software testing that involve the Z language have also been conducted [Hayes86] [Stocks93]. Hayes [Hayes86] proposes abstract data type testing techniques that use data type specifications to produce procedures to check the specification implementations. Stocks and Carrington [Stocks93] describe the derivation of a test template framework from an operation unit

specification. The test templates are constructed from the unit valid input partitions that are heuristically derived from the unit predicates.

A number of test selection techniques are already established. The test selection techniques can be categorized into two main methods: functional method that derive test cases based on the software specification and structural method that derive test cases based on the internal structure of the software [Beizer90] [Myers79]. Cause-effect graphing, one of the functional test selection technique [Myers79], appears to be a promising approach to be implemented for deriving test cases from predicate or Boolean logic based specifications. The technique includes the transformation of a specification into a Boolean cause-effect graph and the derivation of the test cases by tracing the graph backward.

The implementation of the cause-effect graphing to generate test frames from Z specification is the main objective of this thesis. A test frame generation tool that can handle a Z schema with limited notations was developed. The construction of the tool is discussed in this thesis report. The rest of this thesis report also discusses the background of test frame generation and is organized as follows. Chapter II introduces the Z specification technique with a number of simple examples. Chapter III describes the test selection method that will be used for generating test frames. Chapter IV discusses the tool's implementation and evaluation. The last chapter, Chapter V, summarizes this thesis work and describes the possible future work extensions.

CHAPTER II

Z SPECIFICATIONS OVERVIEW

Z is a notation for formal specification and design that uses mathematical disciplines of first-order logic and set theory to model a system [Diller90] [Sommerville92]. Z uses a collection of *schemas* to specify both the static and dynamic aspects of a system [Spivey88].

2.1 Schemas

A schema is a two-dimensional graphical specification that contains a schema's name, *signature*, and *predicate* [Sommerville92]. Figure 1 shows an example of a schema that records a container specification.

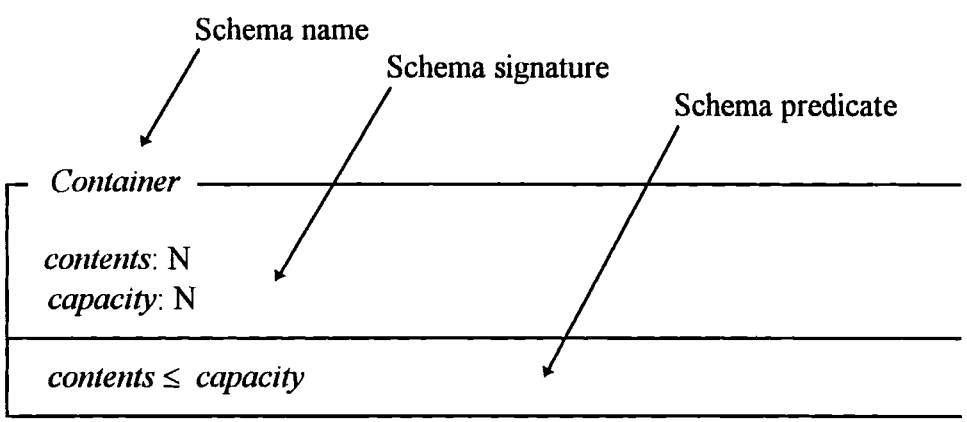

Figure 1. A Z schema for a generic container (Source: [Sommerville92])

The schema name is at the top line of the schema; the schema signature is the part between the top line and the middle line of the schema; the schema predicate is the part between the middle line and the bottom line of the schema. The schema signature contains various declarations that introduce the system entities. The schema predicate specifies the entities' relationships by defining one or more equations and membership predicates over the signature entities.

In Figure 1, the signature of the Container schema introduces two entities, namely *contents* and *capacity*, that are declared as natural numbers. The schema predicate specifies the fact that the contents cannot be greater than the capacity. The predicate is an invariant predicate that must always be TRUE for all operations over the database.

Figure 2 depicts another schema example that specifies a company internal telephone directory. The operations on this directory, which are described in the following sections, broaden the Z specification overview. Most of this background section on Z, including the running example, is based on two main references on Z [Diller90] [Spivey88].

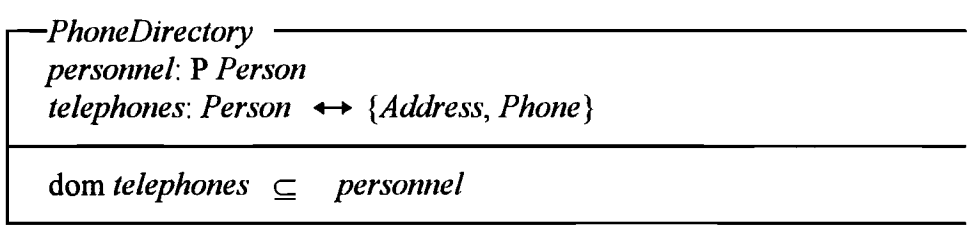

Figure 2. The schema of a telephone directory

The schema *PhoneDirectory* signature in Figure 2 declares two system entities: the set ***personnel*** that consists of all the company employees and the identifier ***telephones***

that gives the relationship between the company personnel and the pairs of the internal addresses and telephone numbers. The schema predicate introduces an invariant predicate that specifies that only the company personnel can have the internal addresses and telephones. The following is an example of a possible state of the system:

personnel = { Asmuni, Mary, John, Teguh}

telephones = { Asmuni → {CC Bldg 100, 4444}, Mary → {Mgmt Bldg 1201, 4001}, John → {Mgmt Bldg. 1701, 4002}, John → {Mgmt Bldg 1701, 4003}, John → {Prod Bldg 1101, 4111}, Teguh → {CC Bldg 100, 4444}}

This simple schema gives the exact system specification that is ordinarily written as a prose specification. The schema specifies that one employee can have more than one telephone and one telephone can be shared by more than one employee. The schema does not impose unwanted limitations on the stored order or the number of telephone entries.

2.2 Schema Decoration and Variable Identifiers

Any operation on a system usually creates a new state of the system. As a convention, the after-operation system schema is represented by decorating the before-operation schema name and variables with a prime [Diller90]. For instance, *PhoneDirectory'* represents the state of the telephone directory after an operation (see Figure 3).

PhoneDirectory′

$personnel' : \mathbb{P}\ Person$
$telephones' : Person \leftrightarrow \{Address, Phone\}$

$\mathrm{dom}\ telephones' \subseteq personnel'$

Figure 3. The after-operation telephone directory schema

The telephone entry addition schema, which is shown in Figure 4, includes the transformation of the system states.

AddEntry

$personnel, personnel' : \mathbb{P}\ Person$
$telephones, telephones' : Person \leftrightarrow \{Address, Phone\}$
$name? : Person$
$address? : Address$
$newnumber? : Phone$

$\mathrm{dom}\ telephones \subseteq personnel$
$\mathrm{dom}\ telephones' \subseteq personnel'$
$name? \in personnel$
$name? \rightarrow \{address?, newnumber?\} \notin telephones$
$telephones' = telephones \cup \{ name? \rightarrow \{address?, newnumber?\}\}$
$personnel' = personnel$

Figure 4. The schema of the telephone directory entry addition

Variables that are ended with a question mark, e.g., *name*? and *newnumber*?, are considered input variables for the operation. In Figure 4, the predicates with the input variables on the left hand side such as

$$name? \in personnel$$
$$name? \rightarrow \{address?, newnumber?\} \notin telephones$$

are the preconditions for the operation, and

$$telephones' = telephones \cup \{ name? \rightarrow \{address?, newnumber?\}\}$$
$$personnel' = personnel$$

are the conducted operations. The other two predicates

$$\text{dom } telephones \subseteq personnel$$
$$\text{dom } telephones' \subseteq personnel'$$

are an invariant precondition and an invariant postcondition, respectively, for the schema *AddEntry* and other operations of the schema *PhoneDirectory*.

A system operation is usually accompanied by an output that reports the completion of operation. Another schema, *Success*, as shown in Figure 5, is added to specify the successful operation information. The exclamation mark at the end of a variable, e.g., *rep*!, indicates that the variable is an output variable.

Success

rep!: *Report*

---

rep! = 'Done'

Figure 5. Successful operation message schema

2.3 The Δ and Ξ Conventions

In an attempt to make a concise specification that includes a state transformation, the Δ (delta) schema is used for representing the combination of the before- and after-operation schemas. For the telephone directory, the Δ schema is shown in Figure 6.

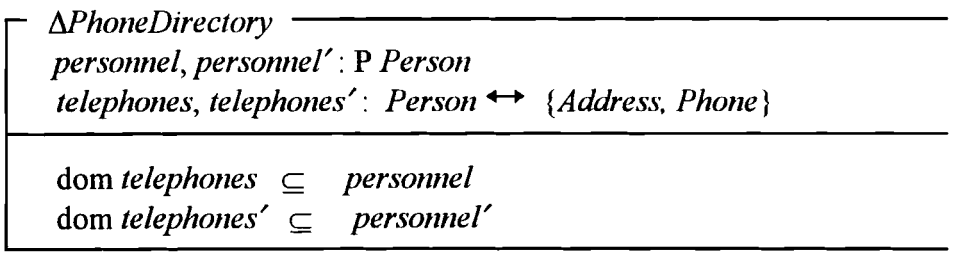

Figure 6. The schema *ΔPhoneDirectory*

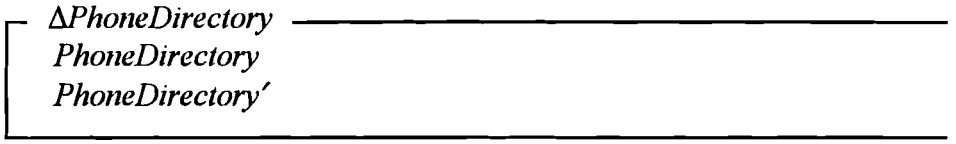

Figure 7. The schema *ΔPhoneDirectory* with schema inclusions

The Δ schema can be represented by using schema inclusions as shown in Figure 7. The schema *AddEntry* also can be written with *ΔPhoneDirectory* inclusion, as depicted in Figure 8.

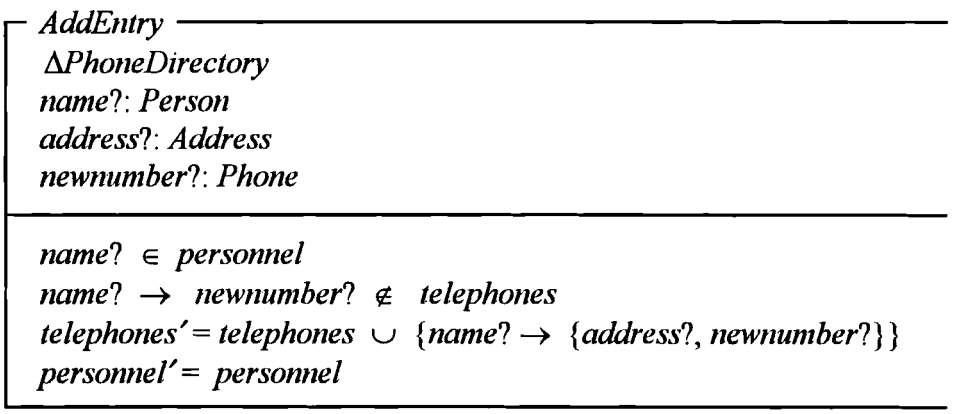

Figure 8. *AddEntry* with *ΔPhoneDirectory* inclusion

Notation Ξ (xi) is used to specify an operation that does not change the system state. The Ξ schema for the schema *PhoneDirectory* is shown in Figure 9. A database

inquiry is an example of an operation that does not change the system state. Figure 10 shows one of the database inquiries of the schema *PhoneDirectory*.

$$\Xi PhoneDirectory$$
$$\Delta PhoneDirectory$$
$$telephones' = telephones$$
$$personnel' = personnel$$

Figure 9. The schema $\Xi PhoneDirectory$

The schemas of entry addition and query operations (discussed above) only cover operations with correct inputs. Since a system might receive incorrect inputs, we must complete the operation specifications by specifying error handling procedures. The next schemas (Figures 11 and 12) specify the operations of entry addition for two kinds of input errors, i.e., when the name entered is not a company employee and when the entry already exists in the directory.

FindPhones
$$\Xi PhoneDirectory$$
$$name?: Person$$
$$numbers!: \mathbb{P}\{Address, Phone\}$$
$$name? \in \mathrm{dom}\ telephones$$
$$numbers! = telephones(|\{name?\}|)$$

Figure 10. *FindPhones* schema

NotEmployee

Ξ*PhoneDirectory*
name?: *Person*
rep!: *Report*

---

name? $\notin$ *personnel*
rep! = 'Not an employee'

Figure 11. The schema of entry addition for the case of a non-employee name

DuplicateEntry

Ξ*PhoneDirectory*
name?: *Person*
address?: *Address*
newnumber?: *Phone*
rep!: *Report*

---

name? $\rightarrow$ {*address*?, *newnumber*?} $\in$ *telephones*
rep! = 'Entry already exists'

Figure 12. The schema of entry addition for the case of a duplicate entry

2.4 Schema Linking

The schemas of parts of a system can be linked together with the propotional connectives $\wedge$ and $\vee$ to form a complete specification for the system. The complete schema for the directory entry addition operation can be defined as follows:

$$CAddEntry = (AddEntry \wedge Success) \vee NotEmployee \vee DuplicateEntry$$

The logical operators $\wedge$ and $\vee$ are used to combine four schemas into one new schema. Figure 13 depicts the expanded schema for the entry addition operation.

CAddEntry

$personnel, personnel' : \mathbb{P}\ Person$
$telephones, telephones' : Person \leftrightarrow \{Address, Phone\}$
$name? : Person$
$address? : Address$
$newnumber? : Phone$
$rep! : Report$

$\mathrm{dom}\ telephones \subseteq personnel$
$\mathrm{dom}\ telephones' \subseteq personnel'$
$((name? \in personnel$
$name? \rightarrow newnumber? \notin telephones$
$telephones' = telephones \cup \{name? \rightarrow \{address?, newnumber?\}\}$
$personnel' = personnel$
$rep! = \text{'Done'})$
$\vee$
$(name? \notin personnel$
$telephones' = telephones$
$personnel' = personnel$
$rep! = \text{'Not an employee'})$
$\vee$
$(name? \rightarrow \{address?, newnumber?\} \in telephones$
$telephones' = telephones$
$personnel' = personnel$
$rep! = \text{'Entry already exists'}))$

Figure 13. The expanded entry addition schema

For the telephone number query operation, a schema for handling input errors should also be added. Figure 14 shows a schema that deals with an unknown person on the query operation. A new schema for telephone number query operations can be created as a combination of the query schemas that cover the correct and the incorrect inputs, i.e.,

$$CFindPhones = (FindPhones \wedge Success) \quad \vee \quad UnknownPerson$$

UnknownPerson

Ξ*PhoneDirectory*
name?: *Person*
rep!: *Report*

---

name? $\notin$ dom *telephones*
rep! = 'Unknown person'

Figure 14. The schema *UnknownPerson*

As discussed above, the use of schemas allows a system specification to be developed gradually and incrementally. Initially, simple schemas for small and manageable pieces of the system can be created. The schemas then can be combined to construct a complete and complex system specification.

CHAPTER III

TEST SELECTION METHOD

The degree of completeness of test cases generated affects the quality of the testing process; i.e., the more complete the test cases, the greater the possibility of finding all the mistakes in a computer program. In most nontrivial cases however, a complete exhaustive test that includes all possible input values consists of too many test cases and is impossible to conduct. For the purpose of minimizing the number of test cases while retaining the effectiveness of testing as much as possible, a minimal subset of the input values that represent the entire input domain should be selected [Myers79] [Rapps85].

There are two main techniques for selecting test cases: functional or black-box methods and structural or white-box methods [Beizer90] [Myers79]. Black-box techniques derive test cases based on the software specification or the external behavior of the software. White-box techniques derive test cases based on the internal structure of the software. One of the techniques that derives test cases from software specifications is the cause-effect graphing method [Myers79].

Cause-effect graphing is a method for selecting test cases that concerns the combinations of program inputs and outputs. The method uses a cause-effect graph that represents a system operation transformation viewpoint. Test cases are selected by methodically tracing back the resulting cause-effect graph [Myers79].

3.1 Cause-Effect Graph

Cause-effect graph is a Boolean graph that links causes (preconditions) and effects (postconditions) of an operation of a system [Myers79]. The causes and effects are represented by nodes, which have either a TRUE(=1) value to indicate that the causes or effects do exist, or a FALSE(=0) value to indicate that the causes or effects do not exist. The logical IDENTITY, NOT, AND, and OR are used to transform or combine the causes and relate them to the effects (see Figure 15). In addition, to state the constraints among the inputs or the outputs, constraint symbols as shown in Figure 16 are used.

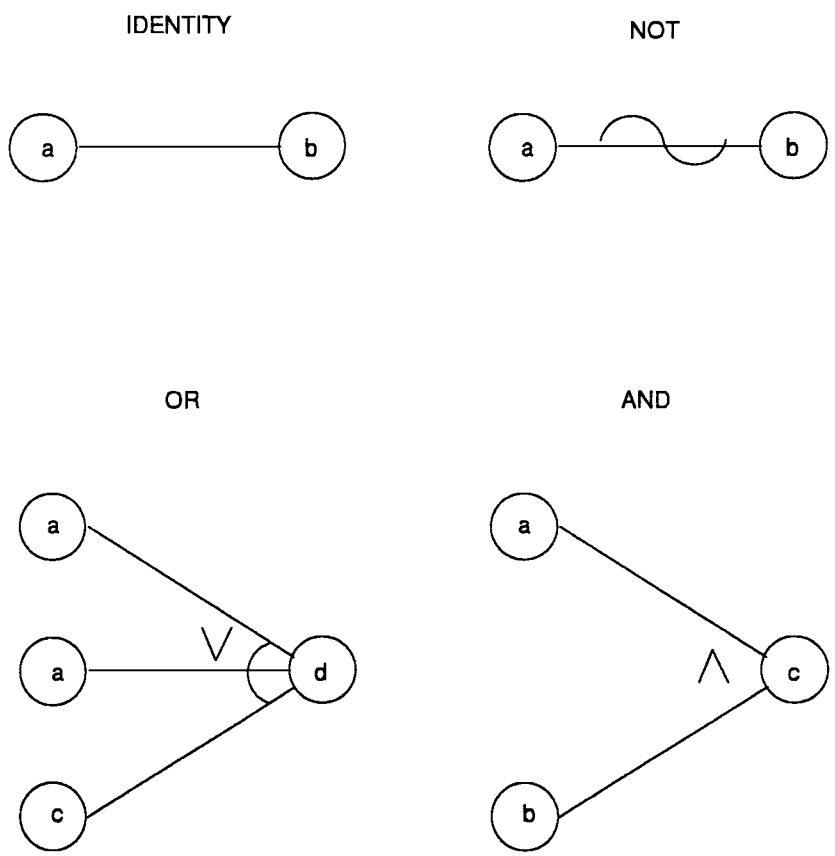

Figure 15. The cause-effect graph basic symbols (Source: [Myers79])

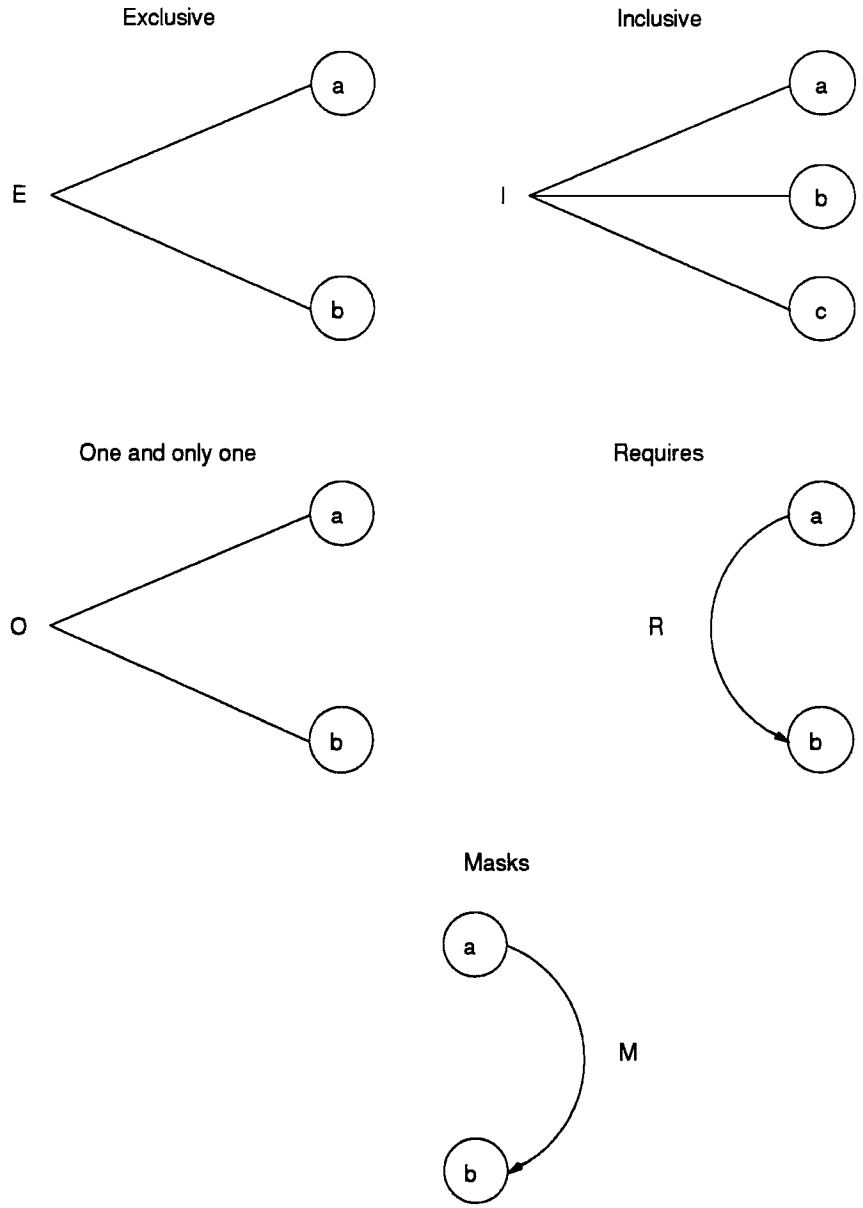

Figure 16. Constraint symbols (Source: [Myers79])

There are four cause constraint symbols (E, I, O, and R) and one effect constraint symbol (M). The E symbol is used to specify that only one of the constrained causes can exist at one time. The I symbol is used to specify that at least one of the constrained causes must exist. The O symbol states that one and only one of the contained causes

must exist. The R symbol states that the existence of one cause requires the existence of the other cause referenced. The M symbol states that the existence of one effect suppresses the existence of another effect referenced [Myers79].

3.2 Test Case Derivation Procedure

Myers provides the main process of cause-effect graphing test case derivation as follows [Myers79].

1. Break down the system specification into smallest independent operational unit specifications. The break down is necessary because a large specification will result in a very complex cause-effect graph.

2. Derive the causes and effects from the unit specification. A cause is an input equivalence class or a diverse input condition, while an effect is a result condition.

3. Construct the Boolean graph of causes and effects based on the unit transformation specifications.

4. Transform the graph into a limited-entry decision table by systematically tracing back the graph. A table row represents a cause or an effect value and a table column represents a test case.

5. Convert the decision table into test cases.

The detailed procedure for decision-table construction proposed by Myers [Myers79] is as follows.

1. Choose one effect to be in the TRUE value.

2. Trace the graph backward from the effect to derive cause combinations that affect the effect value to be TRUE. The rules for reducing the number of cause combinations, when tracing back one node of the graph, are as follows:

   - For an OR node, if the node output is TRUE, consider only cause conditions that lead only one node input to be TRUE, while if the node output is FALSE, consider all possible conditions that lead all the node inputs to be FALSE.

- For an AND node, if the node output is TRUE, consider all possible conditions that lead the output to be TRUE; while if the node output is FALSE, all node input combinations should be considered. For the case that all inputs are FALSE, only one cause situation should be considered. For the other cases, enumerate all cause conditions that lead a node input to be FALSE and consider only one combination of cause that leads a node input to be TRUE.

3. Put the cause and effect values of each cause combination in a column of the table.

4. Derive the value of the other effect nodes for each cause combination and place them on the table.

3.3 Examples of Cause-Effect Graphing Method

As an example of the cause-effect graphing method, a sample specification of a small database based on the specification presented by Myers [Myers79] is used. The database consists of two one-character fields. The specification of the database entry operation is as follows.

> The first field must be an "A" or a "B." The second field must be a digit. If both conditions are satisfied, then a new record is inserted. If the first field is incorrect and the second field is correct, then the message "Field 1 is incorrect" is displayed. If field 1 is correct and field 2 is incorrect, then the message "Field 2 is incorrect" is displayed. If both fields are incorrect, then both messages are displayed.

The derivation of causes and effects of the specification gives the causes as:

c1 - field 1 is "A"

c2 - field 1 is "B"

c3 - field 2 is a digit

and effects as:

e1 - a new record is inserted

e2 - message "Field 1 is incorrect" is displayed

e3 - message "Field 2 is incorrect" is displayed

Figure 17 shows the cause-effect graph of the operation. Intermediate node i1 is created to combine causes c1 and c2 with the OR function. Since causes c1 and c2 cannot exist at the same time, an E constraint is added (see Section 3.1) for the two causes. The decision table constructed by using the method discussed above gives five test frames for the entry operation as shown in Figure 18. By choosing one value for each input equivalence class, the decision table can be converted into test cases as shown in Figure 19.

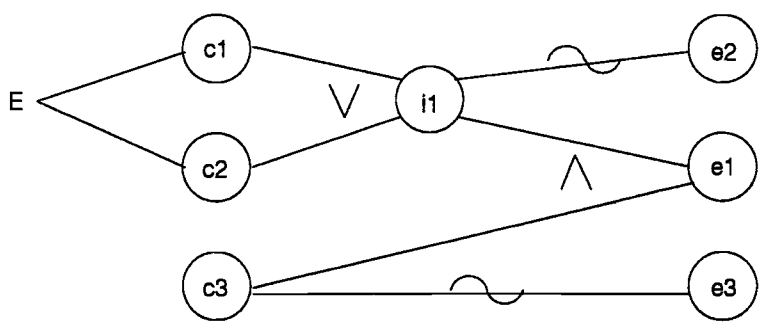

Figure 17. Sample cause-effect graph (Source:[Myers79])

To describe the implementation of the cause-effect graphing method on a test case derivation from a Z specification, the expanded entry addition schema, *CAddEntry*, described in Section 2 is used. The implementation uses specific methods for deriving causes and effects as explained below.

Cause/ Effect	Test Frame Number				
	1	2	3	4	5
c1	1		0	0	1
c2		1	0	0	
c3	1	1	1	0	0
e1	1	1	0	0	0
e2	0	0	1	1	0
e3	0	0	0	1	1

Figure 18. Sample decision table

Test Case	Input or Precondition	Expected Result
1	field 1 = "A" field 2 = "8"	A new record with field 1 = "A" and field 2 = "8" is inserted
2	field 1 = "B" field 2 = "8"	A new record with field 1 = "B" and field 2 = "8" is inserted
3	field 1 = "C" field 2 = "8"	Message " Field 1 is incorrect" is displayed
4	field 1 = "B" field 2 = "C"	Message " Field 2 is incorrect" is displayed
5	field 1 = "C" field 2 = "C"	Messages " Field 1 is incorrect" and "Field 2 is incorrect" are displayed

Figure 19. Sample test cases

The derivation of causes from the schema is accomplished by defining an operation precondition as a cause. The causes derived from *CAddEntry* are listed below.

c1 - dom *telephones* $\subseteq$ *personnel*

c2 - *name*? $\in$ *personnel*

c3 - *name*? $\rightarrow$ {*address*?, *newnumber*?} $\in$ *telephones*

A precondition predicate that forms the negation of another precondition predicate is not defined as a cause. For instance, *name*? $\notin$ *personnel*, the negation of *name*? $\in$ *personnel*, is not defined as a cause.

The effects are derived from the schema *CAddEntry* by defining a postcondition predicate of an operation as an effect. The derived effects are listed below.

e1 - dom *telephones'* $\subseteq$ *personnel'*

e2 - *telephones'* = *telephones* $\cup$ { *name*? $\rightarrow$ {*address*?, *newnumber*?} }

e3 - *personnel'* = *personnel*

e4 - *rep*! = 'Done'

e5 - *telephones'* = *telephones*

e6 - *rep*! = 'Not an employee'

e7 - *rep*! = 'Entry already exists'

By examining the operations and relations in the schema, the cause-effect graph of the schema can be constructed. Figure 20 depicts the cause-effect graph of the entry addition schema. The schema test frames, that are derived from the graph by using the decision-table construction procedure (see Section 3.2), are shown in Figure 21. The possible test cases for the entry addition schema, that are derived form the test frames by choosing one set of cause values for each frame, are shown in Figure 22.

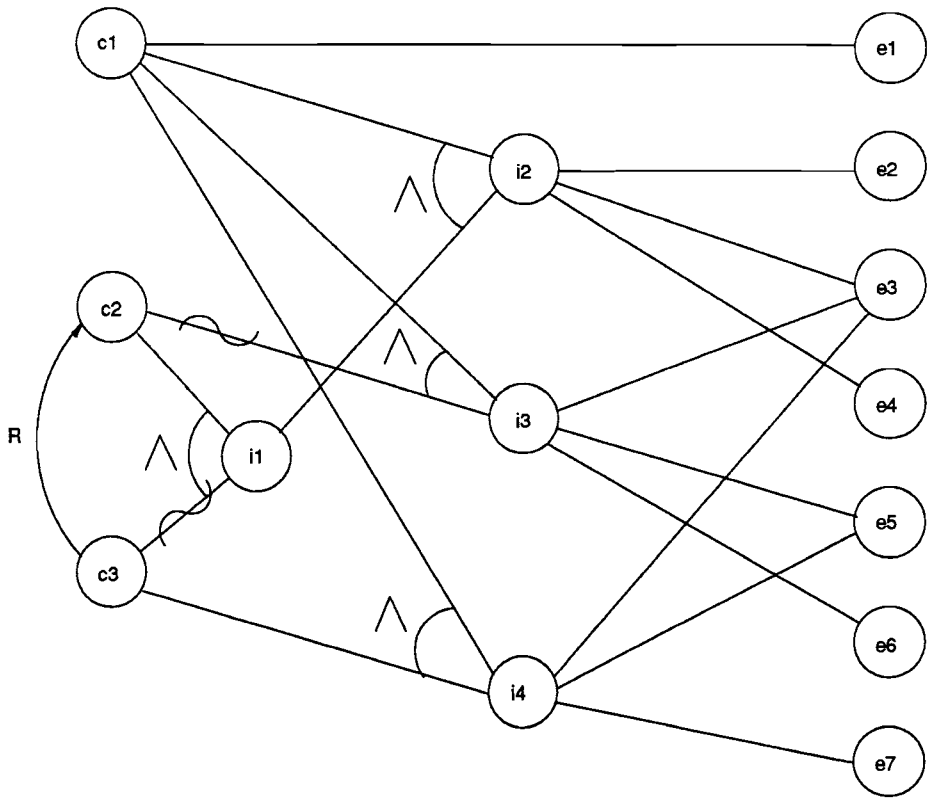

Figure 20. The cause-effect graph of the entry addition schema

Cause/ Effect	Test Frame Number		
	1	2	3
c1	1	1	1
c2	1	0	1
c3	0	0	1
e1	1	1	1
e2	1	0	0
e3	1	1	1
e4	1	0	0
e5	0	1	1
e6	0	1	0
e7	0	0	1

Figure 21. The decision table of the entry addition schema test frames

Test Case	Input or Precondition	Expected Result
1	*personnel* = {Asmuni, Hadi, Mary} *telephones* = { } *name*? = Mary *address*? = Mgmt Bldg 1101 *newnumber*? = 4022	*personnel'* = {Asmuni, Hadi, Mary} *telephones'* = {Mary → {Mgmt Bldg 1101, 4022}} *rep*! = 'Done'
2	*personnel* = {Asmuni, Hadi, Mary} *telephones* = { } *name*? = Tom *address*? = Prod Bldg 1111 *newnumber*? = 4055	*personnel'* = {Asmuni, Hadi, Mary} *telephones'* = { } *rep*! = 'Not an employee'
3	*personnel* = {Asmuni, Hadi, Mary} *telephones* = {Mary → {Mgmt Bldg 1101, 4022}} *name*? = Mary *address*? = Mgmt Bldg 1101 *newnumber*? = 4022	*personnel'* = {Asmuni, Hadi, Mary} *telephones'* = {Mary → {Mgmt Bldg 1101, 4022}} *rep*! = 'Entry already exists'

Figure 22. Test cases for the entry addition schema

In the cause-effect graph, nodes c1 and e1 represent an invariant precondition (cause) and an invariant postcondition (effect), respectively. Node c1 is linked to the cause nodes of all operations of the data entry system. Node e1 is linked solely to node c1 and not affected by the results of all of the schema operations.

In deriving the test frames, the value of the invariant nodes c1 and e1 must always be set to TRUE as a fact that both are invariant nodes. Since the variant effect nodes of an operation are always linked to one intermediate node (the rightmost intermediate node for the operation), test frame derivation can be done by setting the value of the rightmost intermediate node to 1 and tracing the graph backward starting from the node. The value

of the effect nodes then can be derived after all values of the rightmost intermediate nodes have been derived.

CHAPTER IV

IMPLEMENTATION AND EVALUATION

4.1 Introduction

The main thrust of the thesis work involved the development (i.e., design and implementation) of a tool that utilizes the cause-effect graphing method to generate test frames from Z specifications. The tool was developed in the programming language C on a Sequent Symmetry S/81 running the DYNIX/ptx operating system. The implementation uses text files to store the tool input and outputs.

The tool's software consists of sixty eight C procedures, of which forty four procedures are used to create a cause-effect graph and twenty two procedures are used to derive test frames from the cause-effect graph. The input to the tool is the predicate part of the expanded schema of an independent unit of a system. The cause-effect graph traversals and test frame table are the outputs of the tool.

The main data structures and algorithms used to create the graph and to derive the test frames are described later in this chapter. This chapter also describes the tool's input preparations and the evaluation of the tool.

4.2 Input Preparations

Due to the limitations of text files, schema predicates are manually converted to all-text predicates. Certain keywords are used to replace non-ASCII schema symbols. Figure 23 shows the symbol conversion table. In preparing the input, the default link $\wedge$ between two predicates in the same suboperation (that is implicit in a schema) must be explicitly written in the converted specification.

Symbol	Keyword	Symbol	Keyword
$\wedge$	&	$\subset$	subset
$\vee$	|	$\subseteq$	subseteq
$\in$	in	$\not\subset$	notsubset
$\notin$	notin	$\neq$	not=
$\rightarrow$	mapsto	$\cup$	union
'	`		

Figure 23. The non-ASCII symbol conversion table for the tool

As an example, the result of the conversion of the schema predicate of *CAddEntry*, which was described in Chapter II, is shown in Figure 24.

```
dom telephones subseteq members &
dom telephone' subseteq members' &
((name? in members &
name? mapsto newnumber? notin telephones &
telephones' = telephones union name? mapsto newnumber? &
members' = members &
rep! = 'Okay') |
(name? notin members &
telephones' = telephones &
members' = members &
rep! = 'Not a member') |
(name? mapsto newnumber? in telephones &
telephones' = telephones &
members' = members &
rep! = 'Entry already exists'))
```

Figure 24. The conversion of the schema predicate of the schema *CAddEntry*

4.3 Data Structures

4.3.1 Cause-Effect Graph Data Structures

In order to construct a cause-effect graph and to derive test frames from a cause-effect graph, the following requirements are imposed on graph specifications.

- The graph must able to be traced backward and forward.
- Each graph node must be able to be connected with more than one other node backward and forward.
- The cause nodes, constraint nodes, the rightmost intermediate nodes, and the effect nodes must be able to be accessed sequentially.
- All graph nodes should use the same data structure.

Three data structures were used to represent the cause-effect graph: graph header, graph node (vertex), and graph link (edge). The declarations of the three data structures are shown in Figures 25, 26, and 27.

The graph header has a pointer to the first cause, effect, and rightmost intermediate nodes. The header also records the number of cause, intermediate, and effect nodes. The graph node structure is used to represent cause, intermediate, effect, and constraint nodes. Each of the four node types has its own sequence node number.

The node structure has a pointer to the first forward link and the first backward link. A forward link of a cause node connects the cause node to an intermediate node. If a cause node has a constraint relation to other cause nodes, a backward link will connect the cause node to a constraint node.

```
typedef struct graph_header{
  struct graph_node *cause,          /* Pointer to the first cause node */
                    *effect,         /* Pointer to the first effect node */
                    *rightmost_inter,
                                     /* Pointer to the first rightmost
                                        intermediate node  */
                    *constraint;     /* Pointer to the first constraint node */
   int ncause,                       /* The number of cause nodes */
       ninter,                       /* The number of intermediate nodes  */
       neffect;                      /* The number of effect nodes  */
}GRAPH;
```

Figure 25. The declaration of the graph header structure

```
typedef struct graph_node {
  int  type;                    /*  The node type:
                                    1 :  AND intermediate node
                                    2 :  OR intermediate node
                                    3 :  E constraint node
                                    4 :  R constraint node
                                   10 :  Cause node
                                   11 :  Invariant cause node
                                   20 :  Effect node
                                   21 :  Invariant effect node */
  int  number;                  /*  The sequence node number */
  int  scope;                   /*  The node scope level  */
  struct graph_link *forw,      /*  Pointer to the first forward link */
                    *bakw;      /*  Pointer to the first backward link */
  struct graph_node *next;      /*  Pointer to the next sequence similar
                                    node */
} GRPNODE;
```

Figure 26. The declaration of the graph node structure

```
typedef struct graph_link {
  int negation;                 /*  Link negation:
                                    1 : NOT link
                                    0 :  IDENTITY link */
  struct graph_node *node;      /*  Pointer to an adjacent node */
  struct graph_link *next;      /*  Pointer to the next similar function link */
} LINK;
```

Figure 27. The declaration of the graph link structure

A constraint node will only have forward links. For an R constraint node, the first forward link will connect the constraint node to the constrained (affected) cause node, and the next forward links will connect the constraint node to the constraining nodes. For an E constraint node, the sequence of forward links is not important. An effect node will

only use the backward link pointer. The effect node will have backward links to the rightmost intermediate nodes.

The node scope level data in the node structure is used only for an intermediate node. The data is used to arrange the insertion of an intermediate node into a cause-effect graph. An intermediate node can have backward links to the lower level intermediate nodes and/or cause nodes, and forward links to the higher intermediate nodes or effect nodes.

4.3.2 Test Frame Data Structures

The test frame derivation result is recorded by using link lists. Link lists are used since the number of test frames cannot be determined at the beginning of the derivation process and besides the number changes dynamically during the process. The data structures of the test frame link list are: test frame list header and test frame (test frame list element). Figures 28 and 29 show the declaration of the test frame list header structure and the test frame structure, respectively.

One test frame list is used to store the test frame derivation result of the cause-effect graph tracing backward from one intermediate node that is connected to the effect node(s). The list header has a pointer to link the list header and the next test frame list that stores the result of the graph derivation from the next rightmost intermediate node.

The test frame structure has a pointer to the array of the intermediate node derivation status. The status array element is used to indicate whether: a. all possible input

conditions, or b. only one input condition, that leads the value of an intermediate node in a test frame, is needed to be derived.

```
typedef struct test_frame_list_header {
   struct test_frame *head;          /* Pointer to the first test frame */
   struct test_frame *tail;          /* Pointer to the last test frame */
   struct test_frame_list_header *next;
                                     /* Pointer to the next test frame list
                                        header */
} TEST_LIST;
```

Figure 28. The declaration of the test frame list header structure

```
typedef struct test_frame {
   int  *cause;                      /* Pointer to the array of cause node
                                        values */
   int  *inter;                      /* Pointer to the array of intermediate
                                        node values */
   int  *derive;                     /* Pointer to the array of the intermediate
                                        node derivation status */
   int  *effect;                     /* Pointer to the array of effect node
                                        values */
   struct test_frame *next;          /* Pointer to the next test frame */
} TEST_FRAME;
```

Figure 29. The declaration of the test frame structure

4.3.3 Working Data Structures

The tool uses several working data structures for constructing a cause-effect graph and deriving test frames from a cause-effect graph. Two important working data structures are the "predicate" and "operator" structures. These two structures are used in

a cause-effect graph construction to store a scanned predicate and the operator following the predicate. The declaration of the structure predicate is shown in Figure 30 and the declaration of the structure operator is shown in Figure 31.

```
typedef struct predicate {
  char part[3][80];        /* Predicate parts:
                              - Part 1: An entity before an equation or
                                        membership symbol in a predicate
                              - Part 2: An equation or membership symbol,
                              - Part 3: An entity after an equation or
                                        membership symbol in a predicate. */
  int type,                /* Predicate type:
                              10 : Precondition/cause predicate
                              11 : Invariant precondition predicate
                              20 : Postcondition/effect predicate
                              21 : Invariant postcondition predicate */
      scope;               /* Scope level of the predicate */
} PRED;
```

Figure 30. The declaration of the predicate data structure

```
typedef struct operator {
  int type;                /* The operator type:
                              0 : No operator; 1 : AND ; 2 : OR    */
      scope;               /* Scope level of the operator */
} OPER;
```

Figure 31. The declaration of the operator data structure

Both the predicate and the operator contain an element to record their scope level, which is used as one of the parameters to arrange the representation of the precondition predicate relationships in the cause-effect graph. A cause node that represents the next precondition predicate must be linked (through intermediate nodes) to a cause node that

represent the previous precondition predicate if the next predicate has a lower scope level than the previous predicate.

4.4 Algorithms

In the two sections that follow, the tool's algorithms for constructing a cause-effect graph and for deriving test frames from the graph are presented.

4.4.1 Cause-Effect Graph Construction Algorithms

The tool implements the cause-effect graph construction procedures described in Chapter III, except for constraint relations I (Inclusive), O (One and only one), and M (Masks). The following two algorithms are the main algorithms to construct a cause-effect graph from a converted schema predicate.

Algorithm 1 Cause-Effect Graph Construction

Input: Input schema file.

Output: The constructed cause-effect graph, and the table of precondition (cause) and post condition (effect) predicates.

Method:

1. Set the initial value for the predicate and the operator scope level.
2. **while** there is a predicate in the input schema file
   then begin
   Scan a predicate and a connective operator that follows the predicate from the input file schema; In this scanning, decrease the scope level of the predicate and the operator by 1 when a character '(' is scanned and increase the scope level of the predicate and the operator by one when a character ')' is scanned.
   Look the scanned predicate up in the table of predicates and get the predicate sequence number and the predicate negation flag
3. **if** the scanned predicate or the negation of the predicate does not exist in the table of predicates
   then begin
   Insert the predicate into the table and get the predicate sequence number predicate and negation flag

4. **if** the scanned predicate is an effect or invariant effect predicate
   then Add a new effect node to the cause-effect graph and link the node to the related nodes
5. **else begin** /* The scanned predicate is a cause predicate */
   Add a new cause node to the cause-effect graph and link the node to the related nodes
6. **if** the number of the graph cause nodes is more than one and the predicate is not an invariant cause predicate
   then begin
   Search constraints between the new predicate and the existing cause predicates
7. **if** the constraints are found
   then Add the necessary constraint nodes and links to the graph
   end
   end
   end
8. **else**
   Add necessary links between the existing predicate node and the related nodes in the graph

end while

9. Remove any intermediate node duplication

Algorithm 2 Cause/Effect Node and Link Addition

Input: The cause-effect graph, the scanned predicate, the predicate sequence number, the predicate negation flag, the scanned connective operator, and a node addition flag.

Output: Updated cause-effect graph.

Method:

1. **if** the scanned predicate is the negation of the existing predicate in the table of predicates
   then Set the graph link negation to 1.
2. **if** a new cause/effect node is required
   then begin
   Create a new graph node.
3. **if** the predicate is a cause or invariant cause predicate
   then Add the new node to the graph-cause link list.
4. **else** Add the new node to the graph-effect link list.
   end
5. **else** /* A new node is not required */
   Search the appropriate existing cause/effect node to be linked.
6. **if** the type of predicate is invariant cause **or** (the type of the predicate is cause **and** there is a connective operator that follows the predicate **and** (the node is the first cause node **or** the type of the current connective operator is not the same as the previous operator **or** the type of the predicate is effect or invariant effect **or** the scope level of the predicate is less than the scope level of the previous operator))
   then begin /* A new intermediate node is required */
   Create a new intermediate node; Set the scope level of the node to value of the operator scope level.

```
7.          if the graph rightmost intermediate link list is NULL or the type of the previous predicate is
              effect or invariant effect
            then Set the created intermediate node as the new rightmost intermediate node
        end
8.  if the type of the predicate is cause or invariant cause
    then begin      /* Link a cause node */
9.          if the previous predicate is a cause predicate and the new intermediate node is created
            then begin
10.                 if the scope level of the predicate is the same as the scope level of the previous
                      intermediate node and the type of the previous intermediate node is AND
                    then begin
                            Link the cause node with the previous intermediate AND node; Link the
                            new intermediate node and the other intermediate nodes based on their
                            scope level and update the rightmost intermediate node pointer if necessary.
                         end
11.                 else begin
                            Link the cause node with the new intermediate node; Link the new
                            intermediate node with the previous intermediate nodes (the new node is
                            the backward node of the previous node).
                         end
                end
            else        /* The new intermediate node is not created */
                Link the previous intermediate node and the cause node.
        end
12. else begin      /* Link an effect node */
        Link the rightmost intermediate node and the effect node.
13.     if the type of the previous predicate is cause or invariant cause
        then begin
                Add the rightmost intermediate node to the graph rightmost intermediate node list;
                Remove the previous intermediate node if the node is not linked to an effect node and
                only has one backward link and one forward link after the node backward link is
                copied to the forward node; Add a new element (pointing to the rightmost
                intermediate node) to the rightmost intermediate scope list.
             end
    end
14. if the predicate is the last predicate in the input schema file
    then Link a rightmost intermediate node with another rightmost intermediate node based on the node
         sequences pointed by the rightmost intermediate scope list; The later node is linked to the earlier
         node if the scope level of the earlier node is greater than the later node scope level; In this
         linking, the forward links of the later node are removed after they are copies to the earlier node.
```

4.4.2 Test Frame Derivation Algorithms

The general procedures for deriving test cases from a cause-effect graph was presented in Chapter III. The tool implements almost all of the procedures except the derivation of test cases from test frames. In this implementation, the tracing is not started

from an effect node but from one of the graph's rightmost intermediate node, as mentioned at the end of Chapter III. The implementation procedures are described in the following seven algorithms.

Algorithm 3 Test Frame Derivation

Input: A cause-effect graph and the value of the starting derivation node (one of the graph rightmost intermediate nodes).

Output: Test frame lists.

Method:

1. Create the header of the test frame lists.
2. **for** all of the rightmost intermediate nodes that do not connected to an INV_EFFECT node
   begin
   Trace the graph backward starting from the rightmost intermediate node.
3. **if** the test frame header is not linked to a test frame list
   then Link the test frame lists header to the new test frame list creating in the graph tracing.
4. **else** Link the previous test frame list to the new test frame list creating in the graph tracing.
5. **for** all test frame in the new test frame list
   begin
6. **for** all of the other rightmost intermediate nodes
   begin
7. **if** the other rightmost intermediate node is connected to an INV_EFFECT node
   then Set the other rightmost intermediate node in the test frame to 1.
   else Derive the other rightmost intermediate node value by (a) propagating the existing node values or, if the value propagation cannot used to derive the other rightmost intermediate node, (b) setting the other rightmost intermediate node value in the test frame to 0 and derive the unassigned affected node values.
   end
8. **end**
9. Remove any test frame duplication in all test frame lists.
10. Derive all node effect values in all test frames of the new test frame list by propagating the value of the rightmost intermediate nodes .

end

Algorithm 4 Cause-Effect Graph Backward Tracing Starting from One of the Graph Rightmost Intermediate Nodes

Input: The cause-effect graph, one of the graph rightmost intermediate nodes, and the rightmost intermediate node setting value .

Output: A new test frame list.

Method:

1. Create a new test frame list.
2. Create a new test frame and add the new test frame to the new test frame list.
3. Set the value of the rightmost intermediate node in the new test frame to the setting value (=1).
4. Starting from the rightmost intermediate value, recursively trace the graph node backward to derive the possible combinations of the input node values.

Algorithm 5 Recursive Graph Node Backward Tracing

Input: A cause-effect graph, one of the graph intermediate node, and a test frame list.

Output: Updated test frame list.

Method:

```
1. if the node type is AND or OR and the test frame list is not NULL
   then begin
        Derive the input node values                          /* Algorithm 6 */
2.      for all input nodes of the node
            Trace the graph node backward                     /* Algorithm 5 */
    end
```

Algorithm 6 Input Node Values Derivation

Input: A cause-effect graph, an intermediate node of the graph, and a test frame list.

Output: Updated test frame list.

Method:

```
1. for all test frames in the input test frame list
   begin
2.    if the explosion of the test frame at this node is allowed
      then begin
           Create an additional test frame list
3.         if the node output value at the test frame is TRUE
           then begin
4.              if the node is an AND intermediate node
5.              then for all input nodes of the intermediate node and the previous input node
                         value derivation, if any, is succeeded
                         Derive the input node value by setting the value of the input of the
                         intermediate node to be TRUE                    /* Algorithm 7 */
6.              else     /* The node is an OR intermediate node */
7.                   Derive the true OR input node values               /* Algorithm 8 */
                end
8.         else      /* The node output is FALSE */
9.              if the node is an AND intermediate node
10.             then Derive the false AND input values                  /* Algorithm 9 */
11.             else    /* The node is an OR intermediate node */
                     for all input nodes of the intermediate node and the previous input node value
                         derivation, if any, is succeeded
```

```
                    Derive the input node value by setting the value of the input of the
                    intermediate node to be TRUE                    /* Algorithm 7 */
12.         if the input node derivation has not successfully completed
            then Remove the test frame from the list
    end
13. if the additional test frame list is not empty
    then Add all elements of the additional test frame list to the input test frame list.
```

Algorithm 7 One Input Node Value Derivation

Input: A cause-effect graph, a test frame, one of an intermediate node backward link, and the value of the intermediate node input to be propagated to the input node value.

Output: Updated test frame and return code.

Method:

```
1.  if the type of the input node is invariant cause
    then begin
2.          if the value of the input node in the test frame has not been assigned
            then begin
                    Set the value of the input node in the test frame to 1
3.                  if the intermediate node propagation value is not 1
                    then Terminate and return an abnormal termination code
                end
             Terminate and return a normal termination code
        end
4.  else if the type of the input node is cause
5.      then if the value of the input node in the test frame has not been assigned
            then begin
6.                  if the input node has constraint relations
                    then begin
                            Examine whether the propagation of the intermediate node propagation
                            value to the constrained node does not conflict with any existing value of
                            the constraint related nodes
7.                          if there is a conflict
                            then begin
                                    Set the value of the input node in the test frame to 0.
                                    Terminate and return an abnormal termination code
                                end
8.                            else begin
                                    Set the value of the input node in the test frame to the value of
                                    the intermediate node propagation value;
                                    Set the value of the constrained nodes to the value of the node
                                    propagation value in the test frame, if the value of the
                                    constrained nodes have not been assigned;
                                    Terminate and return a normal termination code.
                                end
                        end
9.                  else Set the value of input node in the test frame to the intermediate node
                        propagation value
```

```
                end
10.         else if the intermediate node propagation value is not the same as the existing value of the
                    input node in the test frame
                then Terminate and return an abnormal termination code
11.             else Terminate and return a normal termination code
12.     else    /* The type of the input node is intermediate */
            Set the value of input node in the test frame to the intermediate node propagation value
```

Algorithm 8 True OR Input Node Values Derivation

Input: A cause-effect graph, an OR intermediate node of the graph, a test frame, and an additional test frame list.

Output: Updated test frame, updated additional test frame list, and return code.

Method:

1. Determine the true OR input value combinations
2. **for** all of the combinations but the last one
   begin
      Create a new test frame
      Copy the values in the input test frame to the new test fame
      Derive the value of the input nodes and store them in the temporary test frame
3.   **if** there is a value conflict between the existing value and the propagation value during the derivation
4.   **then** Delete the new test frame
5.   **else** Add the new test frame to the additional test frame list
   end
6. Derive the value of the input nodes using the last value combination and store them in the input test frame
7. **if** there is a value conflict during the last derivation
   then Terminate and return an abnormal termination code
8. **else** Terminate and return a normal termination code

Algorithm 9 False AND Input Node Values Derivation

Input: A cause-effect graph, an False intermediate node of the graph, a test frame, and an additional test frame list.

Output: Updated test frame, updated additional test frame list, and return code.

Method:

1. Determine the false AND input value combinations
2. **for** all of the combinations but the last one
   begin
      Create a new test frame
      Copy the values in the input test frame to the new test fame
      Derive the value of the input nodes and store them in the temporary test frame
3.   **if** there is a value conflict between the existing value and the propagation value during the

derivation
4. **then** Delete the new test frame
5. **else** Add the new test frame to the additional test frame list
end
6. Derive the value of the input nodes using the last value combination and store them in the input test frame
7. **if** there is a value conflict during the last derivation
then Terminate and return an abnormal termination code
8. **else** Terminate and return a normal termination code

4.5 Complexity

The computational complexity of the tool is determined by calculating the execution time and the space usage of the graph construction and test frame derivation separately. Because of the complicated data structures and the recursive algorithms used, for both calculations only the worst cases are considered.

The graph construction execution time and space usage depend on the number of nodes and links that are created. The worst case is found when a cause or effect node is created for each scanned predicate, each intermediate node has two backward links, and there are constraints R among the cause nodes as shown in Figure 32.

For n scanned predicates, the resulting number of graph nodes and links are given below.

- The number of cause, intermediate, and effect nodes = 2 n,

- The number of non-constraint links = 2 n - 1,

- The number of constraint nodes = n - 2, and

- The number of constraint links = (2+3+ ....+ (n-1)) = $\frac{1}{2}$n(n - 1) - 1.

These numbers show that the worst case computational complexity of a cause-effect graph creation process is the square of the number of scanned predicates.

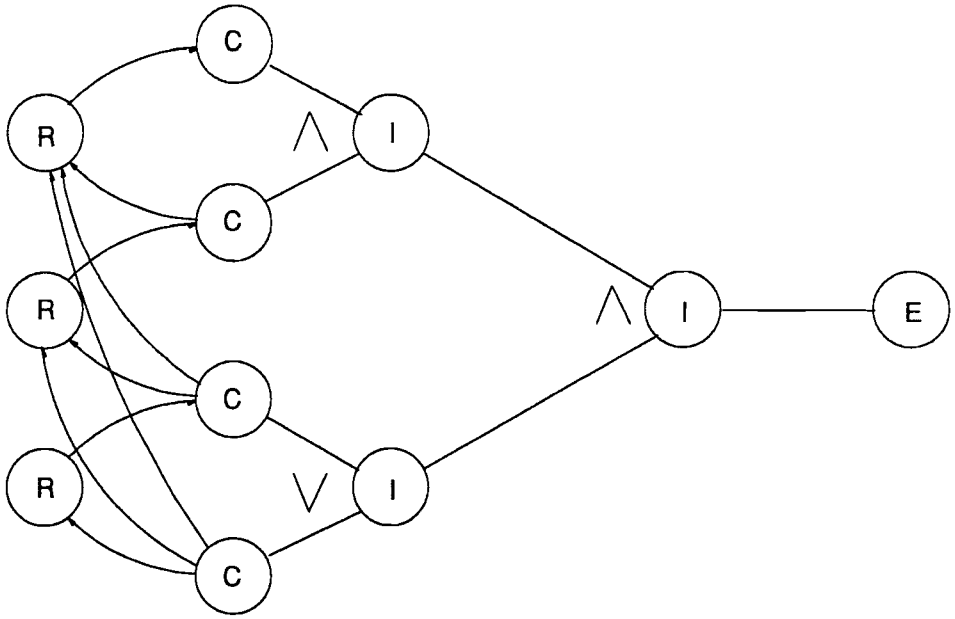

Figure 32. The worst case for the cause-effect graph construction for five scanned predicates

The execution time for the derivation of test frames from a cause-effect graph depends on the number of graph node value assignments, which is proportional to the product of the number of test frames and the number of graph nodes. The space usage for the derivation is also proportional to the product of the number of test frames number and the number of nodes in the cause-effect graph. The worst case for the test frame derivation is found when each scanned predicate is transformed to a cause or effect node, all cause nodes are connected to an AND intermediate node, and the intermediate node value is the negation of an effect value.

The structure of cause-effect graphs resulting in the worst complexity for five scanned predicates is depicted in Figure 33. The number of test frames that will be created during the derivation in the worst case grows proportional to two to the power of

the number of scanned predicates. Thus, the execution time and space usage of the tool is proportional to the product of the number of scanned predicates and two to the power of the number of scanned predicates.

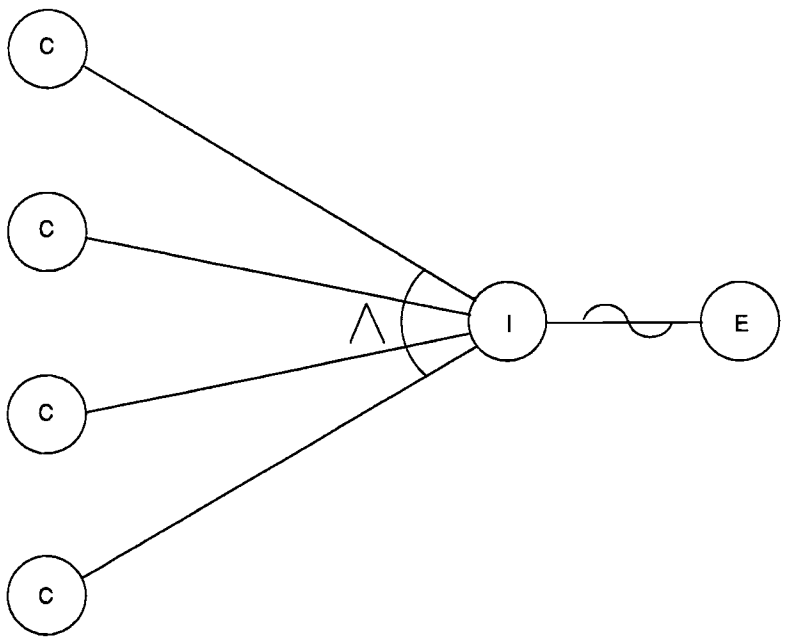

Figure 33. The worst case for test frame derivation for five scanned predicates

4.6 Testing of the Tool

The tool has demonstrated the feasibility of generating test frames from Z specification. Two sample schemas were used to test the tool. The test results (included in Appendix C) showed the capability of the tool in transforming a Z schema to a cause-effect graph and deriving test frames from the graph. The graph creation part of the tool can handle the R and E constraint relations among the schema predicates and can transform them into graph node relations. The test frame derivation part of the tool,

which is based on the cause-effect graphing method, has been done successfully both for regular node derivations and all zeroes input of a false AND intermediate node.

CHAPTER V

SUMMARY AND FUTURE WORK

5.1 Summary

The role of software testing in software development is quite important and formal specifications becoming more widely used. Consequently, conducting studies on test generation from formal specifications is a viable and active area of work. The work that was conducted in this thesis was the development of a tool to generate test frames from Z specifications. In this study, a tool was designed and implemented to be used for limited Z schema notations. The tool implements the cause-effect graphing method for generating test frames [Myers79].

The tool carries out two major processes. The first process converts a Z schema to a cause-effect graph. The tool input is the predicate part of the schema with limited notations. The input should be converted manually to an all-text specification before the processing can begin. The constraint relations among predicates that can be handled by the tool are constraint R's (requires) and E's (exclusive). The second process derives test frames by tracing the cause-effect graph backward. At this process, the tool precisely implements Myers' method [Myers79].

Two sample Z schemas were used to test the tool. The test frames were generated correctly based on the method used. Despite the stated weaknesses and limitations of the tools, the test results demonstrate the promising possibility for generating test data from Z specifications.

5.2 Future Work

A prototype evaluation of the tool indicated a number of weaknesses.

- The tool cannot read a Z schema directly; the schema should be converted into an all-text specification.
- The tool can handle limited Z notations (e.g., it cannot handle universal and existential quantifiers).
- The tool does not provide input syntax checking.
- The tool cannot handle I, O, or M constraint relations among schema predicates.

The above list shows that improvements are needed mainly at the front-end of the tool. The back-end of the tool will need small changes after the tool's front-end is improved.

Designing, implementing, and testing improvement to the tool to eliminate the weaknesses should be considered as extensions of this work. They include the creation of a graphical environment for editing Z schemas, the development of a better scanner, and the addition of a parser for syntax checking. The generation of test cases from test frames can also be a significant work to improve the tool's capability.

REFERENCES

[Basili91] Victor R. Basili and John D. Musa, "The Future Engineering of Software: A Management Perspective," *IEEE Computer*, pp. 90-96, September 1991.

[Beizer90] Boris Beizer, *Software Testing Techniques*, Van Nostrand Reinhold, New York, NY, 1990.

[Boehm76] Barry W. Boehm, "Software Engineering," *IEEE Transactions on Computers*, Vol. C-25, No. 12, pp. 1226-1241, December 1976.

[Denney91] Richard Denney, "Test-Case Generation from Prolog-Based Specifications," *IEEE Software*, pp. 49-57, March 1991.

[Diller90] Antoni Diller, *Z: An Introduction to Formal Methods*, John Wiley & Sons, Chichester, England, 1990.

[Foster84] K. A. Foster, "Sensistive Dest Data for Logic Expresssion," *ACM SIGSOFT Software Engineering Notes*, Vol. 9, No. 2, pp. 120-126, April 1984.

[Hayes86] Ian J. Hayes, "Specification Directed Module Testing," *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 1, pp. 124-133, January 1986.

[Higashino94] T. Higashino and G. v. Bochmann, "Automatic Analysis and Test Case Derivation for a Restricted Class of LOTOS Expresssions with Data Parameters," *IEEE Transactions on Software Engineering*, Vol. 20, No. 1, pp. 29-42, January 1994.

[Jacky95] Jonathan Jacky, "Specifying a Safety-Critical Control System in Z," *IEEE Transactions on Software Engineering*, Vol. 21, No. 2, pp. 99-106, February 1995.

[Luo94] G. Luo, A. Das, and G. v. Bochmann, "Software Testing Based on SDL Specifications with Save," *IEEE Transactions on Software Engineering*, Vol. 20, No. 1, pp. 72-87, January 1994.

[Myers79] Glenford J. Myers, *The Art of Software Testing*, John Wiley & Sons, New York, NY, 1979.

[Ostrand88] Thomas J. Ostrand and Marc J. Balcer, "The Category-Partition Method for Specifying and Generating Functional Tests," *Communications of the ACM*, Vol. 31, No. 6, pp. 676-686, June 1988.

[Ramamoothy75] C. V. Ramamoorthy and Siu-Bun F. Ho, "Testing Large Software with Automated Software Evaluation Systems," *IEEE Transactions on Software Engineering*, Vol. SE-1, No. 1, pp. 46-58, March 1975.

[Rapps85] Sandra Rapps and E. Weyuker, "Selecting Software Test Data Using Data Flow Information," *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 4, pp. 367-375, April 1985.

[Sommerville92] Ian Sommerville, *Software Engineering*, Addison-Wesley, Workingham, England, 1992.

[Spivey88] J. M. Spivey, *Understanding Z*, Cambridge University Press, Cambridge, England, 1988.

[Stocks93] Phil Stocks and David Carrington, "Test Template Framework: A Specification -Based Testing Case Study," *Proceedings of the 1993 International Symposium on Software Testing and Analysis (ISSTA)*, Cambridge, MA, pp. 11-18, June 1993.

[Weyuker94] E. Weyuker, T. Goradia, and A. Singh, "Automatically Generating Test Data from a Boolean Specification," *IEEE Transactions on Software Engineering*, Vol. 20, No. 5, pp. 353-363, May 1994.

APPENDIXES

APPENDIX A

GLOSSARY

Abstract Data Type:	A set of values (a domain) and a set of operations on those values.
Black-Box Techniques:	Software test case selection techniques that derive test cases based on a software specification or the external behavior of the software.
Cause:	An input equivalence class or a diverse input condition.
Cause-Effect Graph:	A graph that links causes and effects.
Decision Table:	A table in which each column specifies the causes (conditions) under which the effects (actions) will take place.
Effect:	An output condition or a system transformation.
Exhaustive Test:	A test that uses all possible input values.
Formal Method:	The use of mathematical notations, such as first-order logic or set theory, to describe system specifications and software designs together with the techniques of validation and verification.
Formal Specification:	A system specification described by using a formal method.
Invariant Preconditions:	Preconditions that always be true and are valid for all operations of a system.
Invariant Postconditions:	Postconditions that always be true and are valid for all operations of a system.
Postconditions:	Conditions that hold after an operation of a system is executed.

Preconditions:	Conditions that hold before an operation of a system is executed.
Predicates:	Statements that can be either true or false.
Schema:	A two-dimensional graphical specification in Z, consisting three components: name, signature, and predicate.
Schema Predicate:	A schema component that specifies relationships by defining equations and membership predicates over the entities that are defined in the schema signature.
Schema Signature:	A schema component that contains various declarations that introduce system entities.
Test Case:	A combination of specific input and output values derived from each test frame to test the correctness of a software module or system.
Test Frame:	A combination of causes (input equivalence classes or a diverse input condition) and effects (expected results) as a frame for developing test data.
White-Box Techniques:	Software test case selection techniques that derive test cases based on the internal structure of a software module or system. Also known as Glass-Box Techniques.
Z:	A notation for formal specifications and designs that use the mathematical disciplines of first-order logic and set theory to model a system.

APPENDIX B

INPUT/OUTPUT LISTINGS

Two different inputs have been used to test the tool. The first input is the converted schema of the *CAddEntry* schema as described in Chapter II. The second input is a converted schema of a birthday data entry schema, *AddBirthRecord*, that is shown in Figure 34 on the next page. Each input is stored in file "dtest" before the tool is run.

The tool output shows the input specification (the converted schema), the traversals of the constructed cause-effect graph, the test frames, and the number of computation for deriving the test frame. The tool outputs for the two cases are shown the next pages following the *AddBirthRecord* schema. An illustration of the cause-effect graph, which was drawn from the second case graph traversal outputs, is included at the end of this appendix (Figure 35).

AddBirthRecord

$Name, Month : \text{seq. of char.}$
$Day, Year : \mathbb{N}$
$known : \mathbb{P}\ Id$
$birth, birth' : Id \rightarrow \{Name, Month, Day, Year\}$
$id? : Id$
$name? : Name$
$month? : Month$
$day? : Day$
$year? : Year$
$rep! : Report$

$id? \in known$
$\#name? = 1..20$
$year? = 0..1995$
$(month? = Feb \land$
$(\text{mod}(year?/4) = 0 \land$
$day? = 1..29 \lor$
$\text{mod}(year?/4) \neq 0 \land$
$day? = 1..28) \lor$
$month? \in \{Apr, May, Jun, Sep, Nov\} \land$
$day? = 1..30 \lor$
$month? \in \{Jan, Mar, Jul, Aug, Oct, Dec\} \land$
$day? = 1..31)$
$birth' = birth \cup \{id? \rightarrow \{name?, month?, day?, year?\}\}$
$rep! = \text{'New record has been inserted'}) \lor$
$(id? \notin known$
$birth' = birth$
$rep! = \text{'Unknown Id'}) \lor$
$(\#name? \neq 1..20$
$birth' = birth$
$rep! = \text{'Invalid name'}) \lor$
$(month? \neq Feb$
$month? \notin \{Apr, May, Jun, Sep, Nov\}$
$month? \notin \{Jan, Mar, Jul, Aug, Oct, Dec\}$
$birth' = birth$
$rep! = \text{'Invalid month'}) \lor$
$(year? \neq 0..1995$
$birth' = birth$
$rep! = \text{'Invalid year'}) \lor$
$(month? = Feb \land$
$(\text{mod}(year?/4) = 0 \land$
$day? \neq 1..29 \lor$
$\text{mod}(year?/4) \neq 0 \land$
$day? \neq 1..28) \lor$
$month? \in \{Apr, May, Jun, Sep, Nov\} \land$
$day? \neq 1..30 \lor$
$month? \in \{Jan, Mar, Jul, Aug, Oct, Dec\} \land$
$day? \neq 1..31$
$birth' = birth$
$rep! = \text{'Invalid day'})$

Figure 34. Schema *AddBirthRecord*

CASE 1 OUTPUT:

```
Specification:
==============

dom telephones subseteq members &
dom telephones' subseteq members' &
((name? in members &
name? mapsto newnumber? notin telephones &
telephones' = telephones union name? mapsto newnumber? &
members' = members &
rep! = 'Okay')  |
(name? notin members &
telephones' = telephones &
members' = members &
rep! = 'Not a member') |
( name? mapsto newnumber? in telephones &
telephones' = telephones &
members' = members &
rep! = 'Entry already exists'))


The graph forward traversals starting from a cause node:
========================================================

(INV_CAUSE,1)((AND,1)((INV_EFFECT,1)),(AND,4)((EFFECT,5),(EFFECT,3),
(EFFECT,7)),(AND,3)((EFFECT,5),(EFFECT,3),(EFFECT,6)),(AND,5)
((EFFECT,2),(EFFECT,3),(EFFECT,4)))

(CAUSE,2)((AND,2)((AND,5)((EFFECT,2),(EFFECT,3),(EFFECT,4))),-(AND,3)
((EFFECT,5),(EFFECT,3),(EFFECT,6)))

(CAUSE,3)(-(AND,2)((AND,5)((EFFECT,2),(EFFECT,3),(EFFECT,4))),
(AND,4)((EFFECT,5),(EFFECT,3),(EFFECT,7)))

** Node representation: [negation/-](<node type>, <sequence number>)


The graph backward traversals starting from an effect node:
===========================================================

(INV_EFFECT,1)((AND,1)((INV_CAUSE,1)))

(EFFECT,2)((AND,5)((AND,2)((CAUSE,2)((R,1)),-(CAUSE,3)((R,1))),
(INV_CAUSE,1)))

(EFFECT,3)((AND,3)(-(CAUSE,2)((R,1)),(INV_CAUSE,1)),(AND,4)((CAUSE,3)
((R,1)),(INV_CAUSE,1)),(AND,5)((AND,2)((CAUSE,2)((R,1)),-(CAUSE,3)
((R,1))),(INV_CAUSE,1)))

(EFFECT,4)((AND,5)((AND,2)((CAUSE,2)((R,1)),-(CAUSE,3)((R,1))),
(INV_CAUSE,1)))

(EFFECT,5)((AND,3)(-(CAUSE,2)((R,1)),(INV_CAUSE,1)),(AND,4)((CAUSE,3)
((R,1)),(INV_CAUSE,1)))
```

```
(EFFECT,6)((AND,3)(-(CAUSE,2)((R,1)),(INV_CAUSE,1)))

(EFFECT,7)((AND,4)((CAUSE,3)((R,1)),(INV_CAUSE,1)))

** Node representation: [negation/-](<node type>, <sequence number>)


The graph cause constraint connections:
=======================================

(R,1)((CAUSE,2)(CAUSE,3))

** Node representation: [negation/-](<node type>, <sequence number>)


True-Effect Test Frames:
========================

-------------------------------------------------------------------
Frame No. ******* Cause No. *******   ******* Effect No.********
          1  2  3                     1  2  3  4  5  6  7
-------------------------------------------------------------------
    1     1  1  0                     1  1  1  1  0  0  0

    2     1  0  0                     1  0  1  0  1  1  0

    3     1  1  1                     1  0  1  0  1  0  1

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

 Causes:

     1 : dom telephones subseteq members
     2 : name? in members
     3 : name? mapsto newnumber? in telephones

 Effects:

     1 : dom telephones' subseteq members'
     2 : telephones' = telephones union name? mapsto newnumber?
     3 : members' = members
     4 : rep! = 'Okay'
     5 : telephones' = telephones
     6 : rep! = 'Not a member'
     7 : rep! = 'Entry already exists'

 Node Values:

     0 : FALSE
     1 : TRUE
     d : Don't care (Either TRUE or FALSE)


 The Number of Computations:

    Zeroes-AND Computations     :      0
    Non Zeroes-AND Computations :      8
```

CASE 2 OUTPUT:

```
Specification:
==============

(id? in known &
#name? = 1..30 &
birth_year? = 0..1995 &
(birth_month? = Feb &
 (mod(birth_year?/4) = 0 &
 birth_day? = 1..29 |
 mod(birth_year?/4) not= 0 &
 birth_day? = 1..28) |
 birth_month? in {Apr,May,Jun,Sep,Nov} &
 birth_day? = 1..30 |
 birth_month? in {Jan,Mar,Jul,Aug,Oct,Dec} &
 birth_day? = 1..31) &
birth' = birth union {id? mapsto
{name?,birth_month?,birth_day?,birth_year?}} &
rep! = 'New record has been inserted') |
id? notin known &
birth' = birth &
rep! = 'Unknown Id' |
#name? not= 1..30 &
birth' = birth &
rep! = 'Invalid name' |
birth_month? not= Feb &
birth_month? notin {Apr,May,Jun,Sep,Nov} &
birth_month? notin {Jan,Mar,Jul,Aug,Oct,Dec} &
birth' = birth &
rep! = 'Invalid birth month' |
birth_year? not= 0..1995 &
birth' = birth &
rep! = 'Invalid birth year' |
(birth_month? = Feb &
 (mod(birth_year?/4) = 0 &
 birth_day? not= 1..29 |
 mod(birth_year?/4) not= 0 &
 birth_day? not= 1..28) |
 birth_month? in {Apr,May,Jun,Sep,Nov} &
 birth_day? not= 1..30 |
 birth_month? in {Jan,Mar,Jul,Aug,Oct,Dec} &
 birth_day? not= 1..31) &
birth' = birth &
rep! = 'Invalid birth day'


The graph forward traversals starting from a cause node:
========================================================

(CAUSE,1)((AND,1)((EFFECT,1),(EFFECT,2)),-(AND,9)((EFFECT,3),
(EFFECT,4)))

(CAUSE,2)((AND,1)((EFFECT,1),(EFFECT,2)),-(AND,10)((EFFECT,3),
(EFFECT,5)))

(CAUSE,3)((AND,1)((EFFECT,1),(EFFECT,2)),-(AND,12)((EFFECT,3),
```

```
(EFFECT,7)))

(CAUSE,4)((AND,2)((OR,6)((AND,1)((EFFECT,1),(EFFECT,2)))),-(AND,11)
((EFFECT,3),(EFFECT,6)),(AND,13)((OR,17)((EFFECT,3),(EFFECT,8))))

(CAUSE,5)((AND,3)((OR,4)((AND,2)((OR,6)((AND,1)((EFFECT,1),
(EFFECT,2))))))),-(AND,5)((OR,4)((AND,2)((OR,6)((AND,1)((EFFECT,1),
(EFFECT,2))))))),(AND,14)((OR,15)((AND,13)((OR,17)((EFFECT,3),
(EFFECT,8))))),-(AND,16)((OR,15)((AND,13)((OR,17)((EFFECT,3),
(EFFECT,8))))))

(CAUSE,6)((AND,3)((OR,4)((AND,2)((OR,6)((AND,1)((EFFECT,1),
(EFFECT,2))))))),-(AND,14)((OR,15)((AND,13)((OR,17)((EFFECT,3),
(EFFECT,8))))))

(CAUSE,7)((AND,5)((OR,4)((AND,2)((OR,6)((AND,1)((EFFECT,1),
(EFFECT,2))))))),-(AND,16)((OR,15)((AND,13)((OR,17)((EFFECT,3),
(EFFECT,8))))))

(CAUSE,8)((AND,7)((OR,6)((AND,1)((EFFECT,1),(EFFECT,2)))),-(AND,11)
((EFFECT,3),(EFFECT,6)),(AND,18)((OR,17)((EFFECT,3),(EFFECT,8))))

(CAUSE,9)((AND,7)((OR,6)((AND,1)((EFFECT,1),(EFFECT,2)))),-(AND,18)
((OR,17)((EFFECT,3),(EFFECT,8))))

(CAUSE,10)((AND,8)((OR,6)((AND,1)((EFFECT,1),(EFFECT,2)))),
-(AND,11)((EFFECT,3),(EFFECT,6)),(AND,19)((OR,17)((EFFECT,3),
(EFFECT,8))))

(CAUSE,11)((AND,8)((OR,6)((AND,1)((EFFECT,1),(EFFECT,2)))),
-(AND,19)((OR,17)((EFFECT,3),(EFFECT,8))))

** Node representation: [negation/-](<node type>, <sequence number>)


The graph backward traversals starting from an effect node:
============================================================

(EFFECT,1)((AND,1)((CAUSE,1),(CAUSE,2),(CAUSE,3),(OR,6)((AND,2)
((CAUSE,4)((E,2)),(OR,4)((AND,3)((CAUSE,5),(CAUSE,6)((R,1),
(R,3),(R,4))),(AND,5)(-(CAUSE,5),(CAUSE,7)((R,1),(R,3),(R,4))))),
(AND,7)((CAUSE,8)((E,2)),(CAUSE,9)((R,3),(R,4))),(AND,8)((CAUSE,10)
((E,2)),(CAUSE,11)((R,4))))))

(EFFECT,2)((AND,1)((CAUSE,1),(CAUSE,2),(CAUSE,3),(OR,6)((AND,2)
((CAUSE,4)((E,2)),(OR,4)((AND,3)((CAUSE,5),(CAUSE,6)((R,1),
(R,3),(R,4))),(AND,5)(-(CAUSE,5),(CAUSE,7)((R,1),(R,3),(R,4))))),
(AND,7)((CAUSE,8)((E,2)),(CAUSE,9)((R,3),(R,4))),(AND,8)((CAUSE,10)
((E,2)),(CAUSE,11)((R,4))))))

(EFFECT,3)((AND,9)(-(CAUSE,1)),(AND,10)(-(CAUSE,2)),(AND,11)
(-(CAUSE,4)((E,2)),-(CAUSE,8)((E,2)),-(CAUSE,10)((E,2))),(AND,12)
(-(CAUSE,3)),(OR,17)((AND,13)((CAUSE,4)((E,2)),(OR,15)((AND,14)
((CAUSE,5),-(CAUSE,6)((R,1),(R,3),(R,4))),(AND,16)(-(CAUSE,5),
-(CAUSE,7)((R,1),(R,3),(R,4))))),(AND,18)((CAUSE,8)((E,2)),
-(CAUSE,9)((R,3),(R,4))),(AND,19)((CAUSE,10)((E,2)),-(CAUSE,11)
((R,4)))))

(EFFECT,4)((AND,9)(-(CAUSE,1)))
```

```
(EFFECT,5)((AND,10)(-(CAUSE,2)))

(EFFECT,6)((AND,11)(-(CAUSE,4)((E,2)),-(CAUSE,8)((E,2)),-(CAUSE,10)
((E,2))))

(EFFECT,7)((AND,12)(-(CAUSE,3)))

(EFFECT,8)((OR,17)((AND,13)((CAUSE,4)((E,2)),(OR,15)((AND,14)
((CAUSE,5),-(CAUSE,6)((R,1),(R,3),(R,4))),(AND,16)(-(CAUSE,5),
-(CAUSE,7)((R,1),(R,3),(R,4))))),(AND,18)((CAUSE,8)((E,2)),
-(CAUSE,9)((R,3),(R,4))),(AND,19)((CAUSE,10)((E,2)),-(CAUSE,11)
((R,4)))))

** Node representation: [negation/-](<node type>, <sequence number>)


The graph cause constraint connections:
=======================================

(R,1)((CAUSE,6)(CAUSE,7))

(E,2)((CAUSE,4)(CAUSE,8),(CAUSE,10))

(R,3)((CAUSE,9)(CAUSE,6),(CAUSE,7))

(R,4)((CAUSE,11)(CAUSE,6),(CAUSE,7),(CAUSE,9))

** Node representation: [negation/-](<node type>, <sequence number>)


True-Effect Test Frames:
========================
```

Frame No.	******* Cause No. *******								******* Effect No.********							
	1	2	3	4	5	6	7	8	1	2	3	4	5	6	7	8
	9	10	11													
1	1	1	1	0	0	0	0	0	1	1	0	0	0	0	0	0
	0	1	1													
2	1	1	1	0	0	0	0	0	1	1	0	0	0	0	0	0
	1	1	1													
3	1	1	1	0	0	1	1	0	1	1	0	0	0	0	0	0
	1	1	1													
4	1	1	1	0	1	1	1	0	1	1	0	0	0	0	0	0
	1	1	1													
5	1	1	1	0	0	0	0	1	1	1	0	0	0	0	0	0
	1	0	1													
6	1	1	1	0	0	1	1	1	1	1	0	0	0	0	0	0
	1	0	1													
7	1	1	1	0	1	1	1	1	1	1	0	0	0	0	0	0

```
          1  0  1

   8      1  1  1  1  0  1  1  0      1  1  0  0  0  0  0  0
          1  0  1

   9      1  1  1  1  1  1  1  0      1  1  0  0  0  0  0  0
          1  0  1

  10      0  1  1  1  0  1  1  0      0  0  1  1  0  0  0  0
          1  0  1

  11      1  0  1  1  0  1  1  0      0  0  1  0  1  0  0  0
          1  0  1

  12      1  1  1  0  0  0  0  0      0  0  1  0  0  1  0  0
          d  0  d

  13      1  1  0  1  0  1  1  0      0  0  1  0  0  0  1  0
          1  0  1

  14      1  1  1  1  0  1  0  0      0  0  1  0  0  0  0  1
          1  0  1

  15      1  1  1  0  0  0  0  0      0  0  1  0  0  0  0  1
          0  1  0

  16      1  1  1  0  1  0  0  0      0  0  1  0  0  0  0  1
          0  1  0

  17      1  1  1  0  0  0  0  1      0  0  1  0  0  0  0  1
          0  0  0

  18      1  1  1  0  1  0  0  1      0  0  1  0  0  0  0  1
          0  0  0

  19      1  1  1  1  0  0  0  0      0  0  1  0  0  0  0  1
          0  0  0

  20      1  1  1  1  1  0  0  0      0  0  1  0  0  0  0  1
          0  0  0
```

Causes:

```
    1 : id? in known
    2 : #name? = 1..30
    3 : birth_year? = 0..1995
    4 : birth_month? = Feb
    5 : mod(birth_year?/4) = 0
    6 : birth_day? = 1..29
    7 : birth_day? = 1..28
    8 : birth_month? in {Apr,May,Jun,Sep,Nov}
    9 : birth_day? = 1..30
   10 : birth_month? in {Jan,Mar,Jul,Aug,Oct,Dec}
   11 : birth_day? = 1..31
```

Effects:

```
    1 : birth' =
```

```
        birth union {id? mapsto
{name?,birth_month?,birth_day?,birth_year?}}
     2 : rep! = 'New record has been inserted'
     3 : birth' = birth
     4 : rep! = 'Unknown Id'
     5 : rep! = 'Invalid name'
     6 : rep! = 'Invalid birth month'
     7 : rep! = 'Invalid birth year'
     8 : rep! = 'Invalid birth day'

 Node Values:

     0 : FALSE
     1 : TRUE
     d : Don't care (Either TRUE or FALSE)


 The Number of Computations:

    Zeroes-AND Computations      :    170
    Non Zeroes-AND Computations  :    428
```

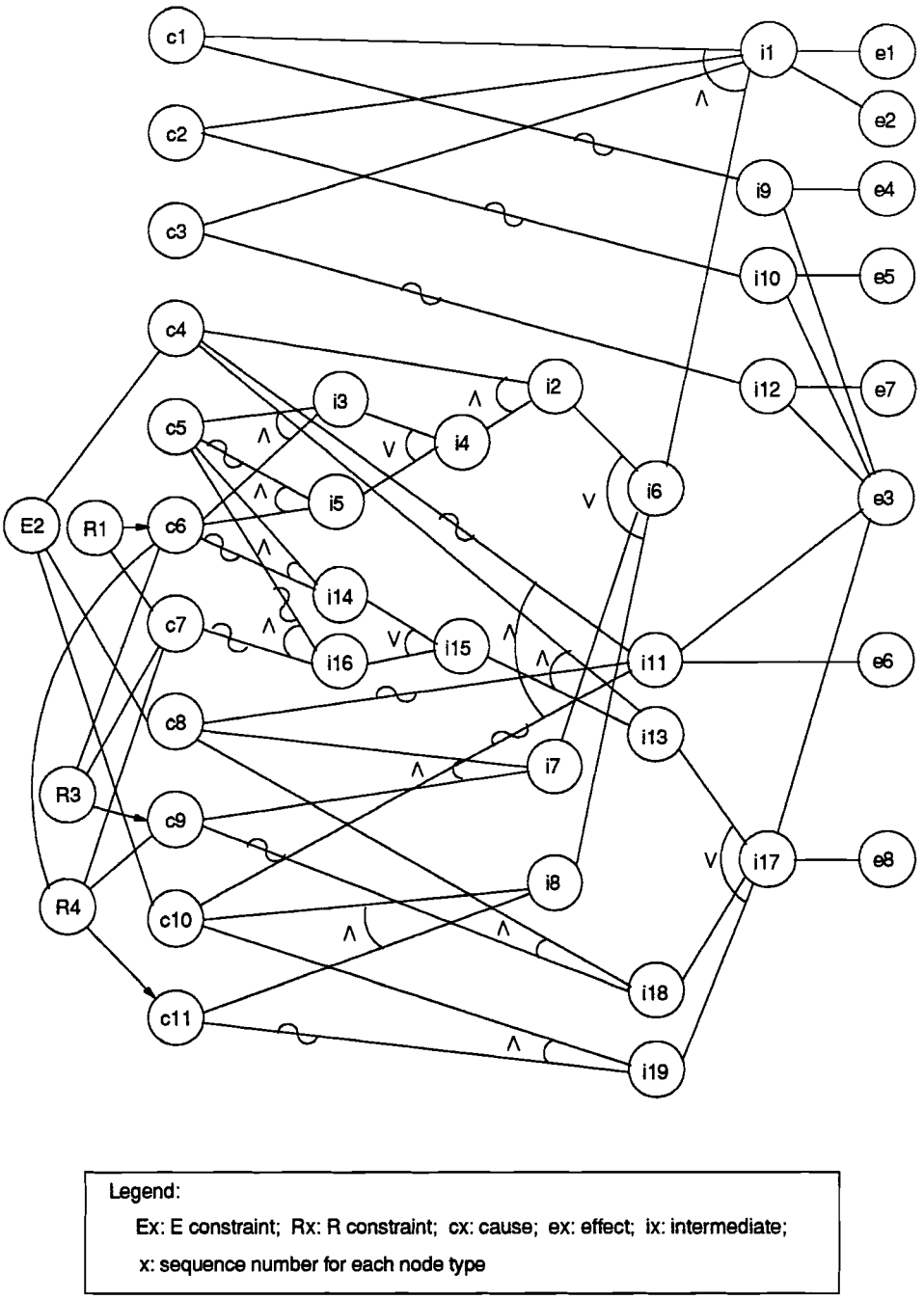

Figure 35. The cause-effect graph of the schema *AddBirthRecord*

APPENDIX C

PROGRAM LISTING

The source code of the tool is stored in two files: z.h and z.c. File z.h contains the data definitions and the structure declarations; File z.c contains the procedure codes. The following pages show the contents of z.h and z.c, respectively.

```
/*****************************************************************************
 *****************************************************************************
   File: z.h
   This file contains the definition of data and the declaration of structures
   used by the test frame data generator procedures.
 *****************************************************************************
 *****************************************************************************/


#define    NOOPER        0
#define    AND           1
#define    OR            2
#define    E             3    /* Exclusive constraint */
#define    R             4    /* Requires constraint */
#define    RR            5    /* Similar predicate indicator */
#define    CAUSE        10
#define    INV_CAUSE    11    /* State invariant cause */
#define    EFFECT       20
#define    INV_EFFECT   21    /* State invariant effect */


typedef struct graph_node {
    int  type;                 /* The node type:
                                   1: AND intermediate node
                                   2: OR intermediate node
                                   3: E constraint node
                                   4: R constraint node
                                  10: Cause node
                                  11: Invariant cause node
                                  20: Effect node
                                  21: Invariant effect node */
    int  number;               /* The sequence node number */
    int  scope;                /* The node scope level */
    struct graph_link *forw, *bakw;
                               /* Pointer to the first forward and
                                  backward links */
    struct graph_node *next;   /* Pointer to the next sequence similar node */
} GRPNODE;


typedef struct graph_link {
    int negation;              /* Link negation:
                                  0: NOT link
                                  1: IDENTITY link  */
    struct graph_node *node;   /* Pointer to an adjacent node */
    struct graph_link *next;
} LINK;


typedef struct graph_header {
    GRPNODE *cause;            /* Pointer to the first cause node */
    GRPNODE *effect;           /* Pointer to the first effect node */
    GRPNODE *rightmost_inter; /* Pointer to the first rightmost intermediate
                                 node */
    GRPNODE *constraint;       /* Pointer to the first constraint node */
    int ncause, ninter, neffect;
                               /* The number of cause, intermediate, and
                                  effect node */
} GRAPH;


typedef struct str_80 {        /* String of 80 characters */
    char str[80];
} STR_80;
```

```
typedef struct value_boundary {
    int    bound_type;          /* The type of boundary */
    int    n_elm;               /* The element number of the value list */
    STR_80 *strlist;            /* Pointer to a string value list */
    float  *numlist;            /* Pointer to a numeric value list */
    float  lower;               /* The lower boundary value */
    float  upper;               /* The upper boundary value */
    char   val_type[5];         /* The boundary value type */
} BOUND_ELM;


typedef struct cause_effect_record {
    int  number;                /* The sequence number of cause or effect
                                   predicates */
    char pred_part[3][80];      /* The three predicate parts */
    BOUND_ELM *boundary;        /* Pointer to a variable boundary record */
    struct cause_effect_record *next;
                                /* Pointer to the next cause (precondition)/
                                   effect (postcondition) record */
} TBL_REC;


typedef struct table {          /* The precondition (cause) and postcondition
                                   (effect) table header */
    TBL_REC *cause;             /* Pointer to the first precondition predicate
                                   record */
    TBL_REC *effect;            /* Pointer to the first postcondition predicate
                                   record */
} TABLE;


typedef struct predicate {
    char part[3][80];          /* The tree predicate parts */
    int  type, scope;          /* The type and scope level of the predicate */
} PRED;


typedef struct operator {
    int  type, scope;          /* The type and scope level of a logical
                                  connective operator */
} OPER;


typedef struct scope_node {  /* The element of the list of the scope level
                                of the rightmost intermediate nodes */
    int scope;               /* The scope level */
    GRPNODE *node;           /* Pointer to an intermediate node */
    struct scope_node *prev; /* Pointer to the previous element */
    struct scope_node *next; /* Pointer to the next element */
} SCOPENODE;


typedef struct scope_list {
    SCOPENODE *head;         /* Pointer to the first list element */
    SCOPENODE *last;         /* Pointer to the last list element */
} SCOPELIST;


typedef struct test_frame {
    int  *cause;             /* Pointer to the array of cause node values */
    int  *inter;             /* Pointer to the array of intermediate node
                                values */
    int  *derive;            /* Pointer to the array of intermediate node
                                derivation status */
    int  *effect;            /* Pointer to the array of effect node values */
    struct test_frame *next; /* Pointer to the next test frame */
} TEST_FRAME;
```

```
typedef struct test_list {
    TEST_FRAME *head;       /* Pointer to the first test frame */
    TEST_FRAME *tail;       /* Pointer to the last test frame */
    struct test_list *next; /* Pointer to the next test fame list */
} TEST_LIST;

typedef struct constraint_element {
    int node_no;            /* The related cause node number */
    int type;               /* The constraint type */
    struct constraint_element *next;
                            /* Pointer to the next constraint element */
} CST_ELM;

typedef struct constraint_list {
    CST_ELM *head;          /* Pointer to the first constraint element */
    CST_ELM *tail;          /* Pointer to the last constraint element */
} CST_LIST;

typedef struct constrained_causes_element {
    GRPNODE *node;          /* Pointer to a constrained cause node */
    struct constrained_causes_element *next;
                            /* Pointer to the next element */
} CC_ELM;

typedef struct constrained_cause_list {
    CC_ELM *head;           /* Pointer to the first constrained cause element */
    CC_ELM *tail;           /* Pointer to the last constrained cause element */
} CC_LIST;
```

```
/*****************************************************************************
 *****************************************************************************
   File: z.c.
   This file contains all procedures for generating test frames from Z
   specifications.
*****************************************************************************
*****************************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "z.h"


/*****************************************************************************
*   PROTOTYPE
*****************************************************************************/
void print_spec();                  /* Print the tool input */
void construct_graph();             /* Construct a cause-effect graph */
int  scan_schema_predicate();       /* Scan a schema predicate and the following
                                       connective operator */
int  lookup_table();                /* Look a predicate record up in a predicate
                                       table */
int  examine_negation();            /* Examine the negation of a predicate */
int  insert_table();                /* Insert a new record to the predicate
                                       table */
void derive_val_boundaries();       /* Derive the boundary value of a variable
                                       declared in a predicate */
void set_value_list();              /* Set a boundary value list */
void set_single_value();            /* Set a single boundary value */
void set_range_values();            /* Set a boundary range value */
BOUND_ELM* alloc_boundary_elm();    /* Allocate a boundary element */
void add_cause_effect_node();       /* Add a cause or an effect node in a cause-
                                       effect graph */
GRPNODE* create_node();             /* Create a graph node */
void mk_link();                     /* Link a graph node to the other node */
void place_inter_node();            /* Place/insert a new intermediate node */
void rm_ineffect_inter();           /* Remove an ineffective intermediate node */
int  count_link_number();           /* Count the number of forward/backward links
                                       of a node */
void maintain_rightmost_inter_scplist();
                                    /* Add a new element to the rightmost inter-
                                       mediate node list */
void arrange_rightmost_inter_links();
                                    /* Arrange the links of the rightmost inter-
                                       mediate nodes */
void cp_forw_link();                /* Copy the forward links of a node  to
                                       another node */
void cp_bakw_link();                /* Copy the backward links of a node to
                                       another node */
void sub_cp_link();                 /* Subroutine to copy the node links */
void rm_forw_link();                /* Remove the forward links of a node */
void rm_bakw_link();                /* Remove the backward links of a node */
void sub_rm_link();                 /* Subroutine to remove a node links */
void replace_rightmost_inter();     /* Replace one of the rightmost intermediate
                                       node with the new one */
void rm_rightmost_inter();          /* Remove one of the rightmost intermediate
                                       nodes */
CST_LIST* search_constraints();     /* Search any constraint relation of a node */
void add_constraint_list();         /* Add a constraint list element */
int  compare_boundary();            /* Compare the boundary of a variable
                                       declared in two different predicates */
int  sub_compare_boundary_1();      /* Subroutine 1 to compare a variable
                                       boundaries */
int  sub_compare_boundary_2();      /* Subroutine 2 to compare a variable
                                       boundaries */
int  sub_compare_boundary_3();      /* Subroutine 3 to compare a variable
                                       boundaries */
int  sub_compare_boundary_4();      /* Subroutine 4 to compare a variable
                                       boundaries */
int  sub_compare_boundary_5();      /* Subroutine 5 to compare a variable
                                       boundaries */
int  sub_compare_boundary_6();      /* Subroutine 6 to compare a variable
```

```
                                    boundaries */
int  search_strval();              /* Search all values of one string list in
                                    another string list */
int  search_numval();              /* Search all values of one numeric values in
                                    another numeric list */
void add_constraint_node_link(); /* Add a constraint node to a cause-effect
                                    graph */
GRPNODE* search_existing_cst();  /* Search a constraint relation in a cause-
                                    effect graph */
void remove_inter_node_duplication();
                                 /* Remove any intermediate node duplication */
int  compare_two_inter();        /* Compare the backward links of two inter-
                                    mediate node */
void print_graph();              /* Print the traversals of a cause-effect
                                    graph */
void print_graph_forw();         /* Print the forward traversals of a cause-
                                    effect graph */
void print_graph_bakw();         /* Print the backward traversals of a cause-
                                    effect graph */
void print_graph_node();         /* Print a graph node */
TEST_LIST *derive_test_frame();  /* Derive test frames from a cause-effect
                                    graph */
TEST_LIST *trace_one_rightmost_inter_backward();
                                 /* Trace a cause-effect graph starting from
                                    one of the rightmost intermediate nodes */
TEST_LIST *alloc_test_list();    /* Allocate a test list header */
TEST_FRAME *add_test_frame();    /* Add a test frame to a test list */
void copy_test_frame();          /* Copy the element values of one test frame
                                    to another test frame */
void trace_one_node_backward();  /* Trace a graph backward one step from a
                                    graph node */
void derive_node_input_values(); /* Derive a node input value combinations */
int  derive_one_input_node_val();
                                 /* Derive a node input value */
int  examine_forward_conflict(); /* Examine any conflict on a forward value
                                    transformation */
int  derive_effected_node_values();
                                 /* Derive the value of effect nodes */
int  derive_false_AND_input_values();
                                 /* Derive the input value combinations of an
                                    AND node */
int  derive_true_OR_input_values();
                                 /* Derive the input value combinations of an
                                    OR node */
int  set_zeroes_AND_input_values();
                                 /* Set a node input value combination effecting
                                    all zeroes AND node input values */
void search_constrained_causes();
                                 /* Find constraint causes in a graph */
int  set_zeroes_AND_cause_values();
                                 /* Set the cause value combination effecting
                                    all zeroes AND node input values */
int  reset_constrained_cause_values();
                                 /* Reset the value of a graph constraint node
                                    values */
void derive_other_rightmost_inter_values();
                                 /* Derive the value of the other rightmost
                                    intermediate node other than the first
                                    derived node */
int  derive_one_rightmost_inter_val();
                                 /* Derive the value of one of the rightmost
                                    intermediate nodes */
void remove_test_frame_duplication();
                                 /* Remove any test frame duplication */
TEST_FRAME *delete_test_frame(); /* Delete a test frame */
void derive_effect_values();     /* Derive the values of a graph effect
                                    nodes */
void print_test_frame();         /* Print test frames */
```

```
/*****************************************************************************
   GLOBAL VARIABLES
*****************************************************************************/
int compute_0=0;                    /* Normal derivation computation */
int compute_1=0;                    /* All zeroes AND inputs computation */


/*****************************************************************************
*   MAIN()
*   The main procedure for generating test frames from a Z specification.
*****************************************************************************/
main()
{

    FILE *fp;                       /* Input file */
    GRAPH graph;                    /* Cause-effect graph */
    TABLE table;                    /* Predicates table */
    TEST_LIST *true_test;           /* Test frame lists */

    /*-------------------------------------------------------------
     *    Open the input file
     */
    if ((fp=fopen("dtest","r"))==NULL) {
       printf("\nCan not open input file\n");
       exit(1);
    }

    /*-------------------------------------------------------------
     *   Set the initial value of the graph and the table elements
     */
    graph.cause = NULL;
    graph.effect = NULL;
    graph.constraint = NULL;
    graph.rightmost_inter = NULL;
    graph.ncause  = 0;
    graph.ninter  = 0;
    graph.neffect  = 0;
    table.cause = NULL;
    table.effect = NULL;

    /*-------------------------------------------------------------
     *    The main processes
     */
    print_spec("dtest");            /* Print the specification */
    construct_graph(fp, &graph, &table);
                                    /* Construct the cause-effect graph */
    print_graph(&graph);            /* Print the cause-effect graph */
    true_test=derive_test_frame(&graph, 1);
                                    /* Derive the test frames */
    print_test_frame(true_test, &graph, &table, 1);
                                    /* Print the test frames */
}


/*****************************************************************************
*   VOID PRINT_SPEC()
*   This procedure is used to print the input Z specification.
*****************************************************************************/
void print_spec()
{
    printf("Specification:\n");
    printf("==============\n\n");
    fflush(stdout);
    system("cat dtest");
    printf("\n\n\n");
}
```

```
/*****************************************************************************
*   VOID CONSTRUCT_GRAPH()
*   This procedure is used to construct the cause-effect graph by scanning
*   the schema predicate of the Z specification and adding the graph components
*   for each scanning of a predicate and any operator that follows the predi-
*   cate.
*****************************************************************************/
void construct_graph(fp, graph, table)
FILE    *fp;
GRAPH   *graph;
TABLE   *table;
{
    PRED   pred;                    /* Predicate */
    OPER   oper;                    /* Predicate operator  */
    int    node_number;             /* Cause/effect node number */
    CST_LIST *constraint=NULL;      /* Cause constraints list */

    while (scan_schema_predicate(fp, &pred, &oper)) {
                                    /* Loop for adding the graph components
                                       while there is a scanned predicate */

       if ((node_number=lookup_table(table, pred)) == 0) {
                                    /* The scanned predicate is not in the table
                                       yet */

          node_number = insert_table(table, pred);
          if (pred.type == EFFECT || pred.type == INV_EFFECT) {
                                    /* The scanned predicate is an effect
                                       predicate */

             (graph->neffect)++;
             add_cause_effect_node(graph, pred, node_number, oper, 1);
          }
          else {
                                    /* The scanned predicate is a cause
                                       predicate */

             (graph->ncause)++;
             add_cause_effect_node(graph, pred, node_number, oper, 1);
             if (graph->ncause > 1 && strchr(pred.part[0], '?') != NULL) {
                                    /* The process for searching constraints of
                                       the cause node and adding the graph
                                       constraint components */

                constraint=search_constraints(graph->ncause, table);
                if (constraint != NULL)
                   add_constraint_node_link(constraint, graph);
             }
          }
       }
       else                         /* The scanned predicate is already in the
                                       table  */

          add_cause_effect_node(graph, pred, node_number, oper, 0);
    }
    remove_inter_node_duplication(graph);
                                    /* Remove any intermediate node
                                       duplication */
}


/*****************************************************************************
*   INT SCAN_SCHEMA_PREDICATE()
*   This procedure is used to scan a predicate and any operator that follows
*   the predicate from the input specification.  The procedure returns 1 if a
*   predicate is scanned and 0 if no predicate is scanned.
*****************************************************************************/
int scan_schema_predicate(fp, predicate, operator)
FILE *fp;
PRED *predicate;
OPER *operator;
```

```
{
    int i, j, ipart=0;          /* Counters */
    char str[80], temp[80];     /* Working and temporary string */
    static int curr_scope=0;    /* Current predicate or operator scope */
    int   prev_pred_type=0;     /* Previous predicate type */

    /*-------------------------------------------------------------
     *  Set the initial value of the predicate and operator elements
     */
    predicate->scope = curr_scope;
    operator->scope = curr_scope;
    operator->type = NOOPER;
    for (i=0; i<=2; i++) memset(predicate->part[i],'\0',80);

    while( fscanf(fp, "%s", str) != EOF || ipart != 0) {
                                /* Loop for getting a predicate and any operator
                                   that follows the predicate by scanning a
                                   string from the input */

       if (ipart==0 && (strcmp(str,"in")==0 || strcmp(str, "notin")==0 ||
           strcmp(str,"=")==0 || strcmp(str,"not=")==0 ||
           strcmp(str,"subset")==0 || strcmp(str,"notsubset")==0 ||
           strcmp(str,"subseteq")==0 || strcmp(str,"notsubseteq")==0)) {
                                /* The following are processes if a comparison
                                   or membership operator is scanned:
                                   -  Copies the string to the second predicate
                                      part
                                   -  Derives the predicate type  */

          strcpy(predicate->part[1],str);
          ipart = 2;
          if (predicate->part[0][j=strlen(predicate->part[0])-1]=='!' ||
              strncmp(predicate->part[0]+j, "'",1)==0)
                                /* The following processes are accomplished if
                                   the end of the first predicate part is
                                   decorated with a prime or exclamation mark */

             if (prev_pred_type==INV_CAUSE)
                predicate->type = INV_EFFECT;
             else
                predicate->type = EFFECT;
          else
                                /* The end of the first predicate part is not
                                   decorated with a prime or exclamation mark */

             if (strchr(predicate->part[0],'?')==NULL)
                                /* If the end of the first predicate part is not
                                   an input predicate */

                predicate->type = INV_CAUSE;
             else
                predicate->type = CAUSE;
       }
       else {
                                /* The following processes are accomplished if
                                   a comparison or membership operator is not
                                   scanned  */

          if (str[0]!='&' && str[0]!='|'&& str[0]!='\0') {
                                /* The following are processes if a predicate
                                   operator and the end of line is not scanned:
                                   - Decrease the predicate and operator scope
                                     if the first character of the string is
                                     open bracket
                                    - Copy the string into the appropriate
                                      predicate part
                                 */
             while (str[0]=='(') {
                                /* Loop for decreasing the predicate and operator
                                   scope and eliminating the first character of
                                   the string */
```

```
                predicate->scope--;
                operator->scope--;

                strncpy(temp, str+1, strlen(str));
                strcpy(str,temp);
            }

            if (predicate->part[ipart][0] != '\0') {
                            /* If this predicate part is not empty */
                predicate->part[ipart][strlen(predicate->part[ipart])]=' ';
                strcat(predicate->part[ipart], str);
            }
            else
                            /* This predicate part is empty */
                strcpy(predicate->part[ipart], str);

            str[0]='\0';
          }
          else {
                            /* The following are processes if a predicate
                               operator or the end of line is scanned:
                               - Set the operator type
                               - Increase the operator scope if the end of
                                 the string is a close bracket
                               - Set the current scope
                             */
            if (str[0]=='&')
                operator->type = AND;
            else
                if (str[0]=='|')
                    operator->type = OR;

            while (predicate->part[2][j=strlen(predicate->part[2])-1] == ')'){
                            /* Loop for increasing the operator scope */
                 predicate->part[2][j] = '\0';
                 ++operator->scope;
            }
            curr_scope = operator->scope;
            return(1);
          }
        }
        prev_pred_type = predicate->type;
                            /* Set the previous predicate type */
    }
    return(0);
}


/*****************************************************************************
*   INT LOOKUP_TABLE()
*   This procedure is used to search for a predicate in the predicate table.
*   If the same predicate is found, the procedure returns the sequence number
*   of  the same predicate.  If the negation of the predicate is found, the
*   procedure returns the negation number of the negation predicate.
*   Otherwise, the procedure returns 0.
*****************************************************************************/
int lookup_table(table, pred)
TABLE *table;
PRED  pred;
{
    TBL_REC *search;          /*  Current predicate in the table */
    TBL_REC *temp1, *temp2;   /*  Working records */
    int     match=0,          /*  Match indicator */
            no,               /*  The sequence predicate number */
            negation=0,       /*  Negation indicator */
            cause_rec=0;      /*  Cause record indicator */


    /*-------------------------------------------------------------
     *   Return 0 if the predicate table is empty
     */
```

```
if ((pred.type==CAUSE || pred.type==INV_CAUSE) && table->cause==NULL ||
    (pred.type==EFFECT || pred.type==INV_EFFECT) && table->effect==NULL)
   return (0);

/*-------------------------------------------------------------
 *   Initialize the current table predicate to be compared
 */
if (pred.type==CAUSE || pred.type==INV_CAUSE) {
   search = table->cause;
   cause_rec=1;
}
else
   search = table->effect;

do {
                        /* Searching loop */

    if (strcmp(search->pred_part[0], pred.part[0])==0) {
                        /* If the first part of the two predicate are the
                           same */

       if (strcmp(search->pred_part[2], pred.part[2])==0)
                        /* If the third part of the two predicate are the
                           same */

          if (strcmp(search->pred_part[1], pred.part[1])==0)
                        /* The second part of the two predicate are also
                           the same */
             match = 1;
          else
                        /* If the second part of the two predicates are
                           not the same, examine the negation between
                           these parts  */
             match = examine_negation(search->pred_part[1], pred.part[1]);

       else
          if (strcmp(search->pred_part[1], pred.part[1])==0  ||
              strcmp(search->pred_part[1], "in")==0 &&
              strstr(pred.part[1], "in")!=NULL) {
                        /* If the second part of the two predicate are the
                           same or the two predicate are specify the set
                           membership, compare the boundaries of the
                           variable specified by the two predicates.  If
                           the two boundaries are the same, the same or
                           the negation of the predicate is found. */

                        /* Derive the boundary value stated by each
                           predicate */
             if (cause_rec)
                temp1 = search;
             else {
                temp1 = (TBL_REC *) malloc (sizeof(TBL_REC));
                strcpy(temp1->pred_part[0], search->pred_part[0]);
                strcpy(temp1->pred_part[1], search->pred_part[1]);
                strcpy(temp1->pred_part[2], search->pred_part[2]);
                derive_val_boundaries(temp1);
             }
             temp2 = (TBL_REC *) malloc (sizeof(TBL_REC));
             strcpy(temp2->pred_part[0], pred.part[0]);
             strcpy(temp2->pred_part[1], pred.part[1]);
             strcpy(temp2->pred_part[2], pred.part[2]);
             derive_val_boundaries(temp2);

                        /* Compare the two boundaries */
             if (compare_boundary(temp1->boundary, temp2->boundary)==RR)
                if (strcmp(pred.part[1], "notin")==0)
                   match = 2;
                else
                   match = 1;

                        /* Free the working space */
             if (!cause_rec) {
```

```
                    if (temp1->boundary->numlist)
                       free(temp1->boundary->numlist);
                    if (temp1->boundary->strlist)
                       free(temp1->boundary->strlist);
                    free(temp1->boundary);
                    free(temp1);
                 }
                 if (temp2->boundary->numlist)
                    free(temp2->boundary->numlist);
                 if (temp2->boundary->strlist)
                    free(temp2->boundary->strlist);
                 free(temp2->boundary);
                 free(temp2);
              }
          }
          no = search->number;
          search = search->next;
      } while (match==0  &&  search != NULL);

      if (match == 0)
         return (0);
      else
         if (match==1)
            return (no);
         else
            return (-no);
}


/*****************************************************************************
*   INT EXAMINE_NEGATION()
*   This procedure is used to examine the negation between two comparison or
*   set membership operators.  If one operator is the negation of the other,
*   then the procedure returns 2.  Otherwise the procedure returns 0.
*****************************************************************************/
int examine_negation(pred_search, pred_in)
char *pred_search, *pred_in;
{

    if (strncmp(pred_search,"in",2)==0 && strncmp(pred_in,"notin",5)==0 ||
       strncmp(pred_search,"=",1)==0 && strncmp(pred_in,"not=",4)==0 ||
       strncmp(pred_search,">",1)==0 && strncmp(pred_in,"<=",2)==0 ||
       strncmp(pred_search,"<=",2)==0 && strncmp(pred_in,">",1)==0 ||
       strncmp(pred_search,">=",2)==0 && strncmp(pred_in,"<",1)==0 ||
       strncmp(pred_search,"<",1)==0 && strncmp(pred_in,">=",2)==0 ||
       strncmp(pred_search,"subset",6)==0&& strncmp(pred_in,"notsubset",9)==0)
       return(2);
    else
       return(0);
}



/*****************************************************************************
*   INT INSERT_TABLE()
*   This procedure is used to insert a new record (predicate) into the predicate
*   table.  If the new record is same as the input predicate, the procedure
*   returns the sequence number of the new record.  If the new record is the
*   negation of the input predicate, then the procedure returns the negation
*   of the record number.
*****************************************************************************/
int insert_table(table, pred)
TABLE *table;
PRED  pred;
{
    TBL_REC *prev, *curr, *new;  /* Previous, current, and new record */
    int negation=0;              /* Negation indicator */


    /*-------------------------------------------------------------
     *   Allocate a new record and stores the first and third predicate
     *   parts
```

```
     */
    new = (TBL_REC *) malloc (sizeof(TBL_REC));
    strcpy(new->pred_part[0], pred.part[0]);
    strcpy(new->pred_part[2], pred.part[2]);
    new->boundary = NULL;
    new->next = NULL;

    /*-------------------------------------------------------------
     *   Store the second predicate part
     */
    if (strncmp(pred.part[1],"not",3)==0) {
                                /* If the beginning of the second part of the
                                   input predicate is a string "not", then
                                   eliminates the string "not" and sets the
                                   negation indicator */

       strcpy(new->pred_part[1], pred.part[1]+3);
       negation = 1;
    }
    else
       strcpy(new->pred_part[1], pred.part[1]);


    /*-------------------------------------------------------------
     *   The insertion of a new record into the cause or effect table
     */
    if ((pred.type==CAUSE || pred.type==INV_CAUSE) &&
        (curr = table->cause)==NULL ||
        (pred.type==EFFECT || pred.type==INV_EFFECT) &&
        (curr = table->effect)==NULL) {
                                /* The cause or effect table is empty */

       if (pred.type==CAUSE || pred.type==INV_CAUSE)
                                /* Insert the new record into the cause table,
if the predicate type is CAUSE or
                                   INV_CAUSE  */
          table->cause = new;
       else
                                /* Insert the new record into the effect
                                   table, if the predicate type is EFFECT  */
          table->effect = new;
       new->number = 1;         /* Sets the new record number */
    }
    else {
                                /* The insertion of the new record if the
                                   table is not empty */

       do {
                                /* Loop for getting the last record in the
                                   table */
          prev = curr;
          curr = curr->next;
       } while (curr != NULL);
       prev->next = new;
       new->number = prev->number + 1;
    }

    if (pred.type==CAUSE)
                                /* If the predicate is a cause predicate,
                                   set any variable value boundaries stated
                                   in the cause predicate */
       derive_val_boundaries(new);

    /*-------------------------------------------------------------
     *   Return the process result code
     */
    if (negation)
       return (-new->number);
    else
       return (new->number);
}
```

```
/*****************************************************************************
*   VOID DERIVE_VAL_BOUNDARIES()
*   This procedure is used to add the table record with any variable boundary
*   information stated in the predicate.
*****************************************************************************/
void derive_val_boundaries(rec)
TBL_REC *rec;
{

    if (strstr(rec->pred_part[1], "in")!=NULL &&
        strncmp(rec->pred_part[2], "{", 1)==0)
                                /* If the variable (stored as the first
                                   predicate part) is a set member, call the
                                   procedure to set the list of the member */
       set_value_list(rec);
    else
       if (strcmp(rec->pred_part[1], "=")==0 &&
              strstr(rec->pred_part[2],"..")==NULL)
                                /* If the variable is equal to a certain
                                   value, call the procedure to store the
                                   value */

          set_single_value(rec);
       else
          if (atof(rec->pred_part[2])!=0 ||
              strcmp(rec->pred_part[2],"0")==0 ||
              (strcmp(rec->pred_part[1], "=")==0 &&
               strstr(rec->pred_part[2],"..")!=NULL) )
                                /* If the variable is in a certain value
                                   range, call procedure to store the lower
                                   and/or the upper boundary value(s) */

             set_range_values(rec);
          else
             if (strcmp(rec->pred_part[1],"in")==0 &&
                 (strcmp(rec->pred_part[2],"R")==0 ||
                  strcmp(rec->pred_part[2], "Z")==0 ||
                  strcmp(rec->pred_part[2], "N")==0 ||
                  strcmp(rec->pred_part[2], "N1")==0)) {
                                /* If the variable value is in a certain
                                   number type, stores the boundary type  */

                rec->boundary = alloc_boundary_elm(6);
                strcpy(rec->boundary->val_type, rec->pred_part[2]);
             }
}


/*****************************************************************************
*   VOID SET_VALUE_LIST()
*   This procedure is used to derive and record the member list of a value
*   set.
*****************************************************************************/
void set_value_list(rec)
TBL_REC *rec;
{
    char set[80], str[80], *p;   /*  Working strings and a character pointer */
    int  i=0, n;                 /*  Counters */


    /*-------------------------------------------------------------
     *   The process to obtain the number of the set member
     */
    strcpy(set, rec->pred_part[2]+1);
    set[strlen(set)-1]='\0';
    for (p=strtok(set,","), n=1; p != NULL;  n++)
        p = strtok('\0', ",");


    /*-------------------------------------------------------------
     *   Get the first member
     */
```

```
    strcpy(set, rec->pred_part[2]+1);
    set[strlen(set)-1]='\0';
    p=strtok(set,",");

    /*-----------------------------------------------------------------
     *  The process to set the member list
     */
    if (atof(p) == 0 && strcmp(set, "0") != 0) {
                                /* The following processes are accomplished
                                   if the value set is a set of strings */

       rec->boundary = alloc_boundary_elm(1);
       rec->boundary->strlist= (STR_80 *) malloc (n * sizeof(STR_80));
       do {
                                /* Loop to obtain a member of the set and to
                                   store the member as the boundary list
                                   element */
          strcpy(str,p);
          while (strncmp(str," ",1)==0)strcpy(str, str+1);
          while (str[strlen(str)-1]== ' ') str[strlen(str)-1]='\0';
          strcpy(rec->boundary->strlist[i].str, str);
          p = strtok('\0', ",");
          i++;
       } while (p!=NULL);
    }
    else {
                                /* The following processes are accomplished
                                   if the value set is a set of numbers */

       rec->boundary = alloc_boundary_elm(2);
       rec->boundary->numlist= (float *) malloc (n * sizeof(float));
       do {
                                /* Loop to obtain a member of the set and to
                                   store the member as the boundary list
                                   element */

          rec->boundary->numlist[i]=atof(p);
          if (strcmp(rec->boundary->val_type,"N")==0 && strstr(p,".")!=NULL)
             strcpy(rec->boundary->val_type,"R");
          p = strtok('\0', ",");
          i++;
       } while (p!=NULL);
    }

    /*-----------------------------------------------------------------
     *   Set the number of the boundary list element
     */
    rec->boundary->n_elm = n;
}


/*****************************************************************************
*   VOID SET_SINGLE_VALUE()
*   This procedure is used to derive and record the value of the variable
*   stated in the predicate.
*****************************************************************************/
void set_single_value(rec)
TBL_REC *rec;
{

    if (atof(rec->pred_part[2])== 0 &&
        strcmp(rec->pred_part[2], "0")!=0) {
                                /* The value is a string */

       rec->boundary = alloc_boundary_elm(1);
       rec->boundary->strlist= (STR_80 *) malloc (sizeof(STR_80));
       strcpy(rec->boundary->strlist[0].str, rec->pred_part[2]);
    }
    else {
                                /* The value is a number */

       rec->boundary = alloc_boundary_elm(2);
```

```
        rec->boundary->numlist= (float *) malloc (sizeof(float));
        *rec->boundary->numlist= atof(rec->pred_part[2]);
        if (strcmp(rec->boundary->val_type,"N")==0 &&
            strstr(rec->pred_part[2],".")!=NULL)
           strcpy(rec->boundary->val_type,"R");
    }

    rec->boundary->n_elm = 1;     /* Set the element number of the boundary
                                     list */
}



/*****************************************************************************
*   VOID SET_RANGE_VALUES()
*   This procedure is used to derive and record the lower and/or upper
*   boundary  of the variable stated in the predicate
*****************************************************************************/
void set_range_values(rec)
TBL_REC *rec;
{
    char *p, str[80], lower_str[40], upper_str[40];
    float upper, lower;

    if (strcmp(rec->pred_part[1], ">")==0) {
                                /* Set the lower boundary if the comparison
                                   operator is ">" */

       rec->boundary = alloc_boundary_elm(3);
       rec->boundary->lower= atof(rec->pred_part[2])+0.000001;
       if (strstr(rec->pred_part[2],".")!=NULL)
          strcpy(rec->boundary->val_type,"R");
    }
    else
       if (strcmp(rec->pred_part[1], "<")==0) {
                                /* Set the upper boundary if the comparison
                                   operator is "<" */

          rec->boundary = alloc_boundary_elm(4);
          rec->boundary->upper= atof(rec->pred_part[2])-0.000001;
          if (strstr(rec->pred_part[2],".")!=NULL)
             strcpy(rec->boundary->val_type,"R");
       }
       else
          if (strcmp(rec->pred_part[1], ">=")==0) {
                                /* Set the lower boundary if the comparison
                                   operator is ">=" */

             rec->boundary = alloc_boundary_elm(3);
             rec->boundary->lower= atof(rec->pred_part[2]);
             if (strstr(rec->pred_part[2],".")!=NULL)
                strcpy(rec->boundary->val_type,"R");
          }
          else
             if (strcmp(rec->pred_part[1], "<=")==0) {
                                /* Set the upper boundary if the comparison
                                   operator is "<=" */

                rec->boundary = alloc_boundary_elm(4);
                rec->boundary->upper= atof(rec->pred_part[2]);
                if ( strstr(rec->pred_part[2],".")!=NULL)
                   strcpy(rec->boundary->val_type,"R");
             }
             else {

                                /* Observe the lower and upper boundary of
                                   the range stated in the third predicate
                                   part */
                strcpy(str, rec->pred_part[2]);
                p = strtok(str, "..");
                strcpy(lower_str, p);
                strcpy(str, rec->pred_part[2]);
```

```
                strcpy(upper_str, str+strlen(lower_str)+2);
                lower = atof(lower_str);
                upper = atof(upper_str);
                if ((lower != 0 || strcmp(lower_str,"0")==0) &&
                    (upper != 0 || strcmp(upper_str,"0")==0)) {
                                /* Set the lower and upper boundary if the
                                   third part of the predicate is a range of
                                   numbers */

                   rec->boundary = alloc_boundary_elm(5);
                   rec->boundary->lower= lower;
                   rec->boundary->upper= upper;
                   if (strstr(lower_str,".")!=NULL ||
                       strstr(upper_str,".")!=NULL)
                      strcpy(rec->boundary->val_type,"R");
                }
             }
}


/*****************************************************************************
*   BOUND_ELM *ALLOC_BOUNDARY_ELM()
*   This procedure is used to allocate and initialize a boundary element of a
*   record of the cause predicate table.
*****************************************************************************/
BOUND_ELM *alloc_boundary_elm(bound_type)
int bound_type;
{
    BOUND_ELM *new;

    new = (BOUND_ELM *) malloc (sizeof (BOUND_ELM));
    new->strlist = NULL;
    new->numlist = NULL;
    new->bound_type = bound_type;
    if (bound_type==1)
       strcpy(new->val_type,"STR");
    else
       strcpy(new->val_type,"N");/* Default numeric type */
    return(new);
}



/*****************************************************************************
*   VOID ADD_CAUSE_EFFECT_NODE()
*   This procedure is used to link a new or an existing cause/effect (predicate)
*   node with a new or an existing intermediate (operator) node of the cause-
*   effect graph.
*****************************************************************************/
void add_cause_effect_node(graph, pred, node_number, oper, add_node)
GRAPH *graph;
PRED  pred;
int   node_number, add_node;
OPER  oper;
{
    static GRPNODE *prev_inter=NULL, *last_cause=NULL,
                   *last_effect=NULL, *last_rightmost_inter=NULL,
                   *rightmost_inter=NULL;
                                /* Intermediate node that is connected to
                                   (an) effect node(s) */

    static SCOPELIST  *rmi_scope=NULL;
                                /* Right most intermediate node scope list */
    static PRED prev_pred;
    static OPER prev_oper;
    int  negation=0;
    GRPNODE *new_ce, *curr_ce, *new_inter, *curr_inter;


    /*-------------------------------------------------------------
```

```
 *   Set negation indicator
 */
if (node_number < 0) {
   node_number = -node_number;
   negation = 1;
}
/*-------------------------------------------------------------
 *   Determine the current cause/effect node to be linked
 */
if (add_node) {
                              /* A new node is required */

   new_ce = create_node(pred.type, node_number, 0);
   if (pred.type == CAUSE || pred.type== INV_CAUSE) {
                              /* Link the new node to the graph-cause
                                 link list */
      if (last_cause==NULL)
         graph->cause = new_ce;
      else
         last_cause->next = new_ce;
      last_cause = new_ce;
   }
   else {
                              /* Add the new node to the graph-effect
                                 link list */

      if (last_effect==NULL)
         graph->effect = new_ce;
      else
         last_effect->next = new_ce;
      last_effect = new_ce;
   }
   curr_ce = new_ce;
}
else {
                              /* Search an appropriate existing node to be
                                 linked */

   if (pred.type==CAUSE || pred.type==INV_CAUSE)
      curr_ce = graph->cause;
   else
      curr_ce = graph->effect;

   while (curr_ce->number != node_number) curr_ce = curr_ce->next;
}

/*-------------------------------------------------------------
 *   Determine the current intermediate node to be linked
 */
if (pred.type==INV_CAUSE || pred.type==CAUSE && oper.type!=NOOPER &&
    (last_cause==NULL || oper.type!=prev_oper.type ||
     prev_pred.type == EFFECT || prev_pred.type == INV_EFFECT ||
     pred.scope < prev_oper.scope)) {
                              /* A new intermediate node is required */

   (graph->ninter)++;
   new_inter = create_node(oper.type,graph->ninter,oper.scope);
   curr_inter = new_inter;
   if (rightmost_inter==NULL || prev_pred.type == EFFECT ||
       prev_pred.type == INV_EFFECT)
      rightmost_inter = curr_inter;
}
else
                              /* Use the previous node */

   curr_inter = prev_inter;


/*-------------------------------------------------------------
 *    Link the cause/effect and the intermediate nodes
 */
if (pred.type == CAUSE || pred.type == INV_CAUSE) {
```

```
                              /* Link a cause node */

   if (prev_inter != NULL && prev_inter != curr_inter &&
       prev_pred.type == CAUSE) {
                              /* The previous node is a cause node and
                                 the current and previous intermediate node
                                 are different */

      if (pred.scope == prev_inter->scope && prev_inter->type==AND){
                              /* Link the current cause node with the
                                 previous intermediate AND node; call a
                                 procedure to link the current intermediate
                                 node and the other intermediate nodes */

         mk_link(curr_ce, prev_inter, negation);
         place_inter_node(graph, prev_inter, &curr_inter, &rightmost_inter);
      }
      else {
                              /* Link the current cause node with the
                                 current intermediate node; Link the current
                                 intermediate node and the previous
                                 intermediate nodes */

         mk_link(curr_ce, curr_inter, negation);
         mk_link(curr_inter, prev_inter, 0);
      }
   }
   else
                              /* The previous and current intermediate node
                                 are the same */

      mk_link(curr_ce, curr_inter, negation);
}
else  {
                              /* Link an effect node */

   if (prev_pred.type==CAUSE || prev_pred.type==INV_CAUSE) {
                              /* The following are processes if the previous
                                 node is a cause node:
                                 - Add an element to the graph rightmost
                                   intermediate node list
                                 - Remove any ineffective previous
                                   intermediate AND node
                                 - Add an element to the rightmost
                                   intermediate scope list
                               */

      if (last_rightmost_inter==NULL)
         graph->rightmost_inter = rightmost_inter;
      else
         last_rightmost_inter->next = rightmost_inter;

      last_rightmost_inter = rightmost_inter;


      maintain_rightmost_inter_scplist(&rmi_scope, rightmost_inter);
      rightmost_inter->scope = prev_oper.scope;
      mk_link(rightmost_inter, curr_ce, negation);
                              /* Link the rightmost intermediate node and
                                 the effect node */
      if (rightmost_inter != prev_inter &&
          count_link_number(prev_inter->bakw)==1 &&
          count_link_number(prev_inter->forw)==1 &&
          prev_inter->type == prev_inter->forw->node->type) {
            rm_ineffect_inter(prev_inter);
            prev_inter = NULL;
      }
   }
   else
      mk_link(rightmost_inter, curr_ce, negation);
                              /* Link the rightmost intermediate node and
                                 the effect node */
```

```
    }

    /*-------------------------------------------------------------
     *    Check if this is the last call for this procedure.  If so,
     *    then call a procedure to arrange the rightmost intermediate
     *    node links.  Otherwise, set the working variables.
     */
    if (rmi_scope->head != rmi_scope->last && last_rightmost_inter->scope < 0 &&
        oper.scope==0 && oper.type==NOOPER)
       arrange_rightmost_inter_links(rmi_scope, graph);
    else {
       if (prev_inter != curr_inter)
          prev_inter = curr_inter;
       prev_oper.scope = oper.scope;
       prev_oper.type = oper.type;
       prev_pred.type = pred.type;
    }
}


/*****************************************************************************
*   GRPNODE* CREATE_NODE()
*   This procedure is used to allocate and initialize a new graph node.
*****************************************************************************/
GRPNODE* create_node(type, number, scope)
int type, number, scope;
{
    GRPNODE *new;

    new = (GRPNODE *) malloc (sizeof(GRPNODE));
    new->number = number;
    new->type = type;
    new->scope = scope;
    new->forw = NULL;
    new->bakw = NULL;
    new->next = NULL;
    return(new);
}


/*****************************************************************************
*   VOID MK_LINK()
*   This procedure is used to make two direction (forward and backward) links
*   between two graph nodes.
*****************************************************************************/
void mk_link(node_1, node_2, negation)
GRPNODE *node_1, *node_2;
int     negation;
{
    LINK *new_link, *curr;
    int i;

    /*-------------------------------------------------------------
     *   Make the forward link (from the node 1 to the node 2)
     */
    new_link = (LINK *) malloc (sizeof(LINK));
    new_link->negation = negation;
    new_link->next = NULL;
    if (node_1->forw == NULL)
       node_1->forw = new_link;
    else {
       curr = node_1->forw;
       while (curr->next != NULL) curr=curr->next;
       curr->next = new_link;
    }
    new_link->node = node_2;

    /*-------------------------------------------------------------
     *   Make the backward link (from the node 2 to the node 1)
     */
    new_link = (LINK *) malloc (sizeof(LINK));
    new_link->negation = negation;
```

```
    new_link->next = NULL;
    if (node_2->bakw == NULL)
       node_2->bakw = new_link;
    else {
       curr = node_2->bakw;
       while (curr->next != NULL) curr=curr->next;
       curr->next = new_link;
    }
    new_link->node = node_1;
}


/*****************************************************************************
*   VOID PLACE_INTER_NODE()
*   This procedure is used to arrange the links of a new intermediate node and
*   the other intermediate nodes.
*****************************************************************************/
void place_inter_node(graph, leftmost_inter, new, rightmost_inter)
GRAPH*  graph;
GRPNODE *leftmost_inter, **new, **rightmost_inter;
{
    GRPNODE *curr, *prev;

    /*---------------------------------------------------------------
     *   Search (an) intermediate node(s) in the graph to be linked with
     *   a new inter_node
     */
    curr = leftmost_inter;
    prev = NULL;
    while (curr->forw != NULL && (curr->scope < (*new)->scope ||
           curr->scope==(*new)->scope && (*new)->type==OR && curr->type==AND)){
                                /* Loop while the scope of the inter_node is
                                   greater than the scope of the search node
                                   and the next forward node is not NULL */
        prev = curr;
        curr = curr->forw->node;
    }

    if ((*new)->type==curr->type && (*new)->scope==curr->scope) {
                                /* The current and the new intermediate node
                                   have the same scope and type;  Free the
                                   new intermediate node */
       free(*new);
       *new = curr;
       graph->ninter--;
    }
    else {
                                /*   Insert the new inter_node */

       if (curr->scope > (*new)->scope) {
                                /* The current intermediate node scope is
                                   greater than the new inter mediate node
                                   scope */
          rm_forw_link(prev);
          mk_link(prev, *new, 0);
          mk_link(*new, curr, 0);
       }
       else {
                                /* The current intermediate node scope is
                                   less than the new intermediate node scope */
          if (curr->forw != NULL) {
             mk_link(*new, curr->forw->node, 0);
             rm_forw_link(curr);
          }
          mk_link(curr, *new, 0);
          if (curr==*rightmost_inter)
             *rightmost_inter = *new;
       }
    }
}
```

```
/*****************************************************************************
*   INT COUNT_LINK_NUMBER()
*   This procedure is used to count the number of one direction node links.
*****************************************************************************/
int count_link_number(link)
LINK *link;
{
    int  count=0;

    while (link != NULL) {
       ++count;
       link = link->next;
    }
    return(count);
}


/*****************************************************************************
*   VOID RM_INEFFECT_INTER()
*   This procedure is used to remove an ineffective intermediate node that only
*   be linked to one backward node and one forward node and the type of the
*   node and the forward node are the same.
*****************************************************************************/
void rm_ineffect_inter(node)
GRPNODE *node;
{
    LINK *flink, *blink;

    flink = node->forw;
    blink = node->bakw;
    mk_link(blink->node, flink->node, abs(blink->negation-flink->negation));
    free(flink);
    free(blink);
    free(node);
}



/*****************************************************************************
*   VOID MAINTAIN_RIGHTMOST_INTER_SCPLIST()
*   This procedure is used to add a new element to the rightmost intermediate
*   scope list.
*****************************************************************************/
void maintain_rightmost_inter_scplist(rmi_scope, rightmost_inter)
SCOPELIST **rmi_scope;
GRPNODE   *rightmost_inter;
{
    SCOPENODE *new_scp;

    /*-------------------------------------------------------------
     *   Allocate and initialize a new scopelist element
     */
    new_scp =(SCOPENODE *) malloc (sizeof(SCOPENODE));
    new_scp->scope = rightmost_inter->scope;
    new_scp->node = rightmost_inter;
    new_scp->next = NULL;

    /*
     *   Add the element to the end of the list
     */
    if (*rmi_scope == NULL) {
                           /* The list is NULL */

       *rmi_scope = (SCOPELIST *) malloc (sizeof(SCOPELIST));
       (*rmi_scope)->head = new_scp;
       (*rmi_scope)->last = (*rmi_scope)->head;
       new_scp->prev = NULL;
    }
    else {
                           /* The list is not NULL */

       (*rmi_scope)->last->next = new_scp;
```

```
        new_scp->prev = (*rmi_scope)->last;
        (*rmi_scope)->last = new_scp;
    }
}


/*****************************************************************************
*   VOID ARRANGE_RIGHTMOST_INTER_LINKS()
*   This procedure is used to arrange any link between any "global" rightmost
*   intermediate node with any effected rightmost intermediate node.  The right-
*   most intermediate scopelist that records the working scope relation among
*   the rightmost intermediate nodes is used to make this arrangement.
*****************************************************************************/
void arrange_rightmost_inter_links(rmi_scope, graph)
SCOPELIST *rmi_scope;
GRAPH     *graph;
{
    SCOPENODE *curr_sc, *prev_sc;
    int       done_scope, bkw_c_curr, bkw_c_higher;
    GRPNODE   *new_rmi, *curr_rmi;

    while (rmi_scope->head != NULL){
                          /* Loop until the scope list is empty  */

                          /* Set the working variables;  The current scope
                             element is the last element of the list. */
       curr_sc = rmi_scope->last;
       curr_rmi = curr_sc->node;
       done_scope = curr_sc->scope;
       prev_sc = curr_sc;

                          /* Process loop to find any previous scope list
                             element that effects the current element, i.e.,
                             any previous element that has greater scope
                             number than the current element and the previous
                             effect elements;  Based on their effect scopes,
                             link the intermediate nodes that are pointed to
                             by the current element with intermediate nodes
                             that are pointed to by the effect elements.  */
       do {

                          /*  Search the effect element */
          prev_sc = prev_sc->prev;
          while (prev_sc->scope <= done_scope) prev_sc = prev_sc->prev;
          if (prev_sc->scope != prev_sc->prev->scope)
             done_scope = prev_sc->scope;

                          /* Arrange the links necessary to connect the
                             rightmost intermediate node pointed to by the
                             current element and the rightmost intermediate
                             node pointed to by the effect element */
          bkw_c_curr=count_link_number(curr_rmi->bakw);
          if ((bkw_c_curr=count_link_number(curr_rmi->bakw)) > 1) {
                          /* The current rightmost intermediate node has more
                             than one backward links;  Inserts the new
                             rightmost intermediate node between the current
                             right most intermediate node and the effect
                             node(s) that previously is/are connected to the
                             current rightmost intermediate node; Replace the
                             current rightmost intermediate node by the new
                             rightmost intermediate node  */
             (graph->ninter)++;
             new_rmi=create_node(AND, graph->ninter, 0);
             cp_forw_link(new_rmi, curr_rmi);
             rm_forw_link(curr_rmi);
             mk_link(curr_rmi, new_rmi, 0);
             replace_rightmost_inter(curr_rmi, new_rmi, graph);
             curr_rmi = new_rmi;
          }

                          /* Link the (new) current rightmost intermediate
                             node and the effect node */
```

```
if ((bkw_c_higher=count_link_number(prev_sc->node->bakw)) == 1) {
                /* The rightmost intermediate node pointed to by the
                   effect element has only one backward link;  Copy
                   the forward links of the effect rightmost inter-
                   mediate node to the current rightmost intermediate
                   node;  If the forward node of the effect rightmost
                   intermediate node is not an effect invariant node,
                   copies the backward links of the effect rightmost
                   intermediate node to the current rightmost
                   intermediate node */

   if (prev_sc->node->forw->node->type!=INV_EFFECT)
      cp_forw_link(curr_rmi, prev_sc->node);
   cp_bakw_link(curr_rmi, prev_sc->node);
}
else {
                /* The rightmost intermediate node pointed to by
                   the effect element has more than one backward
                   link;  Link the current rightmost intermediate
                   node and the effect rightmost intermediate node;
                   Copy the forward links of the effect rightmost
                   intermediate node to the current rightmost
                   intermediate node */

   mk_link(prev_sc->node, curr_rmi, 0);
   cp_forw_link(curr_rmi, prev_sc->node);
}

if (prev_sc == curr_sc->prev) {
                /* The previous(effect) element is directly linked
                   to the current element  */
   if (bkw_c_higher == 1) {
                /* The rightmost intermediate node pointed to by
                   the effect element has only one backward link */
      if (prev_sc->node->forw->node->type!=INV_EFFECT) {
                /* The node pointed to by the forward link of the
                   effect rightmost intermediate node is not an
                   invariant cause node;  Removes the rightmost
                   intermediate node (all links of the effect
                   rightmost intermediate node have been copied
                   before) ;  Remove the forward link of the effect
                   rightmost intermediate node and remove the
                   element of the graph rightmost intermediate list
                   that links to the effect rightmost intermediate
                   node */
         rm_forw_link(prev_sc->node);
         rm_bakw_link(prev_sc->node);
         rm_rightmost_inter(prev_sc->node, graph);
         free(prev_sc->node);
      }
   }
   else {
                /* The rightmost intermediate node pointed to by
                   the effect element has more than one backward
                   link;  Remove the forward link of the effect
                   rightmost intermediate node and remove the
                   element of the graph rightmost intermediate node
                   list that links to the effect rightmost
                   intermediate node */
      rm_forw_link(prev_sc->node);
      rm_rightmost_inter(prev_sc->node, graph);
   }

                /* Remove the previous element  */
   if (prev_sc==rmi_scope->head) {
      rmi_scope->head = NULL;
      rmi_scope->last = NULL;
   }
   else  {
      prev_sc->prev->next = prev_sc->next;
      prev_sc->next->prev = prev_sc->prev;
   }
```

```
            free(prev_sc);

         }
      } while (rmi_scope->head != NULL && prev_sc != rmi_scope->head);

      /*-----------------------------------------------------------------
       *   Remove the current scope list element
       */
      if (rmi_scope->head != NULL)
         rmi_scope->last = curr_sc->prev;
      free(curr_sc);
   }
}


/*****************************************************************************
*   VOID CP_FORW_LINK()
*   This procedure is used to copy any forward link of a node to another node.
*****************************************************************************/
void cp_forw_link(d_node, s_node)
GRPNODE *d_node, *s_node;
{
    sub_cp_link(d_node, s_node, d_node->forw, s_node->forw);
}


/*****************************************************************************
*   VOID CP_BAKW_LINK()
*   This procedure is used to copy any backward link of a node to another node.
*****************************************************************************/
void cp_bakw_link(d_node, s_node)
GRPNODE *d_node, *s_node;
{
    sub_cp_link(d_node, s_node, d_node->bakw, s_node->bakw);
}

/*****************************************************************************
*   VOID SUB_CP_LINK()
*   This procedure is a subroutine called by "cp_forw_link" and "cp_bakw_link"
*   to copy one direction links of a node ("source node") to another node
*   ("destination node").
*****************************************************************************/
void sub_cp_link(d_node, s_node, d_link, s_link)
GRPNODE *d_node, *s_node;
LINK *s_link, *d_link;
{
    int forw=0;
    LINK *curr_s_link, *curr_d_link, *prev_d_link=d_link;

    /*-----------------------------------------------------------------
     *   Determine link direction
     */

    if ((curr_s_link=s_link) == s_node->forw)
       forw=1;

    /*-----------------------------------------------------------------
     *   The copying process
     */
    while (curr_s_link != NULL) {
                          /* Loop for all of the determined direction links
                             of the source node:
                             - Observe if the connection to the node linked
                               by the current source node link is already
                               exist at the destination node,
                             - Copy the current source link if the connection
                               does not exist. */

       for (curr_d_link=d_link; prev_d_link != NULL && curr_d_link != NULL;
                                              curr_d_link=curr_d_link->next)
           if (curr_s_link->node->number==curr_d_link->node->number &&
               curr_s_link->node->type == curr_d_link->node->type)
```

```
                        /* The connection is already exist at the
                           destination node */
            break;

        if (prev_d_link==NULL || curr_d_link==NULL)
                        /* The connection does not exist at the destination
                           node, copy the node link */
           if (forw)
              mk_link(d_node, curr_s_link->node, curr_s_link->negation);
            else
              mk_link(curr_s_link->node, d_node, curr_s_link->negation);

        curr_s_link = curr_s_link->next;
    }
}


/*****************************************************************************
*   VOID RM_FORW_LINK()
*   This procedure is used to remove all forward links of a node
*****************************************************************************/
void rm_forw_link(node)
GRPNODE *node;
{
    sub_rm_link(node, &(node->forw));
}


/*****************************************************************************
*   VOID RM_BAKW_LINK()
*   This procedure is used to remove all backward links of a node
*****************************************************************************/
void rm_bakw_link(node)
GRPNODE *node;
{
    sub_rm_link(node, &(node->bakw));
}


/*****************************************************************************
*   VOID SUB_RM_LINK()
*   This procedure is a subroutine called by "rm_forw_link" and "rm_bakw_link"
*   to remove all of the specified direction links of a node.
*****************************************************************************/
void sub_rm_link(node, first_link)
GRPNODE *node;
LINK **first_link;
{
    LINK *curr_in_link, *prev_in_link, *out_link;
    GRPNODE *l_node;
    int  forw=0;


    /*-------------------------------------------------------------
     *   Determine the link direction
     */

    if (*first_link==node->forw)
       forw=1;

    /*-------------------------------------------------------------
     *   Deletion process loop
     */
    while ((out_link= *first_link) != NULL) {

                        /* Search the pair link from the linked node to
                           the node */
       l_node = out_link->node;
       if (forw)
          curr_in_link = l_node->bakw;
       else
          curr_in_link = l_node->forw;
```

```
        prev_in_link = curr_in_link;

        while(curr_in_link->node != node) {
           prev_in_link = curr_in_link;
           curr_in_link = curr_in_link->next;
        }

                              /* Update the link list of the linked node */

        if (forw && curr_in_link == l_node->bakw)
           l_node->bakw = curr_in_link->next;
        else
           if (!forw && curr_in_link == l_node->forw)
              l_node->forw = curr_in_link->next;
           else
              prev_in_link->next = curr_in_link->next;

                              /* Free the link from the linked node to the node */
        free(curr_in_link);

                              /* Determine the node next link and free the node
                                 current link */
        *first_link = out_link->next;
        free(out_link);
     }
}


/*****************************************************************************
*   VOID REPLACE_RIGHTMOST_INTER()
*   This procedure is used to replace the position of the former rightmost
*   intermediate node in the graph rightmost intermediate node list with the
*   new (current) rightmost intermediate node.
*****************************************************************************/
void replace_rightmost_inter(old_rmi, new_rmi, graph)
GRPNODE *old_rmi, *new_rmi;
GRAPH *graph;
{
    GRPNODE *curr, *prev;

    /*-------------------------------------------------------------
     *   Search the old_rmi in the graph rightmost intermediate node list
     */
    for (curr=graph->rightmost_inter,prev=curr; curr!=old_rmi;
                                  prev=curr,curr=curr->next);

    /*-------------------------------------------------------------
     *   Replace the old node in the list with the new node
     */
    if (curr == graph->rightmost_inter)
       graph->rightmost_inter = new_rmi;
    else
       prev->next = new_rmi;
    new_rmi->next = old_rmi->next;

    /*-------------------------------------------------------------
     *   Remove the link of the old node to the next rightmost intermediate
     *   node
     */
    old_rmi->next = NULL;
}


/*****************************************************************************
*   VOID RM_RIGHTMOST_INTER()
*   This procedure is used to remove one rightmost intermediate node from the
*   graph rightmost intermediate node list.
*****************************************************************************/
void rm_rightmost_inter(rm_rmi, graph)
GRPNODE *rm_rmi;
GRAPH *graph;
{
```

```
    GRPNODE *curr, *prev;

    /*-------------------------------------------------------------
     *   Search the specified node in the graph rightmost intermediate node
     *   list
     */
    for (curr=graph->rightmost_inter,prev=curr; curr!=rm_rmi;
         prev=curr,curr=curr->next);

    /*-------------------------------------------------------------
     *   Disconnect the specified node from the graph rmi list
     */
    if (curr == graph->rightmost_inter)
       graph->rightmost_inter = rm_rmi->next;
    else
       prev->next = rm_rmi->next;
    rm_rmi->next = NULL;
}


/*****************************************************************************
*   CST_LIST *SEARCH_CONSTRAINTS()
*   This procedure is used to search any constraint between the specified cause
*   node and the other cause node.
*****************************************************************************/
CST_LIST *search_constraints(node_number, table)
int    node_number;
TABLE  *table;
{
    TBL_REC *curr, *prev;
    CST_LIST *cstlist=NULL;
    char  pred_part[80], dom_set[80], ran_set[80], dom_in[80], ran_in[80], *p;
    int   dom=0, ran=0, bound_cmp;


    /*-------------------------------------------------------------
     *   Get the specified number cause node
     */
    for (curr=table->cause; curr->number!=node_number; curr=curr->next);

    /*-------------------------------------------------------------
     *   Searching process
     */
    if (strstr(curr->pred_part[0], "mapsto") != NULL &&
        strncmp(curr->pred_part[1], "in", 2)==0 &&
        strncmp(curr->pred_part[2], "{", 1) != 0) {
                        /* The specified cause is a relation set membership
                           predicate:
                           - Find out if there are other causes which
                             specify that the set domain is the subset of
                             the relation domain,
                           - Find out if there are other causes which
                             specify that the set range is the subset of
                             the relation set, If so, call a procedure to
                             make a constraint relation between the
                             specified cause node and the other cause node
                             that specifies the membership of the relation
                             domain/range. */

                       /* Get the relation domain and range name */
       strcpy(pred_part, curr->pred_part[0]);
       p = strtok(pred_part, " ");
       strcpy(dom_in, p);
       p = strtok('\0', " ");
       p = strtok('\0', " ");
       strcpy(ran_in, p);

       for (prev=table->cause; prev != curr; prev=prev->next) {
                       /*  Process loop for all other cause node */
           if (dom && strcmp(prev->pred_part[0],dom_in)==0 &&
               strcmp(prev->pred_part[2],dom_set)==0 &&
```

```
            strcmp(prev->pred_part[1],"in")==0)
                        /* The set domain is the subset of the relation
                           domain */
               add_constraint_list(&cstlist, prev->number, R);
            else
               if (ran &&strcmp(prev->pred_part[0],ran_in)==0 &&
                   strcmp(prev->pred_part[2],ran_set)==0 &&
                   strcmp(prev->pred_part[1],"in")==0)
                        /* The set range is the subset of the relation
                           range */
                  add_constraint_list(&cstlist, prev->number, R);
               else
                  if (dom==0 && strncmp(prev->pred_part[0], "dom", 3)==0 &&
                      strstr(prev->pred_part[0], curr->pred_part[2]) != NULL &&
                      strncmp(prev->pred_part[1], "subset", 6)==0) {
                           /* This other cause node specifies the membership of
                              the set domain */
                     dom = 1;
                     strcpy(dom_set, prev->pred_part[2]);
                  }
                  else
                     if (ran==0 && strncmp(prev->pred_part[0], "ran", 3)==0 &&
                         strstr(prev->pred_part[0],curr->pred_part[2])!=NULL &&
                        strncmp(prev->pred_part[1], "subset", 6)==0) {
                           /* This other cause node specifies the membership of
                              the set range */
                        ran = 1;
                        strcpy(ran_set, prev->pred_part[2]);
                     }
      }
   }
   else {
                        /* The specified cause is not a relation set
                           membership predicate */

      if (curr->boundary != NULL)
                        /* If the specified cause specifies the boundary of
                           a variable, finds out if there is any other cause
                           node that also specifies the boundary of the
                           variable.  If so, add a constraint relation of
                           the specified cause node */
         for (prev=table->cause; prev != curr; prev=prev->next) {
            if (strcmp(curr->pred_part[0], prev->pred_part[0])==0) {
               bound_cmp = compare_boundary(curr->boundary, prev->boundary);
               if (bound_cmp != 0)
                  add_constraint_list(&cstlist, prev->number, bound_cmp);
            }
          }
   }
   return (cstlist);
}


/*****************************************************************************
*   VOID ADD_CONSTRAINT_LIST()
*   This procedure is used to add an element to the constraints list of a cause
*   node.
*****************************************************************************/
void add_constraint_list(cstlist, node_no, type)
CST_LIST **cstlist;
int node_no;
int      type;
{
   CST_ELM *new;

   new = (CST_ELM *) malloc (sizeof(CST_ELM));
   if (*cstlist==NULL) {
      *cstlist = (CST_LIST *) malloc (sizeof (CST_LIST));
      (*cstlist)->head = new;
      (*cstlist)->tail = new;
   }
```

```
    else {
       (*cstlist)->tail->next = new;
       (*cstlist)->tail = new;
    }
    new->node_no = node_no;
    new->type = type;
}


/*****************************************************************************
*   VOID COMPARE_BOUNDARY()
*   This procedure is used to compare two value boundaries of a variable.  This
*   procedure returns 0 if one of the boundaries is empty.  Otherwise, the
*   procedure returns the value of cause constraints (E, R, or RR).
*****************************************************************************/
int compare_boundary(bound1, bound2)
BOUND_ELM *bound1, *bound2;
{

    if (bound1==NULL || bound2==NULL) return(0);
                          /*   Return 0 if one of the boundaries is empty */

    /*-------------------------------------------------------
     *  Compare the two boundaries based on the type of the first boundary
     */
    switch (bound1->bound_type) {
       case 1: return(sub_compare_boundary_1(bound1, bound2));
       case 2: return(sub_compare_boundary_2(bound1, bound2));
       case 3: return(sub_compare_boundary_3(bound1, bound2));
       case 4: return(sub_compare_boundary_4(bound1, bound2));
       case 5: return(sub_compare_boundary_5(bound1, bound2));
       case 6: return(sub_compare_boundary_6(bound1, bound2));
    }
}


/*****************************************************************************
*   INT SUB_COMPARE_BOUNDARY_1()
*   This procedure is called by "compare_boundary" to compare a type 1 boundary
*   (boundary value is a single or a set of string(s)) with another boundary
*   of a variable.
*****************************************************************************/
int sub_compare_boundary_1(bound1, bound2)
BOUND_ELM *bound1, *bound2;
{

    /*-------------------------------------------------------
     *  Process the comparison based on the type of the second boundary
     */
    if (bound2->bound_type == 1) {
                          /* The second boundary is also the type 1 boundary */

       if (bound1->n_elm == bound2->n_elm)
                          /* The element number of the two boundary are the
                             same */
          if (search_strval(bound1, bound2))
                          /* The two boundaries are the same */
             return(RR);
          else
                          /* The two boundaries are not the same */
             return(E);
       else
          if (bound1->n_elm > bound2->n_elm)
                          /* The boundary 1 element number is greater than the
                             boundary 2 element number  */
             if (search_strval(bound2, bound1))
                          /* The boundary 2 is subset of the boundary 1 */
                return(-R);
             else
                          /* The boundary 2 is not subset of the boundary 1 */
```

```
                return(E);
            else
                            /* The boundary 1 element number is less than the
                               boundary 2 element number  */
                if (search_strval(bound1, bound2))
                            /* The boundary 1 is subset of the boundary 2 */
                    return(R);
                else
                            /* The boundary 1 is not subset of the boundary 2 */
                    return(E);
        }
        else
                            /* The second boundary is not the type 1 boundary */

            return(E);
}



/*****************************************************************************
*   INT SUB_COMPARE_BOUNDARY_2()
*   This procedure is called by "compare_boundary" to compare a type 2 boundary
*   (boundary value is a single or a set of number(s)) with another boundary
*   of a variable.
*****************************************************************************/
int sub_compare_boundary_2(bound1, bound2)
BOUND_ELM *bound1, *bound2;
{

    /*-----------------------------------------------------
     *  Process the comparison based on the type of the second boundary
     */
    if (bound2->bound_type == 2) {
                            /* The second boundary is also the type 2 boundary */
        if (bound1->n_elm == bound2->n_elm)
                            /* The element number of the two boundary are the
                               same */
            if (search_numval(bound1, bound2))
                            /* The two boundaries are the same */
                return(RR);
            else
                            /* The two boundaries are not the same */
                return(E);
        else
            if (bound1->n_elm > bound2->n_elm)
                            /* The boundary 1 element number is greater than the
                               boundary 2 element number  */
                if (search_numval(bound2, bound1))
                            /* The boundary 2 is subset of the boundary 1 */
                    return(-R);
                else
                            /* The boundary 2 is not subset of the boundary 1 */
                    return(E);
            else
                            /* The boundary 1 element number is less than the
                               boundary 2 element number  */
                if (search_numval(bound1, bound2))
                            /* The boundary 1 is subset of the boundary 2 */
                    return(R);
                else
                            /* The boundary 1 is not subset of the boundary 2 */
                    return(E);
    }
    else
        if (bound2->bound_type == 1)
                            /* The second boundary is the type 1 boundary */
            return (E);
        else
                            /* The other cases */
            if (search_numval(bound1, bound2))
                            /* The boundary 1 is subset of the boundary 2 */
```

```
            return (R);
         else
                        /* The boundary 1 is not subset of the boundary 2 */
            return (E);
}


/*****************************************************************************
*   INT SUB_COMPARE_BOUNDARY_3()
*   This procedure is called by "compare_boundary" to compare a type 3 boundary
*   (the lower boundary of integer or float numbers) with another boundary
*   of a variable.
*****************************************************************************/
int sub_compare_boundary_3(bound1, bound2)
BOUND_ELM *bound1, *bound2;
{

    /*-------------------------------------------------------
     *  Process the comparison based on the boundary and the value types
     */
    if (bound2->bound_type == 3 &&
        strcmp(bound1->val_type,bound2->val_type)==0) {
                        /* The second boundary is also the type 3 boundary */
       if (bound1->lower == bound2->lower)
                        /* The boundary 1 is the same as the boundary 2 */
          return(RR);
       else
                        /* The boundary 1 is not the same as the
                           boundary 2 */
          if (bound1->lower > bound2->lower)
                        /* The boundary 1 is greater than the boundary 2 */
             return(R);
          else
                        /* The boundary 1 is than the boundary 2 */
             return(-R);
    }
    else
       if (bound2->bound_type == 6) {
                        /* The second boundary is the type 6 boundary */
          if (search_numval(bound1, bound2))
                        /* The boundary 1 is subset of the boundary 2 */
             return (R);
          else
                        /* The boundary 1 is not subset of the boundary 2 */
             return(E);
       }
       else
                        /* The other cases */
          return(E);
}


/*****************************************************************************
*   INT SUB_COMPARE_BOUNDARY_4()
*   This procedure is called by "compare_boundary" to compare a type 4 boundary
*   (the upper boundary of integer or float numbers) with another boundary
*   of a variable.
*****************************************************************************/
int sub_compare_boundary_4(bound1, bound2)
BOUND_ELM *bound1, *bound2;
{

    /*-------------------------------------------------------
     *  Process the comparison based on the boundary and the value types
     */
    if (bound2->bound_type == 4 &&
        strcmp(bound1->val_type,bound2->val_type)==0) {
                        /* The second boundary is also the type 4 boundary */
       if (bound1->upper == bound2->upper)
                        /* The boundary 1 is the same as the boundary 2 */
          return(RR);
```

```
        else
           if (bound1->upper > bound2->upper)
                        /* The boundary 1 is greater than the boundary 2 */
              return(-R);
           else
                        /* The boundary 1 is less than the boundary 2 */
              return(R);
     }
     else
        if (bound2->bound_type == 2) {
                        /* The second boundary is the type 2 boundary */
           if (search_numval(bound2, bound1))
                        /* The boundary 2 is subset of the boundary 1 */
              return (-R);
           else
                        /* The boundary 2 is not subset of the boundary 1 */
              return(E);
        }
        else
           if (bound2->bound_type == 6) {
                        /* The second boundary is the type 6 boundary */
              if (search_numval(bound1, bound2))
                        /* The boundary 1 is subset of the boundary 2 */
                 return (R);
              else
                        /* The boundary 1 is not subset of the boundary 2 */
                 return(E);
           }
           else
                        /* The other cases */
              return(E);
}


/*****************************************************************************
*    INT SUB_COMPARE_BOUNDARY_5()
*    This procedure is called by "compare_boundary" to compare a type 5 boundary
*    (the range of integer or float numbers) with another boundary
*    of a variable.
*****************************************************************************/
int sub_compare_boundary_5(bound1, bound2)
BOUND_ELM *bound1, *bound2;
{

     /*-----------------------------------------------------
      *  Process the comparison based on the boundary and the value types
      */
     if (bound2->bound_type == 2) {
                        /* The second boundary is the type 2 boundary */
        if (search_numval(bound2, bound1))
                        /* The boundary 2 is subset of the boundary 1 */
           return (-R);
        else
                        /* The boundary 2 is not subset of the boundary 1 */
           return (E);
     }
     else
        if (bound2->bound_type == 5 &&
            strcmp(bound1->val_type,bound2->val_type)==0) {
                        /* The second boundary is also the type 6 boundary */
           if (bound1->lower == bound2->lower)  {
                        /* The lower boundary of the boundary 1 is the same
                           as the lower boundary of the boundary 2 */
              if (bound1->upper == bound2->upper)
                        /* The upper boundary of the boundary is the same
                           as the upper boundary of the boundary 2 */
                 return(RR);
              else
                 if (bound1->upper > bound2->upper)
                        /* The upper boundary of the boundary 1 is greater
                           than the upper boundary of the boundary 2 */
                    return(-R);
```

```
                else
                        /* The upper boundary of the boundary 1 is less than
                           the upper boundary of the boundary 2 */
                    return(R);
            }
            else
                if (bound1->lower > bound2->lower) {
                        /* The lower boundary of the boundary 1 is greater
                           than the lower boundary of the boundary 2 */
                    if (bound1->upper <= bound2->upper)
                        /* The upper boundary of the boundary 1 is less than
                           than or the same as the upper boundary of the
                           boundary 2 */
                        return(R);
                    else
                        /* The upper boundary of the boundary 1 is greater
                           than the upper boundary of the boundary 2 */
                        return(E);
                }
                else {
                        /* The lower boundary of the boundary 1 is less than
                           the lower boundary of the boundary 2 */
                    if (bound1->upper >= bound2->upper)
                        return(-R);
                    else
                        return(E);
                }
        }
        else
            if (bound2->bound_type == 6) {
                        /* The second boundary is the type 6 boundary */
                if (search_numval(bound1, bound2))
                        /* The boundary 1 is subset of the boundary 2 */
                    return (R);
                else
                        /* The boundary 1 is not subset of the boundary 2 */
                    return(E);
            }
            else
                        /* The other cases  */
                return(E);
}


/*****************************************************************************
*   INT SUB_COMPARE_BOUNDARY_6()
*   This procedure is called by "compare_boundary" to compare a type 6 boundary
*   (the boundary is specified only by a numeric type) with another boundary
*   of a variable.
*****************************************************************************/
int sub_compare_boundary_6(bound1, bound2)
BOUND_ELM *bound1, *bound2;
{

    /*-------------------------------------------------------
     *  Process the comparison based on the type of the second boundary
     */
    switch (bound2->bound_type) {
        case 1:             /* The boundary 2 is the type 1 boundary */
                return(E);

        case 6:             /* The boundary 2 is also the type 6 boundary */
                if (strcmp(bound1->val_type, bound2->val_type)==0)
                    return(RR);
                else
                    if (bound1->val_type[0]=='R')
                        return(-R);
                    else
                        if (bound2->val_type[0]=='R')
                            return(R);
                        else
                            if (bound1->val_type[0]=='Z')
```

```
                        return(-R);
                    else
                        if (bound2->val_type[0]=='Z')
                            return(R);
                        else
                            if (bound1->val_type[1]=='1')
                                return(R);
                            else
                                return(-R);

        default:            /* The other cases */
                if (search_numval(bound2, bound1))
                            /* The boundary 2 is subset of the boundary 1 */
                    return (-R);
                else
                            /* The boundary 2 is not subset of the boundary 1 */
                    return (E);
    }
}


/*****************************************************************************
*   INT SEARCH_STRVAL()
*   This procedure is used to search all strings of the boundary 1 stringlist
*   in the boundary 2 stringlist. If all strings are found, then the procedure
*   returns 1.  Otherwise, the procedure returns 0.
*****************************************************************************/
int search_strval(bound1, bound2)
BOUND_ELM *bound1, *bound2;
{
    int i, j;

    for (i=0; i < bound1->n_elm; i++) {
                        /* Loop for (all) strings of the boundary 1 */
        for (j=0; j < bound2->n_elm; j++)
                        /* Loop for (all) strings of the boundary 2 */
            if (strcmp(bound1->strlist[i].str, bound2->strlist[j].str)==0)
                        /* The boundary 1 string is found in the boundary
                           2 */
                break;
        if (j==bound2->n_elm)
                        /* The string of the boundary 1 is not in the
                          boundary 2 */
            return(0);
    }
    return(1);
}



/*****************************************************************************
*   INT SEARCH_NUMVAL()
*   This procedure is used to search all numeric boundary values specified by
*   the boundary 1 in the numeric boundary values specified by the boundary 2.
*   If the boundary 1 numeric values are subset of the boundary 2 numeric
*   values, then the procedure returns 1.  Otherwise, the procedure returns 0.
*****************************************************************************/
int search_numval(bound1, bound2)
BOUND_ELM *bound1, *bound2;
{
    int i, j, low_bound2, up_bound2;

    if (bound2->bound_type==6) {
                        /* The boundary 2 is the type 6 boundary */

       if (bound2->val_type[0]=='R')
                        /* The numeric type of the boundary 2 is "R" (real),
                           the boundary 1 is subset of the boundary 2 */
          return (1);
       else
          if (bound1->val_type[0] != 'R') {
                        /* The numeric type of the boundary 1 is not "R" */
```

```
                    /* Derive the lower boundary of the boundary 2 */
        if (bound2->val_type[0] != 'N')
           low_bound2 = 0;
        else
           low_bound2 = 1;

        if (bound1->bound_type==2) {
                    /* The boundary 1 is the type 2 boundary */

                    /* Check if the numerical list of the boundary 1 is
                       greater than the lower boundary of the
                       boundary 2 */
           for (i=0; i < bound1->n_elm; i++)
               if (bound1->numlist[i] < low_bound2)
                  return(0);
           return(1);
        }
        else
           if (bound1->bound_type==3 || bound1->bound_type==5)
                    /* The boundary 1 is the type 3 or 5 boundary */

                    /* Check if the lower of the boundary 1 is greater
                       than the lower boundary of the boundary 2 */
              if (bound1->lower < low_bound2)
                 return(0);
              else
                 return(1);
           else
                    /* The boundary 1 is the type 4 boundary */
              return(0);
     }
     else
                    /* The numeric type of the boundary 1 is "R" */
        return(0);
}
else
   if (bound1->bound_type==2) {
                    /* The boundary 2 is the type 6 boundary */

      if ((bound1->val_type[0]=='R' || bound2->val_type[0]=='R') &&
          bound1->val_type[0]!=bound2->val_type[0])
                    /* The numeric type of one boundary is "R" */
         return(0);
      else
                    /* The numeric type of two boundaries are "R" or are
                       not "R" */

         if (bound2->bound_type==2) {
                    /* The boundary 2 is the type 2 boundary */

                    /* Check if all list values in the boundary 1 are
                       also in the boundary 2 */
            for (i=0; i < bound1->n_elm; i++){
                for (j=0; j < bound2->n_elm; i++)
                if (bound1->numlist[i] == bound2->numlist[j])
                   break;
                if (j==bound2->n_elm)
                   return(0);
            }
            return(1);
         }
         else
            if (bound2->bound_type==3) {
                    /* The boundary 2 is the type 3 boundary */

                    /* Check if all list values of the boundary 1 are
                       greater than the lower boundary of the boundary
                       2 */
               for (i=0; i < bound1->n_elm; i++)
                   if (bound1->numlist[i] < bound2->lower)
                      return(0);
```

```
                return(1);
            }
            else
                if (bound2->bound_type==4) {
                        /*  The boundary 2 is the type 4 boundary */

                        /*  Check if all list values of the boundary 1 are
                            less than the upper boundary of the boundary 2 */
                    for (i=0; i < bound1->n_elm; i++)
                        if ( bound1->numlist[i] > bound2->upper)
                            return(0);
                    return(1);
                }
                else
                    if (bound2->bound_type==5) {
                        /*  The boundary 2 is the type 5 boundary */

                        /*  Check if all list values of the boundary 1 are in
                            the boundary range of the boundary 2 */
                      for (i=0; i < bound1->n_elm; i++)
                          if (bound1->numlist[i] < bound2->lower ||
                              bound1->numlist[i] > bound2->upper)
                              return(0);
                      return(1);
                    }
        }
        else
                        /* The other cases */
           return(0);

}


/*****************************************************************************
*   VOID ADD_CONSTRAINT_NODE_LINK()
*   This procedure is used to link the existing or new created cause node with
*   a cause node and any other related cause node.  The node constraint list
*   that records the constraint relation between the cause node and the other
*   cause node is used as the process driver.
*****************************************************************************/
void add_constraint_node_link(constraint_list, graph)
CST_LIST *constraint_list;
GRAPH *graph;
{
    CST_ELM *cst_elm;
    int     curr_node, exist_code=4;
    GRPNODE *curr_ce, *prev_ce, *curr_cst;
    static GRPNODE *last_cst=NULL;
    static int cst_no=0;

    /*-------------------------------------------------------------
     *   Search for the current cause node
     */
    curr_node = graph->ncause;
    for (curr_ce=graph->cause; curr_ce->number != curr_node;
         curr_ce=curr_ce->next);

    /*-------------------------------------------------------------
     *   Process for all constraint list elements
     */
    for (cst_elm=constraint_list->head; cst_elm!=NULL; cst_elm=cst_elm->next) {

                        /* Search for the related cause node */
        for (prev_ce=graph->cause; prev_ce->number!=cst_elm->node_no;
            prev_ce=prev_ce->next);

                        /* Search for the existing constraint node for this
                           constraint relation.  Create new constraint node
                           if such node does not exist yet */
        curr_cst=search_existing_cst(graph, curr_ce, prev_ce, cst_elm->type,
                                     &exist_code);
```

```
        if (curr_cst==NULL) {
           cst_no++;
           curr_cst = create_node(abs(cst_elm->type), cst_no, 0);
           if (last_cst==NULL)
              graph->constraint = curr_cst;
           else
              last_cst->next = curr_cst;
           last_cst = curr_cst;
        }

                           /* Link the cause node(s) and the constraint node if
                              necessary */
        if (exist_code != 3)
           if (exist_code==0)
                           /* The constraint node is a new created node */
              if (cst_elm->type==-R) {
                 mk_link(curr_cst, curr_ce, 0);
                 mk_link(curr_cst, prev_ce, 0);
              }
              else {
                 mk_link(curr_cst, prev_ce, 0);
                 mk_link(curr_cst, curr_ce, 0);
              }
           else
              if (exist_code==1)
                           /* The link between the related cause node and the
                              existing constraint node already exists */
                 mk_link(curr_cst, curr_ce, 0);
              else
                           /* The link between the cause node and the existing
                              constraint node already exists */
                 mk_link(curr_cst, prev_ce, 0);
    }
}


/*****************************************************************************
*   GRPNODE* SEARCH_EXISTING_CST()
*   This procedure is used to search the existing constraint node and links for
*   a specific constraint relation.  If the constraint node for the relation
*   does not exist yet, the procedure returns NULL.
*****************************************************************************/
GRPNODE *search_existing_cst(graph, curr_ce, prev_ce, cst_type, exist_code)
GRAPH *graph;
GRPNODE *curr_ce, *prev_ce;
int cst_type, *exist_code;
{
    GRPNODE *curr_cst;
    LINK    *link;
    int     prev=0, curr=0;

    /*-------------------------------------------------------------
     *   Searching process
     */
    for (curr_cst=graph->constraint; curr_cst!=NULL; curr_cst=curr_cst->next) {
                           /* Loop for all of the existing constraint nodes  */

        if (curr_cst->type==abs(cst_type))
                           /* The current existing constraint type is the same
                              or the negation of the specified constraint
                              type */

           if (cst_type==R) {
                           /* The specified constraint type is "R"  */
              if (curr_cst->forw->node->number==prev_ce->number) {
                           /* The link between the existing constraint node and
                              the related cause node already exists */
                 prev=1;
                 break;
              }
           }
```

```
            else
               if (cst_type==-R) {
                           /* The specified constraint type is "-R"  */
                  if (curr_cst->forw->node->number==curr_ce->number) {
                           /* The link between the existing constraint node and
                           the current cause node already exists */
                     curr=1;
                     break;
                  }
               }
               else {
                           /* The constraint type is "E"  */

                           /* Search for the link between the constraint node
                              and the related cause node or the current cause
                              node */
                  for (link=curr_cst->forw; link!=NULL; link=link->next) {
                      if (link->node->number==curr_ce->number)
                         curr=1;
                      if (link->node->number==prev_ce->number)
                         prev=1;
                  }
                  if (curr || prev)
                     break;
               }
    }

                           /* Set the variable to record the existing link(s)
                              finding */
    if (prev && curr)
       *exist_code=3;
    else
       if (prev)
          *exist_code=1;
       else
          if (curr)
             *exist_code=2;
          else
             *exist_code=0;

    return(curr_cst);
}




/*****************************************************************************
*   VOID REMOVE_INTER_NODE_DUPLICATION()
*   This procedure is used remove any intermediate node duplication.
*****************************************************************************/
void remove_inter_node_duplication(graph)
GRAPH *graph;
{
    GRPNODE *cause;
    LINK    *link, *next_link;

    /*-------------------------------------------------------------
     *   Compare the backward links of two intermediate nodes, remove one node
     *   if the links of the two nodes are the same
     */
    for (cause=graph->cause; cause->next!=NULL; cause=cause->next)
                           /* Loop for all of the graph cause nodes */

        for (link=cause->forw; link->next!=NULL; link=link->next)
                           /* Loop for all intermediate nodes that are connected
                              to the cause node  */

                           /* Search for the next intermediate node that has
                              similar backward links as the current intermediate
                              node */
```

```
            for (next_link=link->next; next_link!=NULL;
                 next_link=next_link->next)
               if (compare_two_inter(link->node, next_link->node)) {
                           /* The backward links of the two nodes are the
                              same  */
                  cp_forw_link(link->node, next_link->node);
                  rm_forw_link(next_link->node);
                  rm_bakw_link(next_link->node);
                  free(next_link->node);
               }
}


/*****************************************************************************
*   INT COMPARE_TWO_INTER()
*   This procedure is used to compare the backward links of two intermediate
*   nodes.  The procedure returns 1 is the backward links of the two nodes are
*   the same.   Otherwise, the procedure returns 0.
*****************************************************************************/
int compare_two_inter(inter_1, inter_2)
GRPNODE *inter_1, *inter_2;
{
    LINK *link_1, *link_2;

    if (count_link_number(inter_1->bakw) != count_link_number(inter_2->bakw))
                           /*  Return 0 if the number of the backward links of
                               the two node are not the same   */
       return (0);

    /*-------------------------------------------------------------
     *   Compare the backward links of the two nodes
     */
    for (link_1=inter_1->bakw; link_1 != NULL; link_1=link_1->next) {
                           /* Loop for all of the node 1 links  */

                           /* Search for a link of the node 2 that the same
                              as the node 1 link */
        for (link_2=inter_2->bakw; link_2!=NULL; link_2=link_2->next)
            if (link_1->node->number==link_2->node->number &&
                link_1->node->type == link_2->node->type &&
                link_1->negation == link_2->negation)
                           /* A link of the node 2 is similar to the node 1
                              link  */
               break;
        if (link_2==NULL)
                           /* No link of the node 2 is the same as the node 1
                              link */
           return(0);
    }
    return(1);
}




/*****************************************************************************
*   VOID PRINT_GRAPH()
*   This procedure is used to print a graph forward traversals starting from
*   the graph cause nodes, the graph backward traversals starting from the
*   graph effect nodes, and the cause constraint relations.
*****************************************************************************/
void print_graph(graph)
GRAPH *graph;
{
    GRPNODE *node;
    int     c, i;
    LINK *link;

    printf("The graph forward traversals starting from a cause node: \n");
    printf("========================================================= ");
```

```
    node = graph->cause;
    while (node != NULL) {
        printf("\n\n");
        c = 0;
        print_graph_forw(node, &c, 0);
        node = node->next;
    }
    printf("\n\n");
    printf("** Node representation: [negation/-](<node type>,");
    printf(" <sequence number>)\n");

    printf("\n\n\n");
    printf("The graph backward traversals starting from an effect node: \n");
    printf("=============================================================");
    node = graph->effect;
    while (node != NULL) {
        printf("\n\n");
        c = 0;
        print_graph_bakw(node, &c, 0);
        node = node->next;
    }
    printf("\n\n");
    printf("** Node representation: [negation/-](<node type>,");
    printf(" <sequence number>)\n");

    if ((node = graph->constraint)!=NULL) {
       printf("\n\n\n");
       printf("The graph cause constraint connections: \n");
       printf("=======================================");
       while (node != NULL) {
           printf("\n\n");
           c = 0;
           print_graph_node(node, &c, 0);
           printf("(");
           c++;
           for (link=node->forw, i=0; link!=NULL; link=link->next, i++) {
               if (i>1){
                  printf(",");
                  c++;
               }
               print_graph_node(link->node, &c, 0);
           }
           printf(")");
           node = node->next;
       }
       printf("\n\n");
       printf("** Node representation: [negation/-](<node type>,");
       printf(" <sequence number>)\n");
    }
}


/*****************************************************************************
*   VOID PRINT_GRAPH_FORW()
*   This recursive procedure is called by "print_graph" to prints all forward
*   nodes of one cause node.
*****************************************************************************/
void print_graph_forw(node, c, negation)
GRPNODE *node;
int *c;
int negation;
{
    LINK *link;
    int i=0;

    print_graph_node(node, c, negation);
    if ((link = node->forw)!=NULL) {
       if (*c > 57) {
          printf("\n");
          *c = 0;
       }
       printf("(");
```

```
        (*c)++;
        while (link != NULL) {
            i++;
            if (i > 1) {
               printf(",");
               (*c)++;
            }
            print_graph_forw(link->node, c, link->negation);
            link = link->next;
        }
        printf(")");
        (*c)++;
    }
}


/*****************************************************************************
*   VOID PRINT_GRAPH_BAKW()
*   This recursive procedure is called by "print_graph" to prints all backward
*   nodes of one effect node.
*****************************************************************************/
void print_graph_bakw(node, c, negation)
GRPNODE *node;
int *c;
int negation;
{
    LINK *link;
    int i=0;

    print_graph_node(node, c, negation);
    if ((link = node->bakw)!=NULL) {
       if (*c > 57) {
          printf("\n");
          *c = 0;
       }
       printf("(");
       (*c)++;
       while (link != NULL) {
           i++;
           if (i > 1) {
              printf(",");
              (*c)++;
           }
           print_graph_bakw(link->node, c, link->negation);
           link = link->next;
       }
       printf(")");
       (*c)++;
    }
}


/*****************************************************************************
*   VOID PRINT_GRAPH_NODE()
*   This procedure is called by "print_graph_forw"  and "print_graph_bakw"
*   to print a graph node.
*****************************************************************************/
void print_graph_node(node, c, negation)
GRPNODE *node;
int *c;
int negation;
{

    if (*c > 58) {
       printf("\n");
       *c = 0;
    }
    if (negation) {
       printf("-");
       (*c)++;
    }
    printf("(");
```

```
    switch (node->type) {
       case AND         :  printf("AND");
                           *c = *c + 6;
                           break;
       case OR          :  printf("OR");
                           *c = *c + 5;
                           break;
       case CAUSE       :  printf("CAUSE");
                           *c = *c + 8;
                           break;
       case INV_CAUSE   :  printf("INV_CAUSE");
                           *c = *c + 11;
                           break;
       case EFFECT      :  printf("EFFECT");
                           *c = *c + 9;
                           break;
       case INV_EFFECT  :  printf("INV_EFFECT");
                           *c = *c + 13;
                           break;
       case R           :  printf("R");
                           *c = *c + 4;
                           break;
       case E           :  printf("E");
                           *c = *c + 4;
                           break;
       default          :  printf("%d",node->type);
                           *c = *c + 5;
    }

    printf(",%d)", node->number);
    if (node->number < 10)
       *c = *c + 1;
    else
       if (node->number < 100)
          *c = *c + 2;
       else
          *c = *c + 3;

}


/*****************************************************************************
*   TEST_LIST* DERIVE_TEST_FRAME()
*   This procedure is used to derive test frames from a cause-effect graph. The
*   derivation is done by setting a value of an intermediate node that is linked
*   to (an) effect node(s) (the rightmost intermediate node - rmi) and determi-
*   ning all other node values by tracing the graph backward and forward. The
*   process is repeated for all rmi nodes. The derivation can be done by using
*   two kind of setting values, i.e., "1" for the true-effect test frame  deri-
*   vation and "0" for the false-effect test frame derivation.
*****************************************************************************/
TEST_LIST *derive_test_frame(graph, set_val)
GRAPH *graph;
int   set_val;
{
    TEST_LIST *new_l, *total_l=NULL;
    GRPNODE   *curr_rmi;
    TEST_FRAME *new_f;

    /*---------------------------------------------------------------
     *   Reset the computation counters
     */
    compute_0 = 0;
    compute_1 = 0;

    /*---------------------------------------------------------------
     *   Derivation processes
     */
    for (curr_rmi=graph->rightmost_inter; curr_rmi != NULL;
         curr_rmi=curr_rmi->next) {
                        /* Loop for all of the rightmost intermediate
```

```
                        nodes */

        if (curr_rmi->forw->node->type != INV_EFFECT){
                        /* Derive test frames if the node is not a rmi of the
                           invariant node  */

           if (total_l==NULL) {
                        /* Total test frame list is NULL  */

              total_l=trace_one_rightmost_inter_backward(curr_rmi, graph,
                                                         set_val);
              new_l = total_l;
           }
           else {
                        /* Total test frame list is not NULL  */

              new_l->next = trace_one_rightmost_inter_backward(curr_rmi, graph,
                                                               set_val);
              new_l = new_l->next;
           }
           derive_other_rightmost_inter_values(new_l, curr_rmi, graph, set_val);
           remove_test_frame_duplication(new_l, total_l, graph);
           derive_effect_values(new_l, graph);
        }
    }
    return(total_l);
}


/*****************************************************************************
*   TEST_LIST* TRACE_ONE_RIGHTMOST_INTER_BACKWARD()
*   This procedure is used to derive cause node values for the specific
*   rightmost intermediate node value by tracing the graph backward
*   from the rightmost intermediate node.
*****************************************************************************/
TEST_LIST *trace_one_rightmost_inter_backward(rmi_node, graph, set_val)
GRPNODE *rmi_node;
GRAPH   *graph;
int     set_val;
{
    TEST_LIST *new_l;
    TEST_FRAME *new_f;

    new_l = alloc_test_list();
    new_f = add_test_frame(new_l, graph);
    new_f->inter[rmi_node->number-1] = set_val;
    trace_one_node_backward(rmi_node, new_l, graph);
    return(new_l);
}


/*****************************************************************************
*   TEST_LIST* ALLOC_TEST_LIST()
*   This procedure is used to allocate a new test frame list.
*****************************************************************************/
TEST_LIST *alloc_test_list()
{
    TEST_LIST *new;

    new = (TEST_LIST *) malloc (sizeof(TEST_LIST));
    new->head = NULL;
    new->tail = NULL;
    new->next = NULL;
    return(new);
}


/*****************************************************************************
*   TEST_LIST* ADD_TEST_FRAME()
*   This procedure is used to add a new frame to a specified test frame list.
*****************************************************************************/
TEST_FRAME *add_test_frame(test, graph)
```

```
TEST_LIST *test;
GRAPH *graph;
{
    TEST_FRAME *new;
TEST_FRAME *curr;
    int i;

    /*-------------------------------------------------------------
     *   Allocate  a new frame and initiates  the value of the new frame
     *   elements
     */
    new = (TEST_FRAME *) malloc (sizeof(TEST_FRAME));
    new->cause = (int *) malloc (graph->ncause * sizeof(int));
    new->inter = (int *) malloc (graph->ninter * sizeof(int));
    new->derive = (int *) malloc (graph->ninter * sizeof(int));
    new->effect = (int *) malloc (graph->neffect * sizeof(int));
    new->next = NULL;
    for (i=0;i < graph->ncause; i++) new->cause[i]=-1;
    for (i=0;i < graph->ninter; i++){
        new->inter[i]=-1;
        new->derive[i]=1;
    }
    for (i=0;i < graph->neffect; i++) new->effect[i]=-1;

    /*-------------------------------------------------------------
     *    Add the new frame to the specified list
     */
    if (test->head==NULL) {
       test->head = new;
       test->tail = new;
    }
    else {
       test->tail->next = new;
       test->tail = new;
    }
    return(new);
}



/*****************************************************************************
*   VOID COPY_TEST_FRAME()
*   This procedure is used to copy the element values of one test frame to
*   another test frame.
*****************************************************************************/
void copy_test_frame(graph, r_frame, s_frame)
GRAPH *graph;
TEST_FRAME *r_frame, *s_frame;
{
    int i, j;
    TEST_FRAME *curr_frame;

    for (i=0;i < graph->ncause; i++)
           r_frame->cause[i]= s_frame->cause[i];
    for (i=0;i < graph->ninter; i++) {
        r_frame->inter[i]= s_frame->inter[i];
        r_frame->derive[i]= s_frame->derive[i];
    }
    for (i=0;i < graph->neffect; i++)
        r_frame->effect[i]= s_frame->effect[i];
}


/*****************************************************************************
*   VOID TRACE_ONE_NODE_BACKWARD()
*   This recursive procedure is used to derive the value of a specific node
*   inputs and trace all of the input nodes backward.
*****************************************************************************/
void trace_one_node_backward(node, test, graph)
GRPNODE *node;
TEST_LIST *test;
GRAPH *graph;
```

```
{
    LINK  *link;

    if ((node->type == AND || node->type == OR) && test->head!=NULL) {
                          /* Counduct tracing if the specified node is not a
                             cause node */
       derive_node_input_values(node, test, graph);
       for (link = node->bakw; link != NULL; link = link->next)
                          /* Trace all input nodes backward  */
          trace_one_node_backward(link->node, test, graph);
    }
}


/*****************************************************************************
*   VOID DERIVE_NODE_INPUT_VALUES()
*   This procedure is used to derive the possible combinations of a node
*   input values that lead the node output to be as specified in a test frame.
*   If there are more than one input combinations, (a) new test frame(s) is/
*   are created in attempt to record all input combinations. If the original
*   test frame "derivation code" is "1", some of the derivation rules are
*   follows. For an OR node, if the node output is TRUE (=1), consider only
*   combinations with only one TRUE input; if the node output is FALSE, consi-
*   der all combinations of input values that lead the node output to be FALSE.
*   For an AND node, if the node output is FALSE, consider all possible combi-
*   nations of input. For the last case, the derivation status of a test frame
*   that record a combination of all false inputs is set to "0".  If the origin-
*   test frame derivation code is "0", only one combination of input values is
*   considered.
*****************************************************************************/
void derive_node_input_values(node, test, graph)
GRPNODE *node;
TEST_LIST *test;
GRAPH *graph;
{
    TEST_FRAME *curr, *prev;
    TEST_LIST *add_test=NULL;
    LINK *link;
    int success;

    add_test = alloc_test_list();
                          /* Allocate a temporary (working) test list */

    if (node->type == AND)
                          /* The node is an AND intermediate node  */

                          /* Conduct the input value derivation for all test
                             frames */
       for (curr=test->head; curr != NULL; ) {
           prev = curr;

           if (curr->derive[node->number-1] > 0) {
                          /* The derivation for this node is allowed */

              if (curr->inter[node->number-1]==1) {
                          /* The node output is TRUE  */

                          /* Set all input values to be "1"  */
                 for (link = node->bakw, success=1; link != NULL && success;
                                                    link = link->next)
                     success=derive_one_input_node_val(graph, curr, link, 1);

                 if (!success)
                          /* At least one input value determination is not
                             successfully conducted (conflict with the existing
                             input value) */
                    curr=delete_test_frame(test, curr);
              }
              else
                          /* The node output is FALSE  */

                 if (derive_false_AND_input_values(curr, node, graph,
```

```
                                                        add_test)== 0)
                        /* Delete the current test frame if the derivation
                           for all zero input is not successfully completed */
                    curr=delete_test_frame(test, curr);

            }
                        /* Determine the next test frame to be processed */
            if (curr==prev)
               curr=curr->next;
         }
      else
                        /* The node is an OR intermediate node  */

                        /* Conduct the input value derivation for all test
                           frames */
         for (curr=test->head; curr != NULL; ) {
            prev = curr;

            if (curr->derive[node->number-1] > 0) {
                        /* The derivation for this node is allowed */

               if (curr->inter[node->number-1]==0) {
                        /* The node output is FALSE  */

                        /* Set all input values to be "0"  */
                  for (link = node->bakw, success=1; link != NULL && success;
                                                        link = link->next)
                      success=derive_one_input_node_val(graph, curr, link, 0);

                  if (!success)
                     curr=delete_test_frame(test, curr);
               }
               else
                        /* The node output is TRUE  */

                  if (derive_true_OR_input_values(curr, node,
                                                graph, add_test)== 0)
                     curr=delete_test_frame(test, curr);
            }
            if (curr==prev)
               curr=curr->next;
         }

      if (add_test->head != NULL) {
                        /* Add the temporary list to the test list, if the
                           temporary list is not empty */
         if (test->head==NULL)
            test->head = add_test->head;
         else
            test->tail->next = add_test->head;
         test->tail = add_test->tail;
         add_test->head = NULL;
         add_test->tail = NULL;
      }
      free(add_test);
}

/*****************************************************************************
*    INT DERIVE_ONE_INPUT_NODE_VAL()
*    This procedure is used to derive the value of an input node by combining
*    the given input value and the value of the link that is connected  to the
*    input node.    If the value of the node in the specified test frame has been
*    set and the value is conflict with the combination result, then the proce-
*    dure returns 0.    If the determination is successfully completed, then
*    the procedure returns 1.    If the input node is a cause node and the node
*    has (a) constraint relation(s), the the procedure calls a procedure to
*    deteminer the value of an/the effected node(s).
*****************************************************************************/
int derive_one_input_node_val(graph, test_f, link, in_val)
GRAPH *graph;
TEST_FRAME *test_f;
LINK *link;
```

```
int  in_val;
{
    int rc;
    TEST_LIST *temp_l;
    TEST_FRAME *temp_f;
    GRPNODE *node=link->node;

    compute_1++;            /* Increase the computation counter by 1 */
    if (node->type==INV_CAUSE) {
                            /* The input node is a cause invariant node */
       if (test_f->cause[node->number-1]==-1) {
          test_f->cause[node->number-1] = 1;
          if (abs(in_val-link->negation)==0) {
                            /* The calculating node value is conflict with the
                               nature of an invariant  */
             rc=0;
          }
       }
       else
          rc=1;
    }
    else
       if (node->type==CAUSE) {
                            /* The input node is a cause node */
          if (test_f->cause[node->number-1]==-1) {
             test_f->cause[node->number-1]=abs(in_val-link->negation);
             if (node->bakw != NULL && (test_f->cause[node->number-1]==1 ||
                 (test_f->cause[node->number-1]==0 && node->bakw->node->type==R
                  && node->bakw->node->forw->node==node))) {
                            /* The input node is a cause node and has at least
                               one constraint link */

                            /* Create temporary test list and frame  */
                temp_l = alloc_test_list();
                temp_f = add_test_frame(temp_l, graph);
                copy_test_frame(graph, temp_f, test_f);

                if ((rc=derive_effected_node_values(node, temp_f)))
                            /* Examine if the existing constrained node values
                               do not conflict with any constraint tranformation
                             */
                   copy_test_frame(graph, test_f, temp_f);
                else
                   test_f->cause[node->number-1]=0;
                            /* Delete temporary test list and frame  */
                delete_test_frame(temp_l, temp_f);
                free(temp_l);
             }
             else
                rc=1;
          }
          else
             if (test_f->cause[node->number-1]!=
                 abs(in_val-link->negation)) {
                            /* The calculating node value is conflict with the
                               existing node value */
                rc=0;
             }
             else
                rc=1;
       }
       else {               /* The input node is an intermediate node */
          test_f->inter[node->number-1] = abs(in_val-link->negation);
          rc = 1;
       }
    return(rc);
}

/*****************************************************************************
*   INT  DERIVE_EFFECTED_NODE_VALUES()
*   This recursive procedure is used to derive the value of the other cause
```

```
*   node(s) that is/are effected by the constraint relation with the specified
*   cause node.
*****************************************************************************/
int derive_effected_node_values(node, test_f)
GRPNODE *node;
TEST_FRAME *test_f;
{
    LINK *link, *c_link;
    int  rc=1;

    for (link=node->bakw; rc && link!=NULL; link=link->next) {
                        /* Loop for all of the specified node constraint
                           links  */
        c_link=link->node->forw;
        if (link->node->type==R && c_link->node != node &&
            test_f->cause[node->number-1]==1) {
                        /* The node requires another node; If the
                           node value is "1", then the required cause
                           node must be "1"  */
           if (test_f->cause[c_link->node->number-1]==-1) {
              test_f->cause[c_link->node->number-1]=1;
              rc=derive_effected_node_values(c_link->node, test_f);
           }
           else
              if (test_f->cause[c_link->node->number-1]==0)
                 rc=0;
              else
                 rc=derive_effected_node_values(c_link->node, test_f);
        }
        else
           if ((link->node->type==R && test_f->cause[node->number-1]==0 &&
               c_link->node==node) || (link->node->type==E &&
               test_f->cause[node->number-1]==1))
                        /* If the constraint type is E and the node value is
                           "1" or the constraint type is R and the node is
                           required by the other nodes and the node value is
                           "0", then the other cause nodes must be "0"  */
              for (;rc && c_link!=NULL;c_link=c_link->next)
                  if (c_link->node != node)
                     if (test_f->cause[c_link->node->number-1]==-1) {
                        test_f->cause[c_link->node->number-1]= 0;
                        rc=derive_effected_node_values(c_link->node, test_f);
                     }
                     else
                        if (test_f->cause[c_link->node->number-1]==1)
                           rc=0;
                        else
                           rc=derive_effected_node_values(c_link->node, test_f);
    }
    return(rc);
}

/*****************************************************************************
*   INT EXAMINE_FORWARD_CONFLICT()
*   This recursive procedure is used to examine whether the forward transforma-
*   tion of a node value is conflict with the forward node value or not. If the
*   values are conflict then the procedure returns 1;  Otherwise, the procedure
*   returns 0.
*****************************************************************************/
int examine_forward_conflict(test_f, node, node_val, node_0)
TEST_FRAME *test_f;
GRPNODE *node, *node_0;
int node_val;
{
    LINK *link;
    GRPNODE *f_node;
    int     f_node_in, rc;

    /*-------------------------------------------------------------
     *   Search for the forward node
     */
    for (link=node->forw; link != NULL && (f_node=link->node)!=node_0  &&
         test_f->inter[f_node->number-1]==-1; link=link->next);
```

```
/*-------------------------------------------------------------
 *   Examination process
 */
if (link==NULL || f_node == node_0) {
                        /* The forward node is the originator AND node that
                           changed the derivation code to "0"  or is not set
                         */
   if (link==NULL)
      rc=0;
   else
      rc=1;
}
else {
                        /* The forward node is not the originator AND node
                           that changed the derivation code to "0"  */

                        /* Examine the conflict */
   if (f_node->type==OR)
                        /* The forward node is an OR node  */

      if (test_f->inter[f_node->number-1]==0) {
                        /* The forward node value is 0  */

         test_f->inter[f_node->number-1]=1;
         rc=examine_forward_conflict(test_f, f_node, 1, node_0);
      }
      else {
                        /* The forward node is 1; Check if the value of the
                           forward node is not effected by the change of the
                           node value */

         for (link=f_node->bakw; link != NULL; link=link->next) {
             if (link->node->type==AND || link->node->type==OR)
                f_node_in=test_f->inter[link->node->number-1];
             else
                f_node_in=test_f->cause[link->node->number-1];

             if (f_node_in==-1 || abs(f_node_in-link->negation)==1)
                break;
         }

         if (link==NULL) {
            test_f->inter[f_node->number-1]=0;
            rc=examine_forward_conflict(test_f, f_node, 0, node_0);
         }
         else
            rc=0;
      }
   else
                        /* The forward node is an AND node  */

      if (test_f->inter[f_node->number-1]==1) {
                        /* The forward node value is 1  */

         test_f->inter[f_node->number-1]=0;
         rc=examine_forward_conflict(test_f, f_node, 0, node_0);
      }
      else {
                        /* The forward node is 0; Check if the value of the
                           forward node is not effected by the change of the
                           node value */

         for (link=f_node->bakw; link != NULL; link=link->next) {
             if (link->node->type==AND || link->node->type==OR)
                f_node_in=test_f->inter[link->node->number-1];
             else
                f_node_in=test_f->cause[link->node->number-1];

             if (f_node_in==-1 || abs(f_node_in-link->negation)==0)
                break;
         }
```

```
            if (link==NULL) {
               test_f->inter[f_node->number-1]=1;
               rc=examine_forward_conflict(test_f, f_node, 1, node_0);
            }
            else
               rc=0;
         }
    }
    return(rc);
                        /* Return the examination result */
}


/*****************************************************************************
*   INT DERIVE_FALSE_AND_INPUT_VALUES()
*   This  procedure is used to derive the combinations of the input values of
*   a false AND node and the possible input node values for each combination.
*****************************************************************************/
int derive_false_AND_input_values(curr, node, graph, add_test)
TEST_FRAME *curr;
GRPNODE *node;
TEST_LIST *add_test;
GRAPH *graph;
{
    LINK *link;
    TEST_FRAME *new;
    int  i, j, nlink, nc=1, *val, success;

    /*-------------------------------------------------------------
     *   Determine the number of combinations
     */
    for (nlink=0, link=node->bakw; link != NULL; nlink++, link=link->next) {
        nc = nc * 2;
    }

    /*-------------------------------------------------------------
     *   Allocate working variables and initiate the variable values
     */
    val = (int *) malloc (nlink * sizeof(int));
    for (i=0; i < nlink; i++) val[i] = 0;

    /*-------------------------------------------------------------
     *   The derivation for all input value combination but the first:
     *   Set the input values of each combination and derive the value of
     *   the input nodes and store them in a new test frame.  If the determi-
     *   nations are not succesfully completed, delete the new test frame.
     */
    for (i=2; i < nc; i++) {
                        /* Loop to process all combinations but the first */

        new=add_test_frame(add_test, graph);
        copy_test_frame(graph, new, curr);

        for (j=nlink-1; j >= 0 ; j--)
                        /* Loop to set input values of a combination */
            if (val[j]==0) {
               val[j] = 1;
               break;
            }
            else
               val[j] = 0;

        for (link=node->bakw, j=0, success=1; link != NULL && success;
             link=link->next, j++)
                        /* Loop to derive the value of the input nodes */
            if (val[j] && (success=derive_one_input_node_val(graph, new,
                link, val[j])))
               new->derive[link->node->number-1] = 2;

        if (!success)
                        /* Delete the new test frame if the determinations
                           are not successfully completed */
```

```
            delete_test_frame(add_test, new);
    }

    /*-------------------------------------------------------------
     *   The derivation of the first combination (using the input test frame)
     */
     return (set_zeroes_AND_input_values(graph,curr,node));
}


/*****************************************************************************
*   INT DERIVE_TRUE_OR_INPUT_VALUES()
*   This  procedure is used to derive the combinations of the input values of a
*   true OR node and the possible input node values for each combination.
*****************************************************************************/
int derive_true_OR_input_values(curr, node, graph, add_test)
TEST_FRAME *curr;
GRPNODE *node;
TEST_LIST *add_test;
GRAPH *graph;
{
    LINK *link;
    TEST_FRAME *new;
    int  i, j, nlink, *val, success;


    /*-------------------------------------------------------------
     *   Determine the number of node inputs
     */
    for (nlink=0, link=node->bakw; link != NULL; nlink++, link=link->next);

    /*-------------------------------------------------------------
     *   Allocate the working variables and initiate the working values
     */
    val = (int *) malloc (nlink * sizeof(int));
    for (i=0; i < nlink; i++) val[i] = 0;

    /*-------------------------------------------------------------
     *   The derivation for all input value combination but the first
     */
    for (i=1; i < nlink; i++) {

                          /* Create a new test frame and initiate the frame
                             element values */
        new=add_test_frame(add_test, graph);
        copy_test_frame(graph, new, curr);

                          /* Set the combination input values */
        val[i] = 1;
        val[i-1] = 0;

        for (link=node->bakw, j=0, success=1; link!=NULL && success;
             link=link->next, j++)
                          /* Loop to derive the value of input nodes  */
            success=derive_one_input_node_val(graph, new, link, val[j]);

        if (!success)
                          /* Delete the new test frame if the determinations
                             are not successfully completed */
           delete_test_frame(add_test, new);
    }

    /*-------------------------------------------------------------
     *   The derivation of the first combination (using the input test frame)
     */
    val[0] = 1;
    val[nlink-1] = 0;
    for (link=node->bakw, j=0, success=1; link != NULL && success;
         link=link->next, j++)
                          /* Loop to derive the input node values of the
                             first combination */
        success=derive_one_input_node_val(graph, curr, link, val[j]);
```

```
    if (success)
                            /* The determinations for the first combination are
                               succeeded */
       return(1);
    else
                            /* The determinations for the first combination are
                               not succeeded */
       return(0);
}


/*****************************************************************************
*   INT SET_ZEROES_AND_INPUT_VALUES()
*   This  procedure is used to set all zero input values and the cause value
*   combinations of a false AND node.
*****************************************************************************/
int  set_zeroes_AND_input_values(graph, test_f, node)
GRAPH *graph;
TEST_FRAME *test_f;
GRPNODE *node;
{
    LINK *link;
    CC_LIST *cclist=NULL;
    int rc;


    for (link=node->bakw, rc=1; rc && link!=NULL; link=link->next) {
                            /* Loop for all input nodes;  Call a recursive
                               procedure to set initial value of the cause nodes
                             */
        if (set_zeroes_AND_cause_values(test_f, link, 0, node)==0)
           rc=0;
    }
    if (rc) {
                            /* The initial cause value setting is successfully
                               completed; Search for constrained causes that
                               effect the AND node; If the constrained causes
                               are found, then reset the value of the nodes for
                               neglecting the later value conflict with the
                               related node(s).  */
       search_constrained_causes(&cclist, node);
       if (cclist != NULL)
          rc=reset_constrained_cause_values(graph, test_f, cclist, node);
    }
    return(rc);
}


/*****************************************************************************
*   VOID SEARCH_CONSTRAINED_CAUSES()
*   This recursive procedure is used to search for constrained causes that
*   effect a false AND node.
*****************************************************************************/
void search_constrained_causes(cclist, node)
CC_LIST **cclist;
GRPNODE *node;
{
    LINK *link;
    CC_ELM *new;

    if (node->type==CAUSE || node->type==INV_CAUSE) {
                            /* The specified node is a cause node */
       if (node->bakw!=NULL) {
                            /* The node has at least one constraint relation;
                               Add an element to the constrained cause list */
          new = (CC_ELM *) malloc (sizeof(CC_ELM));
          if (*cclist==NULL) {
             *cclist = (CC_LIST *) malloc (sizeof (CC_LIST));
             (*cclist)->head = new;
             (*cclist)->tail = new;
          }
          else {
```

```
            (*cclist)->tail->next = new;
            (*cclist)->tail = new;
         }
         new->node=node;
         new->next=NULL;
      }
   }
   else
      for (link=node->bakw; link!=NULL; link=link->next)
                        /* Call this procedure recursively for all backward
                           nodes */
          search_constrained_causes(cclist, link->node);
}



/*****************************************************************************
*   INT SET_ZEROES_AND_CAUSE_VALUES()
*   This  recursive procedure is used to set the initial value of cause nodes
*   that effect all input of an AND node to be zeroes.
*****************************************************************************/
int set_zeroes_AND_cause_values(test_f,link,val, node_0)
TEST_FRAME *test_f;
LINK *link;
int  val;
GRPNODE *node_0;
{
   LINK *blink;
   int rc=1;

   compute_0++;            /* Increase the computation counter by 1 */

   if (link->node->type==INV_CAUSE) {
                        /* The linked node is a cause node;  Check if the
                           transformed AND node value is not conflict with
                           the existing/mandatory cause value;  If the
                           values are conflict, then call the procedure to
                           examine whether the change of the cause node will
                           change the affected AND input value or not.  */
      test_f->cause[link->node->number-1]=1;
      if (abs(link->negation-val) != 1)
         if (examine_forward_conflict(test_f, link->node, 1, node_0))
            rc=0;
   }
   else
      if (link->node->type==CAUSE) {
         if (test_f->cause[link->node->number-1]!=-1 &&
             test_f->cause[link->node->number-1]!=abs(link->negation-val)){
         if (examine_forward_conflict(test_f, link->node,
                           test_f->cause[link->node->number-1], node_0))
            rc=0;
         }
         else
            test_f->cause[link->node->number-1]=abs(link->negation-val);
      }
      else {
                        /* The linked node is an intermediate node;  Set
                           the intermediate node value and call this
                           procedure recursively for all backward links */
         test_f->derive[link->node->number-1]=0;
         test_f->inter[link->node->number-1]=abs(link->negation-val);

         if (link->node->type==OR) {

            for (blink=link->node->bakw, rc=0; blink!=NULL; blink=blink->next){
                if (set_zeroes_AND_cause_values(test_f, blink,
                           test_f->inter[link->node->number-1], node_0)==1)
                rc=1;
            }
         }
         else
```

```
            for (blink=link->node->bakw, rc=1; rc && blink!=NULL;
                                                    blink=blink->next) {
                if (set_zeroes_AND_cause_values(test_f, blink,
                                test_f->inter[link->node->number-1], node_0)==0)
                   rc=0;
            }
        }

    return(rc);
}


/*****************************************************************************
*   INT RESET_CONSTRAINED_CAUSE_VALUES()
*   This procedure is used to reset the value of the constrained cause values
*   that effect a zeroes AND node inputs to neglect the later value conflict
*   with the related node(s).  If the cause nodes are connected to E constraint,
*   then tries to reset the cause values to 0 (if the initial values is 1);
*   If the node is constrained by an R constraint, then tries to reset the
*   cause node to 1 (if the initial cause value is 0);  If the node effects
*   a related R constrained node, then tries to reset the cause value to 0
*   (if the initial value is 1).
*****************************************************************************/
int reset_constrained_cause_values(graph, test_f, cclist, node_0)
GRAPH *graph;
TEST_FRAME *test_f;
CC_LIST *cclist;
GRPNODE *node_0;
{

    CC_ELM *cc_elm;
    TEST_LIST  *new_l, *temp_l;
    TEST_FRAME *new_f, *curr_f, *prev_f;
    LINK   *link;
    int  et_val, rf_val, rt_val;


    /*-------------------------------------------------------------
     *   Create a new and a temporary test list
     */
    new_l = alloc_test_list();
    temp_l = alloc_test_list();
    new_f = add_test_frame(new_l, graph);
    copy_test_frame(graph, new_f, test_f);

    /*-------------------------------------------------------------
     *   Reset the constrained cause values;  If the constrained cause value
     *   is 1 and the node has E constraint or R0 (requires) constraint, then
     *   reset the value to 0.   If the constrained cause value is 0 and the
     *   node has R1 (required) constraint, then the value is reset to 1.
     */
    for (cc_elm = cclist->head; cc_elm != NULL; cc_elm=cc_elm->next) {
                            /* Loop for all constrained cause nodes */

                            /* Determine the constraint types */
        et_val=0;
        rf_val=0;
        rt_val=0;

        for (link=cc_elm->node->bakw; link!=NULL; link=link->next) {
                            /* Loop for all constraint nodes linked to the cause
                               node */

            if (link->node->type==R) {
               if (link->node->forw->node==cc_elm->node)
                  rt_val=1;
               else
                  rf_val=1;
            }
            else
               et_val=1;
        }
```

```
                    /* Resetting process */
    for (curr_f=new_l->head; curr_f!=NULL;) {
                    /* Loop for all test frames */

        if (curr_f->cause[cc_elm->node->number-1] == 0) {
                    /* The cause value is 0 */

           if (rt_val) {

                    /* Create a test frame variation */
              new_f = add_test_frame(temp_l, graph);
              copy_test_frame(graph, new_f, curr_f);
              new_f->cause[cc_elm->node->number-1] = 1;

                    /* Examine if the existing constrained node value
                       does not conflict with the false AND input node
                       value */
              if (!examine_forward_conflict(new_f , cc_elm->node, 1,
                                                          node_0)) {
                 if (!et_val && !rf_val) {
                    copy_test_frame(graph, curr_f, new_f);
                    delete_test_frame(temp_l, new_f);
                 }
              }
              else
                 delete_test_frame(temp_l, new_f);
           }
        }
        else
                    /* The cause value is 1 */

           if (et_val || rf_val) {

                    /* Create a test frame variation */
              new_f = add_test_frame(temp_l, graph);
              copy_test_frame(graph, new_f, curr_f);
              new_f->cause[cc_elm->node->number-1] = 0;

                    /* Examine if the existing constrained node value
                       does not conflict with the false AND input node
                       value */
              if (!examine_forward_conflict(new_f , cc_elm->node, 0,
                                                          node_0)) {
                 if (!rt_val) {
                    copy_test_frame(graph, curr_f, new_f);
                    delete_test_frame(temp_l, new_f);
                 }
              }
              else
                 delete_test_frame(temp_l, new_f);
           }

                    /* Add any valid new test frame and determine the
                       next test frame to be processed */
        if (temp_l->head!=NULL) {
                    /* Insert the test frame variation into the original
                       test list  */
           new_f->next = curr_f->next;
           curr_f->next = new_f;
           temp_l->head=NULL;
           curr_f = new_f->next;
        }
        else
           curr_f = curr_f->next;
    }
}
free(temp_l);

/*-------------------------------------------------------------
 *   Examine whether the constraint transformation of the reset cause
```

```
     *   values conflict with the existing related cause node values.  If so,
     *   delete the test frame contains the conflict values.
     */
    for (curr_f=new_l->head; curr_f!=NULL;) {
                          /* Loop for all test frames */

        prev_f = curr_f;

        for (cc_elm=cclist->head; cc_elm!=NULL;cc_elm=cc_elm->next)
                          /* Loop for all constrained cause nodes */

                          /* Examine if the existing constrained node values
                             do not conflict with any constraint transformation
                           */
            if (curr_f->cause[cc_elm->node->number-1] == 1 &&
                derive_effected_node_values(cc_elm->node, curr_f) == 0) {
               curr_f = delete_test_frame(new_l, curr_f);
               break;
            }

         if (curr_f==prev_f)
            curr_f=curr_f->next;
    }

    /*-------------------------------------------------------------------
     *   Delete the constrained cause list
     */
    while(cclist->head!=NULL) {
        cc_elm = cclist->head;
        cclist->head=cc_elm->next;
        free(cc_elm);
    }
    free(cclist);


    /*-------------------------------------------------------------------
     *   If there are valid test frames, then get one of them and returns 1.
     *   Otherwise, return 0.
     */
    if (new_l->head!=NULL) {
       copy_test_frame(graph, test_f, new_l->head);

       while (new_l->head!=NULL)
           delete_test_frame(new_l, new_l->head);
       free(new_l);
       return(1);
    }
    else {
       free(new_l);
       return(0);
    }
}


/*****************************************************************************
*   VOID DERIVE_OTHER_RIGHTMOST_INTER_VALUES()
*   This  procedure is used to derive the other rightmost intermediate values.
*****************************************************************************/
void derive_other_rightmost_inter_values(test_l, curr_rmi, graph, set_val)
TEST_LIST *test_l;
GRPNODE   *curr_rmi;
GRAPH *graph;
int set_val;
{
    GRPNODE    *other_rmi;
    TEST_FRAME *curr_f, *next_f;
    int        i, rmi_val, default_val;

    /*-----------------------------------------------------------
     *   Process the derivation for all test frames
     */
```

```
    for (curr_f = test_1->head; curr_f != NULL; curr_f = curr_f->next) {

                          /* Derive all other rmi; The default value of
                             the nodes are the negation of the current rmi
                           */
        for (other_rmi=graph->rightmost_inter; other_rmi!=NULL;
             other_rmi=other_rmi->next)
            if (other_rmi != curr_rmi &&
                other_rmi->forw->node->type != INV_EFFECT) {
               default_val = 1-set_val;
               curr_f->inter[other_rmi->number-1]=
                  derive_one_rightmost_inter_val( curr_f, other_rmi,
                                                  default_val, graph);
            }
    }
}


/*****************************************************************************
*   VOID DERIVE_ONE_RIGHTMOST_INTER_VAL()
*   This  recursive procedure is used to derive the value of one rightmost
*   intermediate node other than the test driver node.  The derivation is first
*   derivation cannot give the (rmi) node value, the other (rmi) node is set to
*   default value and the cause node values are then derived by propagating
*   the value backward.
*****************************************************************************/
int derive_one_rightmost_inter_val(test_f, node, default_val, graph)
TEST_FRAME *test_f;
GRPNODE    *node;
int        default_val;
GRAPH      *graph;
{
    int bkwnode_val;
    LINK *link, *prev_link;
    GRPNODE *bkwnode;
    static int first_try = 1;

    /*-----------------------------------------------------------------
     *   Derive the node value by propagating the existing node values
     *   forward
     */
    for (link=node->bakw; first_try; link=link->next) {
                          /* Loop for all backward links  */

                          /* Get the value of a backward node */
        if ((bkwnode=link->node)->type==CAUSE ||
            (bkwnode=link->node)->type==INV_CAUSE)
                          /* The link is connected to a cause node;  Get the
                             node value */
           bkwnode_val=test_f->cause[bkwnode->number-1];
        else
                          /* The link is connected to an intermediate node;
                             Calls a procedure to derive the node value, if
                             the node value is not set yet  */
           if ((bkwnode_val=test_f->inter[bkwnode->number-1])==-1)
              bkwnode_val=derive_one_rightmost_inter_val(test_f, bkwnode,
                                abs(default_val-link->negation), graph);
           else
              bkwnode_val=test_f->inter[bkwnode->number-1];

                          /* Examine the derivation result */
        if (node->type==AND && abs(bkwnode_val-link->negation)==0 ||
            node->type==OR && abs(bkwnode_val-link->negation)==1)
                          /* The backward node value can be used to determine
                             the node value */
           return (abs(bkwnode_val-link->negation));

        if (link->next==NULL)
           first_try = 0;
    }

    /*-----------------------------------------------------------------
```

```
     *   Set the node value to default value;  Derive any related
     *   cause node value
     */
    for (link=node->bakw; link != NULL; link=link->next) {
                        /* Loop for all backward links  */
        if ((bkwnode=link->node)->type==CAUSE ||
            (bkwnode=link->node)->type==INV_CAUSE) {
                        /* The linked node is a cause node;  Set the node
                           value if the node value is not set yet */
           if (test_f->cause[bkwnode->number-1]==-1) {
              bkwnode_val = abs(default_val-link->negation);
              test_f->cause[bkwnode->number-1]=bkwnode_val;
              if (bkwnode->bakw!=NULL)
                 derive_effected_node_values(bkwnode, test_f);
           }
        }
        else
                        /* The linked node is an intermediate node;  Call
                           this procedure to set the node value, if the node
                           value is not set yet  */
           if (test_f->inter[bkwnode->number-1]==-1) {
              bkwnode_val = abs(default_val-link->negation);
              test_f->inter[bkwnode->number-1]=bkwnode_val;
              derive_one_rightmost_inter_val(test_f, bkwnode,
                              bkwnode_val, graph);
           }
    }
    first_try = 1;
    return (default_val);
}


/*****************************************************************************
*   VOID REMOVE_TEST_FRAME_DUPLICATION()
*   This procedure is used to remove any test frame duplication in the test
*   frame list.
*****************************************************************************/
void remove_test_frame_duplication(curr_l, test, graph)
TEST_LIST *curr_l, *test;
GRAPH *graph;
{
    TEST_LIST *other_l=NULL;
    TEST_FRAME *other_f, *curr_f, *prev_f, *prev_other_f;
    int  i, j, curr_dntcare, other_dntcare;

    for (other_l=test; other_l != NULL; other_l=other_l->next)
                        /*  Loop for all test lists  */

        for (curr_f = curr_l->head; curr_f != NULL;) {
                        /*  Loop for all test frame in the current test
                            list */

            prev_f = curr_f;
            for (other_f=other_l->head; other_f!=NULL && other_f != curr_f;){
                        /* Loop for all other test frames before the current
                           test frame */

                        /* Compare the current frame with the other test
                           frame cause node values */
               prev_other_f = other_f;

               for (i=0, j=0; i < graph->ninter; i++) {
                        /* Loop for all intermediate nodes  */
                   if (other_f->inter[i] != curr_f->inter[i])
                        /* The cause node value of the two frames are not
                           the same */
                      break;
                   if (other_f->derive[i]==2 && curr_f->derive[i]==2)
                        /* The derivation value of the two frames are
                           the same and equal to two */
                      j=1;
```

```
                }
                if (i==graph->ninter && j) {
                        /* The two frames are identical  */
                   other_f=delete_test_frame(other_l, other_f);
                   break;
                }

                curr_dntcare = 0;
                other_dntcare = 0;
                for (i=0; i < graph->ncause; i++) {
                        /* Loop for all cause nodes  */

                    if (other_f->cause[i] != curr_f->cause[i])
                        /* The cause node value of the two frames are not
                           the same */
                       if (other_f->cause[i]==-1) {
                           /* The node value of the other frame is not set */
                           other_dntcare++;
                          if (curr_dntcare != 0)
                            break;
                       }
                       else
                          if (curr_f->cause[i]==-1) {
                           /* The node value of the current frame is not
                              set */
                             curr_dntcare++;
                             if (other_dntcare != 0)
                                break;
                          }
                          else
                             break;
                }

                        /* Examine the comparison result;  If the frames
                           are identical, delete the frame that has smaller
                           number of unset values. */
                if (i == graph->ncause) {
                        /* The two frames are identical  */
                   if (curr_dntcare <= other_dntcare)
                      curr_f = delete_test_frame(curr_l, curr_f);
                   else {
                      other_f=delete_test_frame(other_l, other_f);
                   }
                   break;
                }

                        /* Determine the (next) other frame */
                if (other_f == prev_other_f)
                   other_f = other_f->next;
            }
                        /* Determine the (next) current frame */
            if (curr_f == prev_f)
               curr_f = curr_f->next;
        }
}


/*****************************************************************************
*   TEST_FRAME* DELETE_TEST_FRAME()
*   This procedure is used to delete a test frame and returns the pointer to
*   previous test frame or a test list head (if there is no previous test
*   frame).
*****************************************************************************/
TEST_FRAME *delete_test_frame(test_l, del_f)
TEST_LIST *test_l;
TEST_FRAME *del_f;
{
    TEST_FRAME *curr_f, *prev_f;

    /*-------------------------------------------------------------
     *   Search for the frame to be deleted
     */
```

```
    for (curr_f=test_l->head, prev_f=curr_f; curr_f != del_f; prev_f=curr_f,
        curr_f=curr_f->next);

    /*--------------------------------------------------------------------
     *   Determine the previous frame , the test head and tail
     */
    if (curr_f==test_l->head) {
       test_l->head = curr_f->next;
       prev_f = test_l->head;
    }
    else
       prev_f->next = curr_f->next;

    if (curr_f==test_l->tail)
       test_l->tail = prev_f;

    /*--------------------------------------------------------------------
     *   Delete the specified frame
     */
    free(curr_f->cause);
    free(curr_f->inter);
    free(curr_f->derive);
    free(curr_f->effect);
    free(curr_f);

    /*--------------------------------------------------------------------
     *   Return the previous frame pointer
     */
    return(prev_f);
}



/*****************************************************************************
*   VOID DERIVE_EFFECT_VALUES()
*   This procedure is used to derive the unassigned effect node values in all test
*   frames of a test frame list.
*****************************************************************************/
void derive_effect_values(test_l, graph)
TEST_LIST *test_l;
GRAPH *graph;
{
    TEST_FRAME *test_f;
    GRPNODE  *effect;
    LINK     *link;

    for (test_f = test_l->head; test_f != NULL; test_f = test_f->next)
                        /* Loop for all test frames */

        for (effect=graph->effect; effect != NULL; effect=effect->next){
                        /* Loop for all effect nodes  */

            if (effect->type==INV_EFFECT)
                        /* The node is an invariant node  */
               test_f->effect[effect->number-1] = 1;
            else {
                        /* The node is not an invariant node;  Set the
                           effect node value to 1 if there is a linked node
                           (an intermediate node) that has value = 1 */
               for (link=effect->bakw; link != NULL ; link=link->next)
                   if (test_f->inter[link->node->number-1]==1) {
                      test_f->effect[effect->number-1] = 1;
                      break;
                   }
               if (test_f->effect[effect->number-1] == -1)
                        /* Set the effect node value to 0 if the node is
                           unassigned */
                  test_f->effect[effect->number-1] = 0;
            }
        }
}
```

```
/*****************************************************************************
*   VOID PRINT_TEST_FRAME()
*   This procedure is used to print all test frame values (cause and effect
*   values).
*****************************************************************************/
void print_test_frame(test, graph, table, default_val)
TEST_LIST *test;
GRAPH *graph;
TABLE *table;
int default_val;
{
   TEST_LIST *curr_l;
   TEST_FRAME *curr_f;
   TBL_REC     *curr_c, *curr_e;
   int nc, ne, i, j, c, e, k, f;

   /*-------------------------------------------------------------
    *   Print the test frame table header
    */
   if (default_val==1) {
      printf("\n\n\nTrue-Effect Test Frames:\n");
      printf("=======================\n\n");
   }
   else {
      printf("\n\n\nFalse-Effect Test Frames:\n");
      printf("=======================\n\n");
   }
   printf("-------------------------------------------------------------\n");
   printf("Frame No. ******* Cause No. *******   ******* Effect No.********\n");
   nc = graph->ncause;
   ne = graph->neffect;

   /*-------------------------------------------------------------
    *   Print the cause and effect node numbers in the table header
    *   (maximum 8 numbers in one row)
    */
   for (c=0, e=0; nc-c > 0 || ne-e > 0; ) {
       printf("          ");
       for (k=0; k < 8; k++) {
           if (nc-c > 0) {
              c++;
              printf("%2d ", c);
           }
           else
              printf("   ");
       }
       printf("    ");
       for (k=0; k < 8; k++) {
           if (ne-e > 0) {
              e++;
              printf("%2d ", e);
           }
           else
              printf("   ");
       }
       printf("\n");
   }
   printf("-------------------------------------------------------------\n");



   /*-------------------------------------------------------------
    *   Print all test frame cause and effect node values
    */
   for (curr_l = test, f=1; curr_l != NULL; curr_l = curr_l->next)
                           /* Loop for all test list    */

       for (curr_f=curr_l->head; curr_f!=NULL; curr_f=curr_f->next, f++) {
```

```
                    /* Loop for all test frame in the current test
                       list */

        printf ("  %3d     ", f);
        for (c=0, e=0; nc-c > 0 || ne-e > 0; ) {
                    /* Loop for all cause and effect nodes  */

                    /* Print the value of cause nodes */
            for (k=0; k < 8; k++) {
                if (nc-c > 0) {
                   if (curr_f->cause[c]==-1)
                      printf(" d ");
                   else
                      printf(" %1d ", curr_f->cause[c]);
                   c++;
                }
                else
                   printf("    ");
            }
            printf("    ");

                    /* Print the value of effect nodes */
            for (k=0; k < 8; k++) {
                if (ne-e > 0) {
                   if (curr_f->effect[e]==-1)
                      printf(" d ");
                   else
                      printf(" %1d ", curr_f->effect[e]);
                   e++;
                }
                else
                   printf("    ");
             }
             printf("\n          ");
        }
       printf("\n");
   }
printf("-----------------------------------------------------------------\n");

/*-------------------------------------------------------------
 *   Print the legend of the cause nodes
 */
printf("\n Causes:\n");
for (c=0, curr_c=table->cause; curr_c != NULL ; c++, curr_c=curr_c->next) {
    printf("\n    %2d : ", c+1);
    for (i = 0; i < 3; i++) {
        if (i==2 & strlen(curr_c->pred_part[0])+strlen(curr_c->pred_part[1])+
            strlen(curr_c->pred_part[2]) > 60)
           printf("\n          ");
        printf("%s ", curr_c->pred_part[i] );
    }
}

/*-------------------------------------------------------------
 *   Print the legend of the effect nodes
 */
printf("\n\n Effects:\n");
for (e=0, curr_e=table->effect; curr_e != NULL ; e++, curr_e=curr_e->next) {
    printf("\n    %2d : ", e+1);
    for (i = 0; i < 3; i++) {
        if (i==2 & strlen(curr_e->pred_part[0])+strlen(curr_e->pred_part[1])+
            strlen(curr_e->pred_part[2]) > 60)
           printf("\n          ");
        printf("%s ", curr_e->pred_part[i] );
    }
}


/*-------------------------------------------------------------
 *   Print the legend of the nod values
 */
printf("\n\n Node Values:\n");
```

```
    printf("\n     0 : FALSE");
    printf("\n     1 : TRUE");
    printf("\n     d : Don't care (Either TRUE or FALSE)");


    /*---------------------------------------------------------------
     *   Print the number of computations
     */
    printf("\n\n\n The Number of Computations:\n");
    printf("\n    Zeroes-AND Computations     : %6d", compute_0);
    printf("\n    Non Zeroes-AND Computations : %6d\n\n\n", compute_1);
}
```

VITA

Teguh Rahardjo

Candidate for the Degree of

Master of Science

Thesis: TEST FRAME GENERATION FROM **Z** SPECIFICATIONS

Major Field: Computer Science

Biographical:

Personal Data: Born in Yogyakarta, Indonesia, On October 21, 1956, son of Soetar Reksoatmodjo and Roekminah.

Education: Graduated from Sekolah Menengah Atas Negeri III, Yogyakarta, Indonesia in December 1975; received Bachelor of Engineering in Electrical Power Engineering from Institut Teknologi Bandung, Bandung, Indonesia in March 1983. Completed the requirements for the Master of Science degree in Computer Science in July 1995.

Experience: Employed as Systems Analyst by the Computing Center of Nusantara Aircraft Industries Ltd., Bandung, Indonesia, 1983 to 1992.

Professional Membership: IEEE Computer Society.