Unit testing database applications using SpecDB: A database of software specifications

Rana Farid Mikhail

University of South Florida

Follow this and additional works at: http://scholarcommons.usf.edu/etd

Part of the American Studies Commons

Scholar Commons Citation

http://scholarcommons.usf.edu/etd/2633

This Dissertation is brought to you for free and open access by the Graduate School at Scholar Commons. It has been accepted for inclusion in Graduate Theses and Dissertations by an authorized administrator of Scholar Commons. For more information, please contact scholarcommons@usf.edu.
Unit Testing Database Applications Using SpecDB:

A Database of Software Specifications

by

Rana Farid Mikhail

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
Department of Computer Science and Engineering
College of Engineering
University of South Florida

Co-Major Professor: Abraham Kandel, Ph.D.
Co-Major Professor: Donald Berndt, Ph.D.
Kenneth Christensen, Ph.D.
Miguel Labrador, Ph.D.
Dewey Rundus, Ph.D.

Date of Approval:
April 26, 2006

Keywords: Software Testing, Formal Specifications, Reverse Engineering,
Automated Code, Constraint Generation

© Copyright 2006, Rana Farid Mikhail
DEDICATION

To my beloved late father, Dr. Farid Halim Mikhail; All I am, I owe to you. Your words of wisdom will remain in my heart and mind as long as I live.
ACKNOWLEDGMENTS

There are many people who helped contribute to the successful completion of this dissertation. First I would like to thank my two advisors Dr. Abraham Kandel and Dr. Donald Berndt for providing me with invaluable assistance in performing my research; their feedback always served to improve the work in meaningful ways. I would also like to thank the members of my committee, Dr. Ken Christensen, Dr. Miguel Labrador, and Dr. Dewey Rundus and also Dr. Geoffrey Okogbaa, the chair of my dissertation defense, for their valuable comments and help. I deeply acknowledge Mrs. Judy Hyde, the office manager for NISTP for her administrative help that enabled me to reach my goal.

I am also deeply grateful to my family, for being there for me. My beloved husband, Emad, for his understanding, love and patience, my dear mother, Nadia, for her continuous enthusiastic support and interest in what I was doing and for my brother Ramy for always believing in me. Finally, but most importantly, my deepest gratitude is for my late father Dr. Farid Mikhail. Acquiring my doctorate degree was his dream. I am deeply saddened by the fact that he is no longer here to see me realize his wish. But I am sure that completing this dissertation will please him; I am honored to be able to accomplish this and rise to my father's expectations.
TABLE OF CONTENTS

LIST OF TABLES ...................................................................................................................... v

LIST OF FIGURES ................................................................................................................... vii

ABSTRACT ................................................................................................................................. ix

CHAPTER 1 INTRODUCTION AND LITERATURE SURVEY .................................................. 1

1.1 The Need For Testing ........................................................................................................ 1
1.2 Why Does Software Have Errors .................................................................................. 2
1.3 Software Testing Definitions ......................................................................................... 4
  1.3.1 Software Errors, Faults and Failures ..................................................................... 4
  1.3.2 Software Testing .................................................................................................... 5
1.4 When Should Testing Start in the Software Lifecycle ................................................... 6
1.5 Types of Testing Techniques ........................................................................................... 9
1.6 UML ................................................................................................................................... 10
1.7 Formal Specification Languages ...................................................................................... 10
1.8 Current Technology and Available Software Testing Systems in Today’s Market ......... 11
  1.8.1 Types of Automated Testing Tools ....................................................................... 11
  1.8.2 Design and Visual Modeling Tools ....................................................................... 12
    1.8.2.1 Rational Rose ............................................................................................... 12
    1.8.2.2 Oracle Designer .......................................................................................... 13
  1.8.3 Automated Testing Tools ....................................................................................... 14
  1.8.4 Disadvantage of Record and Play Back Test Automation Tools ......................... 18
1.9 Related Literature ........................................................................................................... 18
  1.9.1 AGENDA: A Test Generator for Relational Database Applications .................. 19
  1.9.2 Executable Software Specifications .................................................................... 21
1.10 Research Goals and Contributions ............................................................................. 22
1.11 Objectives and Accomplishments .............................................................................. 27

CHAPTER 2 SPECDB: A DATABASE REPRESENTATION OF SOFTWARE SPECIFICATIONS .................. 29

2.1 Introduction ..................................................................................................................... 29
2.2 The Advantages of a Database Representation of Specifications ................................. 30
2.3 The Design of the Specification Database, SpecDB ....................................................... 32
2.4 Entities in the Design and How Each Represents Specific Software Requirements ..................................................................................................... 37
  2.4.1 Operator and Valid Operands .................................................................................................................. 37
  2.4.2 List_of_Tables and Table Description .................................................................................................... 38
  2.4.3 Types, Types_LOV and Type Restrictions .............................................................................................. 39
    2.4.3.1 List of Valid Values (LOV) ......................................................................................................... 39
    2.4.3.2 Other Restrictions .................................................................................................................. 39
  2.4.4 Variables and Restrictions on Variable Values .......................................................................................... 41
  2.4.5 Procedure, Function_Declaration, and Subroutine_IP_Description ...................................................... 42
  2.4.6 Subroutine_IP_Instance, Designed_SR_Post_Conditions and Runtime_SR_Post_Conditions ................. 44
  2.4.7 Dataflow, Triggers and Input_Output_Definition ................................................................................. 46
  2.4.8 Assignment and Calculation ................................................................................................................. 48
  2.4.9 Predicate .............................................................................................................................................. 50
  2.4.10 Database Operations, Tables Involved in Database Operations ......................................................... 52
  2.4.11 Classes, Objects and Class Relations ................................................................................................. 55

2.5 Conclusion ............................................................................................................................................. 56

CHAPTER 3 REPRESENTING FORMAL SPECIFICATIONS IN SPECDB: A TRANSLATION ALGORITHM .............................................................................................................. 58

  3.1 The Translation Algorithm From Formal Specifications to SpecDB .......................................................... 59
  3.2 Assumptions and Restrictions .................................................................................................................... 60
  3.3 Preparing the Formal Specifications for Translation ..................................................................................... 60
  3.4 Storing Formal Specification Statements into SpecDB ................................................................................ 61
    3.4.1 Defining Schema Names and Operations ............................................................................................. 61
    3.4.2 Types of Schemas .................................................................................................................................. 62
    3.4.3 User-Defined Types ............................................................................................................................ 63
    3.4.4 Specifying Values for a User-Defined Type ...................................................................................... 64
    3.4.5 Defining Sets or Tables ........................................................................................................................ 65
    3.4.6 Function Declaration ........................................................................................................................... 66
    3.4.7 Subroutine Input Restriction ................................................................................................................. 67
    3.4.8 Variable Definition and Subroutine Inputs ............................................................................................ 68
    3.4.9 Output Definition .................................................................................................................................... 69
    3.4.10 Setting Variable Restrictions ............................................................................................................. 70
    3.4.11 Assignment Operation ........................................................................................................................ 71
    3.4.12 Assignment Operation and Function Call ........................................................................................... 72
    3.4.13 Assignment Operation for Sets ........................................................................................................... 74
    3.4.14 Assignment Operation for a Set .......................................................................................................... 75
    3.4.15 Inserting Records in the Database ...................................................................................................... 75
    3.4.16 Output Table Definition ..................................................................................................................... 77
    3.4.17 Assigning a Variable to a Calculation ................................................................................................. 78
    3.4.18 Constraining the Values Populating a Table or Set ........................................................................... 80
  3.5 Conclusion ............................................................................................................................................. 83
<table>
<thead>
<tr>
<th>Chapter 4</th>
<th>Automated Constraints Generator: An Application of SpecDB</th>
<th>84</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>The Design of the Additional Tables in SpecDB</td>
<td>86</td>
</tr>
<tr>
<td>4.1.1</td>
<td>Using List_of_Tables and Extending Table_Description</td>
<td>87</td>
</tr>
<tr>
<td>4.1.2</td>
<td>Adding the Constraints Table to SpecDB and Using the Predicate Table</td>
<td>90</td>
</tr>
<tr>
<td>4.1.3</td>
<td>Other Tables Used from SpecDB, Type_Excluded_Ranges and Types_LOV</td>
<td>94</td>
</tr>
<tr>
<td>4.1.3.1</td>
<td>Representing List of Values (LOV) Restrictions</td>
<td>94</td>
</tr>
<tr>
<td>4.1.3.2</td>
<td>Representing Other Field Restrictions</td>
<td>94</td>
</tr>
<tr>
<td>4.2</td>
<td>Automatically Generating Database-Level Constraints: The Algorithm</td>
<td>95</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Generating Simple Static Constraints – The Algorithm</td>
<td>96</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Generating Complex Dynamic Business Rules Enforcement Code</td>
<td>100</td>
</tr>
<tr>
<td>4.2.2.1</td>
<td>Complex Business Rules Generation Algorithm</td>
<td>102</td>
</tr>
<tr>
<td>4.3</td>
<td>Conclusion</td>
<td>104</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter 5</th>
<th>A Reverse Engineering Testing Tool Using SpecDB</th>
<th>105</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Technique</td>
<td>107</td>
</tr>
<tr>
<td>5.1.1</td>
<td>Input and Pre Conditions Classification and Categorization</td>
<td>108</td>
</tr>
<tr>
<td>5.1.1.1</td>
<td>Examples of Inputs from the Database Displayed by the RE Testing Tool</td>
<td>111</td>
</tr>
<tr>
<td>5.1.1.2</td>
<td>Examples of Inputs from the GUI Displayed by the RE Testing Tool</td>
<td>112</td>
</tr>
<tr>
<td>5.1.2</td>
<td>Output and Post Conditions Classification and Categorization</td>
<td>112</td>
</tr>
<tr>
<td>5.1.2.1</td>
<td>Examples of Outputs Displayed by the RE Testing Tool</td>
<td>115</td>
</tr>
<tr>
<td>5.2</td>
<td>Case Study</td>
<td>116</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Problem Statement for the Vacation Salesman Commission Program</td>
<td>116</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Implementation</td>
<td>117</td>
</tr>
<tr>
<td>5.2.3</td>
<td>Program Graph for the Vacation Salesman Commission Problem</td>
<td>121</td>
</tr>
<tr>
<td>5.2.4</td>
<td>DD-Path Graph for the Vacation Salesman Commission Problem</td>
<td>122</td>
</tr>
<tr>
<td>5.3</td>
<td>Examples</td>
<td>123</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Test Case 1</td>
<td>124</td>
</tr>
<tr>
<td>5.3.1.1</td>
<td>Scenario</td>
<td>124</td>
</tr>
<tr>
<td>5.3.1.2</td>
<td>Expected Output</td>
<td>124</td>
</tr>
<tr>
<td>5.3.1.3</td>
<td>Expected GUI Output</td>
<td>124</td>
</tr>
<tr>
<td>5.3.1.4</td>
<td>The RE Tool Display of the Inputs for Test Case1</td>
<td>125</td>
</tr>
<tr>
<td>5.3.1.5</td>
<td>Process and Actual Output</td>
<td>126</td>
</tr>
<tr>
<td>5.3.1.6</td>
<td>Comparing Expected to Actual Output</td>
<td>128</td>
</tr>
<tr>
<td>5.3.2</td>
<td>Test Case 2</td>
<td>134</td>
</tr>
<tr>
<td>5.3.2.1</td>
<td>Process and Actual Output</td>
<td>135</td>
</tr>
<tr>
<td>5.3.3</td>
<td>Test Case 3</td>
<td>137</td>
</tr>
<tr>
<td>5.4</td>
<td>Conclusion</td>
<td>139</td>
</tr>
</tbody>
</table>
CHAPTER 6 ENHANCING OTHER TESTING TOOLS USING SPECDB .......... 141

6.1 Using the Database Representation of SpecDB to Expand the Testing
   Domain ............................................................................................................. 142
6.2 Enhancing State Validation Tools Using SpecDB ................................. 145
6.3 Expanding the Testing Scope Beyond Variables and Database States ...... 147
6.4 Testing Different Operation Types ......................................................... 147
6.5 Conclusion ............................................................................................... 148

CHAPTER 7 CONCLUSION AND FUTURE WORK ........................................ 149

7.1 A Comparison Between Testing Tools .................................................. 150
  7.1.1 Test Cases and Scenarios ................................................................. 150
    7.1.1.1 Test Case 1 ................................................................................. 150
    7.1.1.2 Test Case 2 ................................................................................. 151
    7.1.1.3 Test Case 3 ................................................................................. 151
    7.1.1.4 Test Case 4 ................................................................................. 152
    7.1.1.5 Test Case 5 ................................................................................. 152
    7.1.1.6 Test Case 6 ................................................................................. 153
  7.1.2 Comparing the Ability of Testing Tools to Identify Errors ............... 153
7.2 Research Accomplishments at a Glance ............................................. 157
7.3 Conclusion and Recommendations ..................................................... 158
7.4 Future Work at a Glance ...................................................................... 161

REFERENCES .................................................................................................... 165

APPENDICES ...................................................................................................... 173

  Appendix A: SQL Scripts for SpecDB Entities ........................................... 174
  Appendix B: SQL Scripts for SpecDB Constraints ................................... 191
  Appendix C: PL-SQL Code for the Constraints Generator ....................... 221

ABOUT THE AUTHOR .................................................................................. End Page
LIST OF TABLES

Table 1.1: Automated Testing Tools ................................................................. 17
Table 4.1: Snapshot of SpecDB Table_Field_Description for Constraint 1 ........ 91
Table 4.2: Snapshot of SpecDB Constraint Table for Constraint 1 ..................... 91
Table 4.3: Snapshot of SpecDB Predicate Table for Constraint 1 ....................... 91
Table 4.4: Snapshot of SpecDB Table_Field_Description for Constraint 2 ........ 92
Table 4.5: Snapshot of SpecDB Constraint Table for Constraint 2 ....................... 92
Table 4.6: Snapshot of SpecDB Predicate Table for Constraint 2 ....................... 92
Table 5.1: Input and Pre Conditions Classification and Categorization .......... 111
Table 5.2: Examples of Inputs from a Database .............................................. 112
Table 5.3: Examples of GUI Inputs ................................................................. 112
Table 5.4: Output and Post Conditions Classification and Categorization .............. 115
Table 5.5: Example of Outputs from the GUI .................................................. 115
Table 5.6: DD-Path Graph for the Vacation Salesman Commission Problem ...... 122
Table 5.7: The RE Testing Tool Displayed Database Inputs for Test Case 1 ......... 126
Table 5.8: The RE Testing Tool Displayed GUI Input for Test Case 1 ................. 126
Table 5.9: GUI Output for Test Case 1 – Part1 ............................................... 127
Table 5.10: GUI Output for Test Case 1 – Part2 ............................................... 127
Table 5.11: GUI Output for Test Case 1 – Part3 ............................................... 127
Table 5.12: GUI Output for Test Case 1 After Fault Injection ............................. 130
Table 5.13: SpecDB Dataflow Table Snapshot – Original Specifications .......... 131
Table 5.14: SpecDB Var Table Snapshot – Original Specifications .............................. 131
Table 5.15: SpecDB Input_Output_Definition Table Snapshot – Original Specifications .............................................................................................. 132
Table 5.16: SpecDB Var Table Snapshot – RE Tool Implementation Specifications ........................................................................................................ 133
Table 5.17: RE Testing Tool Display of the GUI Output for Test Case 2 ..................... 135
Table 5.18: SpecDB Var Table Snapshot – RE Tool Specifications – Test Case 2 ...................................................................................................................... 136
Table 5.19: SpecDB Input_Output_Definition Table Snapshot – RE Tool Implementation Specifications – Test Case 2 ............................................ 137
Table 5.20: Database Output for Example 3............................................................. 139
Table 7.1: A Comparison Between How Testing Tools/ Techniques Handle Test Cases 1-6 ............................................................................................. 154
Table 7.2: A Comparison Between the Capabilities of Testing Tools and Techniques ........................................................................................................ 156
LIST OF FIGURES

Figure 1.1: The Relationship Between a Program Specification and its Implementation ................................................................. 5
Figure 1.2: Dissertation Contribution: Entities and Chapter Layout ................................................................. 27
Figure 2.1: An Overall Diagram of SpecDB’s Design ................................................................. 34
Figure 2.2: A Focused Diagram of the General SpecDB Design (Left Section) ................................................ 35
Figure 2.3: A Focused Diagram of the General SpecDB Design (Right Section) ................................................ 36
Figure 2.4: Representing Operators and Operator Overloading in SpecDB .................................................. 37
Figure 2.5: Representing User Tables and Sets and their Description in SpecDB ........................................ 38
Figure 2.6: Representing Types, Type Restrictions and Types Valid Values in SpecDB ................................................................. 40
Figure 2.7: Representing Variables, Variable Restrictions and Variable Valid Values in SpecDB ................................................................. 42
Figure 2.8: Representing Units (Sub-routines) in SpecDB ................................................................. 43
Figure 2.9: Representing Sub-routine Instances and Post Conditions in SpecDB ................................................ 45
Figure 2.10: Representing Dataflow, Triggers and System Input and Outputs in SpecDB ................................................................. 48
Figure 2.11: Representing Assignment Operations and Calculations in SpecDB ................................................................. 49
Figure 2.12: Representing Predicates and Conditions in SpecDB ................................................................. 51
Figure 2.13: Representing Database Operations in SpecDB ................................................................. 54
Figure 2.14: Representing Object Oriented Classes and Objects in SpecDB ................................................................. 56
Figure 3.1: AddBirthday Schema ........................................................................................................ 61
Figure 3.2: Report, User-defined Type Specification ........................................................................................................ 64
Figure 3.3: FindBirthday Schema ................................................................. 69
Figure 3.4: Success Schema ........................................................................ 72
Figure 3.5: InitBirthdayBook Schema ......................................................... 75
Figure 3.6: Remind Schema ......................................................................... 78
Figure 3.7: RAddBirthday Schema .............................................................. 79
Figure 4.1: SpecDB Original Table Description .......................................... 87
Figure 4.2: Table Field Description Table ..................................................... 88
Figure 4.3: Constraint Table ......................................................................... 91
Figure 4.4: Representing Complex Constraints in SpecDB ......................... 93
Figure 4.5: Representing Types, Extending the Initial Representation .......... 95
Figure 5.1: Program Graph for the Vacation Salesman Commission Problem .. 121
Figure 5.2: DD-Path Graph for the Vacation Salesman Commission Problem .. 122
Figure 5.3: RE Tool Model Diagram ............................................................. 123
Figure 5.4: The Expected GUI Output for Test Case 1 ............................... 125
Figure 5.5: GUI Output for Test Case 2 ......................................................... 135
Figure 6.1: Representing AGENDA Data in SpecDB .................................... 144
In this dissertation I introduce SpecDB, a database created to represent and host software specifications in a machine-readable format. The specifications represented in SpecDB are for the purpose of unit testing database operations. A structured representation aids in the processes of both automated software testing and software code generation, based on the actual software specifications. I describe the design of SpecDB, the underlying database that can hold the specifications required for unit testing database operations.

Specifications can be fed directly into SpecDB, or, if available, the formal specifications can be translated to the SpecDB representation. An algorithm that translates formal specifications to the SpecDB representation is described. The Z formal specification language has been chosen as an example for the translation algorithm. The outcome of the translation algorithm is a set of machine-readable formal specifications.
To demonstrate the use of SpecDB, two automated tools are presented. The first automatically generates database constraints from represented business rules in SpecDB. This constraint generator gives the advantage of enforcing some business rules at the database level for better data quality. The second automated application of SpecDB is a reverse engineering tool that logs the actual execution of the program from the code. By automatically comparing the output of this tool to the specifications in SpecDB, errors of commission are highlighted that might otherwise not be identified. Some errors of commission including coding unspecified behavior together with correct coding of the specifications cannot be discovered through black box testing techniques, since these techniques cannot observe what other modifications or outputs have happened in the background. For example, black box, functional testing techniques cannot identify an error if the software being tested produced the correct specified output but moreover, sent classified data to insecure locations. Accordingly, the decision of whether a software application passed a test depends on whether it coded all the specifications and only the specifications for that unit. Automated tools, using the reverse engineering application introduced in this dissertation, can thus automatically make the decision whether the software passed a test or not based on the provided specifications.
CHAPTER 1
INTRODUCTION AND LITERATURE SURVEY

1.1 The Need For Testing

One of the common but very true jokes about the software industry tells of a software entrepreneur proudly stating that if the automobile industry had advanced in the past decade with the same rate as that of the computer industry, we would all be driving $25 cars that run for a 1,000 miles per gallon of gasoline. To which an automobile CEO replies, that if cars were like today’s software, they would crash twice a day for no reason, and when you call for service, they would tell you to shut down the engine, as well as all windows, get out of the car, lock it, and start over! Indeed, in the past decade, computer hardware has advanced significantly offering much higher processing capability and speed. Consequently, the complexity and size of software has increased exponentially too. Software is vital to almost every industry and we are becoming ever more dependent on our computers to perform everyday tasks. As the complexity and size of software grow, so does the need to insure the reliability and precision of software outputs. Software errors can be fatal in numerous industries; including airlines, medical and the military. Billions of dollars are spent on software and software testing and maintenance annually by the DoD. Software errors result in hundreds of millions in losses annually from the DoD alone [1] (software errors are defined in Section 1.3.1).
Accordingly, a lot of attention is now directed towards assuring software reliability, accuracy, security and usefulness. It is crucial to dedicate a lot of resources, including time, effort and a budget during the software lifecycle to verify that the end product is useful, usable and accomplishes the tasks, it was built to perform.

1.2 Why Does Software Have Errors

Software errors result from many reasons, from misunderstanding of the requirements, to poor design; also from tired or careless programmers to rushed deadlines [2]. Often testing is delayed until later in the software development life cycle, and little time is given for testing, debugging, and re-testing. Coverage gaps are also a problem; this happens when not all parts of the software are tested.

Some software failures happen due to a specific scenario, a sequence of events that might have been tested individually but not in that particular sequence, that causes the software failures. The operating environment, various connected hardware peripherals, other concurrently running software and different operating systems could also be a source of software failures [4].

It was proven that complete testing is impossible [3]. Meaning that it is impossible to test all values of each possible input, valid and invalid, with their respective combinations for a software application with considerable size and complexity; the number of test cases will be astronomical and the time needed for testing and verification will be almost endless. For
example, if a very simple program takes only three inputs, an integer *Age*, between 0 and 99, a character, *Initial*, between A and Z, and another integer representing a *Month* between the integers 1 and 12.

\[
\text{Age: integer } [0..99], \quad \text{Initial: char } [A..Z], \quad \text{Month: integer } [1..12]
\]

There are 100 possible valid values for Age, 26 valid values for Initial, and 12 valid values for Month. Accordingly there are \(100 \times 26 \times 12 = 31,200\) possible combinations of valid values only for this simple program. If complete testing was required, there would be 31,200 test cases that should be carried out to test the operation of this program on valid values alone. Even if the 31,200 test cases were run, this would not be considered *complete testing*. The number of test cases required for *complete testing* will be much greater, when the invalid values are added. Invalid values could be in one, two or all three inputs. The number will increase exponentially if we want to create test cases for all these cases; where only one of the three variables contains an invalid value, a combination of two of the variables are invalid, or all three. For example, the Age variable is an integer, one can enter any integer from the negative integer boundary to the positive integer boundary. Also, one can try to enter characters, real numbers or any other value for the Age integer value and test the program behavior and how it handles exceptions. The possibilities and combinations of all those cases for this very simple three input program are thus almost endless. Also it is impossible to completely test all values of real numbers, even in a small range, and for strings representing names for example, etc.
Some transient errors are revealed in complex distributed or embedded control systems after prolonged usage. This may be a result of a memory overflow, or minute system clock discrepancies that accumulate to a noticeable difference after prolonged usage. Such errors are very difficult to recognize [77], and require high volume system testing or long sequence testing techniques, and extended random regression testing [79]. Hence there is a need for better software testing techniques, which focus on testing parts of the input domain, yet, attain good coverage to verify software quality.

1.3 Software Testing Definitions

1.3.1 Software Errors, Faults and Failures

A software error is a mismatch between the program and its specifications, provided that the specifications are accurate and complete. A software application is considered erroneous if it fails to be useful. A software fault is defined as the execution of an error [5]. Faults of commission are those resulting from incorrectly coding specifications, or coding unspecified requirements. Faults of omission result from failing to implement specified requirements. A software failure happens when a fault of commission is executed while running the software application [5].

From Figure 1.1, the goal of testing is to make sure that

\[ S \cap P = S = P \]

If the above equation is reached in testing a software application, the resulting product is very well tested and functional, provided that the specifications are correct, accurate, complete and do not contain any contradictions.
1.3.2 Software Testing

Software testing is becoming an important area of research aimed at producing more reliable systems and minimizing programming errors [78]. Since it is impossible to completely test how a software application performs on all values of all of its inputs, as described in Section 1.2, a good strategy to test software has to be followed in order to choose test cases that test the software without coverage gaps. However, there is much more to good testing than running a program several times to check if it functions correctly [6]. In order to systematically test software, we need to model the environment in which the software is intended to run, select test scenarios that attain good system coverage, execute and evaluate the test cases, correct the errors, and measure the testing progress [4]. Some faults may result from correcting other errors, and hence regression
testing is often used, which entails re-running all test cases after an error has been corrected to verify that the fix did not inject any errors in a part of the software that was previously running correctly. Software Testing has been defined in many ways. Some authors define it as the process of running a software application to verify its correct execution of its valid and complete specifications, and verify that it runs correctly in the environment it was created for [5,86]. Other authors argue that test scenarios that do not reveal errors or failures are also of great importance, since the testing team is assuring by such scenarios that the system actually implemented the required specifications correctly [6]. Software can also fail by executing too slowly or using too much memory; reasonable execution speed and memory usage are obvious requirements that might not be in the specification documents. A number of good references for software testing are [5,6,553]. Accordingly, it is crucial to create test cases that reflect a good coverage of all areas of the software and determine its usability.

1.4 When Should Testing Start in the Software Lifecycle

One of the most important and useful practices in software engineering is the incorporation of software testing from the very beginning of the software lifecycle. The software life cycle is described in more detail in [7,8,9]. Waiting until the software is implemented and then starting to test the software was proven in the last decade by many authors and software team managers to be a very costly and difficult task to manage and carryout. Testing should start as soon as the analysts finish writing the first draft of the specification document describing what the required system should be built to accomplish. Testers can then begin identifying test cases, which are called system level test cases. System level test cases are very easily
interpreted and basically describe in a broad spectrum what the main functionalities of the software should be. By reading the system level test cases, both the development team and the end users can reach an early agreement and common understanding of the functionality of the software under construction.

After agreeing on the specifications, and the system level test cases, the designers can then start defining the details of the system, and answer many questions from as broad as what programming language should be used to the minute details of the functionality of each sub-routine. Testers have a very important role to play during the design phase. During that phase testers should generate the integration level and unit level test cases. Integration level test cases are targeted towards testing the correctness of the interaction between the different sub-routines (methods, functions, procedures, etc.) and the correctness of the interaction between the different modules or subsystems of the software under construction [5].

Once implementation starts, the programmers should have both the design and the test cases that will be exercised against their implementation. Having the complete set of test cases, that cover all possible outputs of the functionality a programmer is implementing, will help the programmer to understand the design and implement it correctly. There is a strong possibility that the programmer will make sure the implemented code passes the given tests before submitting the code for integration with the rest of the system. There are many advantages for the above approach of starting testing early in the software lifecycle. In this situation, when the programmer is given both the design and test cases, testing starts very early in the software lifecycle. A test case will be executed multiple times to check that the
system passes it correctly at the unit level as well as the system level after integration with the rest of the system modules. There will be no ambiguity in what exactly the programmer should implement, even if the programmer misinterprets the design and specified functionality, reading the test cases, and their respective expected output, will clear much of the ambiguity, and will assure a higher level of software quality, compared to that when only a design is given to the programmer. Once the implementation phase is over, the majority of the testing phase will have been implemented already. The cost of integration and system level testing will be minimal compared to that of delaying testing until the last phase in the software lifecycle. Also the overall quality of the implemented software will be much higher than might result given the same resources of time and money, but leaving testing till after implementation. It is worth noting that the waterfall model of software development where all the specifications of the software are known from the beginning is not what is generally followed in real world software development environments. Test cases should be created for all the specifications as they are gathered for the different builds or versions of the software under construction. Testing should be carried out for each of those versions to make sure that the specifications thus far collected, are all implemented correctly. Specifications should be tested to make sure that they do not contradict previously specified requirements. Google, Microsoft, and Sun Microsystems are investing $7.5 million to fund research on alternative software development models to the waterfall model with higher reliability and faster deployment, at UC Berkeley [101].
In conclusion, it is essential that the testing process starts from the beginning of the software lifecycle and be concurrent with all the phases as a better software engineering practice.

1.5 Types of Testing Techniques

There are two approaches to identifying test cases: functional and structural. Functional testing, also often referred to as black box testing, is the process of testing the executable program without any reference to the source code of the software. It is called functional since we treat the software as a function, we provide inputs, and observe its output. Functional test cases can be reused even when the code changes, is translated into another language, or its structure differs. Another advantage of functional testing is that the test cases can be created during the early phases of development, even before implementation starts, since it is based on the specifications. There are a number of functional testing techniques including boundary value analysis, equivalence class testing, and decision table-based testing. The reader can refer to [5] for a good coverage of those three techniques. Other black box testing techniques are domain driven, stress driven, specification driven, risk driven, random / statistical, function, regression, scenario, use case, user and exploratory testing. The reader can refer to [7] for an explanation of these techniques. In structural testing on the other hand, also referred to as white or clear box testing, test cases are created to execute specific branches and paths in the source code. Structural testing techniques include decision-to-decision and define-use testing. The advantage of structural testing is in its ability to confirm that all paths in the source code have been executed in test cases, and thus those testing techniques can be used as a coverage matrix to guide other functional testing techniques.
1.6  UML

To better understand some types of today’s automated testing tools that are targeted towards
testing in the requirements definition and design phases, one should understand the Unified
Modeling Language (UML). The reader can refer to [11,12,13,14] for UML syntax and
diagrams. The majority of software development organizations today, use UML in
describing a system design. System designers represent the design of a system using UML, a
set of diagrams that both technical and non-technical system end users can understand and
use as a basis for discussions and reassurance that the software under development is what
the end-users actually have in mind. Those UML diagrams are then used as a basis for more
detailed system design, implementation and testing. In their book, *Business Modeling with
UML* [11], Eriksson and Penker describe the nine different UML diagrams and the use of
each for business modeling. They state that ‘UML standardizes notation for describing a
process, but it doesn’t standardize a process for producing those descriptions’ [11]. This
leads to the first research objective, namely, to create a standard representation for storing
UML or software specifications represented by a formal specification languages like Z, since
UML diagrams are not machine interpretable.

1.7  Formal Specification Languages

Formal specifications, also known as formal methods, are a mathematical-like notation used
to describe the specifications, i.e. what a system should execute, without necessarily going
into the details of how the requirements should be implemented. Formal specifications
where developed primarily to result in accurate, unambiguous software specifications to
replace informal natural language specifications for better engineered software. The reader can refer to [15-24,65] for a detailed description of formal methods, use, semantics and syntax.

1.8 Current Technology and Available Software Testing Systems in Today’s Market

1.8.1 Types of Automated Testing Tools

Depending on where in the software lifecycle a development team would like to introduce an automated testing tool, the types of tools can be divided into tools for the analysis and specifications gathering phase, others for the requirements definition phase, design phase, implementation phase, and finally quality assurance and debugging phase. There are also metrics and testing organization tools to aid in the process of managing the testing effort. Some tools are targeted for specific purposes like test data generators [57], simulators and prototyping tools.

There are dozens of automated software testing tools in the market today. The tools range in their functionality between functional-black box testing, to source code analysis tools. A good list can be found at the Aptest: Software Testing Resources and Tools website [25]. A lot of the available tools record and play back test scenarios the tester inputs while doing exploratory or scenario testing. Regression testing is the most commonly automated testing technique.
Some of the tools are targeted towards detecting run-time errors and memory leaks. Among those tools are Purify from Rational Software, C Verifier from PolySpace, SWAT from Coverity, Insure++ from Parasoft and GlowCode from Electric Software.

Other tools are test case organization tools, to help in the organization of the testing process. There are a number of automated test data generators and a lot of research in the automation of test data creation [69-72,80-85,87-95]. Test data generators generate data to simulate real life data in operational systems for the purpose of testing. Simulators of test environments are also available as well as automated stress and load testing tools. The following sections include a more detailed look at some of the features and underlying technology in several of the most widely used tools.

1.8.2 Design and Visual Modeling Tools

Design and visual modeling tools are mainly used in the analysis and design phases. These tools help the development team as well as the end users to reach a common understanding of the functionality of the system under construction. Some of the available tools also generate the system backbone from the design generated by the visual modeling tools, including the database tables, classes, modules, etc. The programmers can then write the code in the designated locations. Rational Rose and Oracle Designer are two of the common design and visual modeling tools available in the market.

1.8.2.1 Rational Rose

Rational Rose provides a channel to communicate a system’s architecture between various parties of the development team, from analysts and designers to developers.
and other team members. Rational Rose allows the designers to visually map system architecture to unified models and diagrams using models similar to UML. The diagrams go in their detailed level only to define the interaction between the different parts of the software. You can write code in the subroutine definition area, and thus when you choose to generate the code from Rational Rose, the code that you wrote will also exist where it should in your application. However, Rational Rose does not go into the details of the design or interpret the functionality of the software as it does with the higher level design. Accordingly, the testing tools that are now available and produced by Rational, do not depend on the specified functionality, but on the backbone, and the actual implementation and user scenarios.

1.8.2.2 Oracle Designer

Oracle is one of the leading companies in database and information systems solutions. One of the products offered by Oracle is Oracle Designer. As stated by Oracle, Designer is “a complete toolset to model, generate, and capture the requirements and design of enterprise applications” [26]. Oracle Designer has many modeling tools to help in the design and analysis capture of a database application. Among the tools included in Designer are the process Modeler, Dataflow Diagrammer, function Hierarchy Diagrammer, Entity relationship Diagrammer, Design Editor, Dependency Manager and Matrix Diagrammer. None of the above tools captures the detailed design and the functionality of the application, in a way that could be interpreted by an automated testing tool. The diagrams and matrixes are used to communicate system requirements and design between the different development team members, but not
for an automated testing tool, that can intelligently understand the specifications and
design of the software under construction and accordingly generate good test cases to
test it. In the design editor, one could design the forms and the components of the
application’s windows, etc. One can also determine what type of triggers run, and
what triggers them, but the code and the meaning of the implementation in those
triggers can either be written in plain English as comments for the programmers or can
be written in actual PL-SQL code. However, what is written in the triggers or stored
procedures in designer, whether it is the actual code or plain English description, is
stored as is to allow for it to exist after the user chooses to generate the application
from the stored design in Oracle Designer.

Designer automatically generates a script for the specified database tables, constraints,
modules for the application, and also forms, triggers, etc. Designer thus stores the
structure of the application, whether it is the database schema architecture or the
higher level process interaction and application dataflow and user interface.
Accordingly, Oracle Designer lacks the feature that is proposed by this research to
create a well defined representation of the detailed design of the application under
construction.

1.8.3 Automated Testing Tools

It was essential to survey the current automated testing tools in today's market, in order
to identify where improvements can be made. The conclusions reached after surveying
the market guided much of this research. The products of the most popular vendors of
automated testing tools are discussed in this section, including Mercury Interactive, Rational Software, Segue and Compuware.

In the United Kingdom, Mercury Interactive holds at least 50% of the market share in software testing tools. They have a number of automated test tools, including TestDirector [45], WinRunner [46], QuickTest [47] and LoadRunner [48].

Rational Software offers the most complete lifecycle toolset, including testing for the windows platform. They are the market leaders in object-oriented development tools. The leading testing tools available from Rational Software include Rational Robot, Functional Tester, Performance Tester and Rational Purify. Rational Robot is a test management tool that gives test cases for objects such as lists, menus, and bitmaps. It also provides test cases specific to objects in the development environment. As described by Rational [41] it is a "General purpose test automation tool for client/server applications". Functional Tester, on the other hand is an "automated functional and regression testing of Java, Web and VS.NET WinForm-based applications", as described by Rational. It aids the process of test script writing and managing,[42]. Performance Tester verifies application response time and scalability, it aids in emulating a real time web-based environment where lots of users are concurrently using the software. Rational Purify aids in the detection if memory leak and memory corruption for Linux and UNIX [44].

Compuware's major product is QARun. Similar to other automated testing tools, QARun depends on scenario testing, by recording the user’s actions and respective system
responses into test scripts to test some of the application’s functions [52]. Compuware also has a load testing tool.

Segue is another organization that provides some comparable software testing tools to Mercury Interactive and Rational Software. Among the products that it offers are SilkCentral Test Manager, SilkTest 5, and SilkPerformer. Comparable to Rational Robot, SilkCentral is a test management and organization tool [49]. SilkTest is comparable to QuickTest, and Rational Functional Tester. It is a regression testing tool that records and plays back the test cases [50]. SilkPerformer is an automated load, stress and performance testing tool [51], comparable to Rational's Performance Tester and Mercury's LoadRunner.

All four major automated testing tools vendors have products similar in functionality to the products of their competitors. The tools available are mainly categorized into four major types; namely, test management, capture and replay, functional and regression testing and performance or stress testing tools. Table 1.1 summarizes the functionality of the different testing tools. Test management tools like Mercury's TestDirector, Rational TestManager, SilkCentral by Segue and QACenter by Compuware all organize the testing process. Test management tools as described by Mercury, thus provide a consistent, repeatable process for gathering requirements, planning and scheduling tests, analyzing results, and managing defects and issues [45]. Testers can create test cases and log them in the test management tools to organize the testing process.
Table 1.1: Automated Testing Tools

<table>
<thead>
<tr>
<th>Type of Tool</th>
<th>Organization</th>
<th>Test Management</th>
<th>Capture and Replay</th>
<th>Functional &amp; Regression Testing</th>
<th>Performance &amp; Stress Testing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mercury Interactive</td>
<td>TestDirector</td>
<td>WinRunner</td>
<td>QuickTest</td>
<td>LoadRunner</td>
<td></td>
</tr>
<tr>
<td>Rational Software</td>
<td>TestManager</td>
<td>Rational Robot</td>
<td>Functional Tester</td>
<td>Performance Tester, Purify</td>
<td></td>
</tr>
<tr>
<td>Segue</td>
<td>SilkCentral</td>
<td>SilkTest</td>
<td>SilkTest5</td>
<td>SilkPerformer</td>
<td></td>
</tr>
<tr>
<td>Compuware</td>
<td>QACenter</td>
<td>QARun</td>
<td>TestPartner</td>
<td>QACenter</td>
<td></td>
</tr>
</tbody>
</table>

Capture and replay tools usually automatically generate test scripts from test scenarios captured during a tester's navigation through the software. Examples of the commonly used capture and playback tools are Mercury's WinRunner, Rational Robot, SilkTest by Segue, and QARun by Compuware.

Functional and regression testing tools are also common tools in software test automation. Regression testing tools run test scripts after issues have been resolved, to determine whether the fixes caused other problems in other parts of the software that were previously running correctly. Examples of regression and functional testing tools are Mercury's QuickTest, Rational Functional Tester, Segue's SilkTest5, and Compuware's TestPartner.

Performance and stress testing tools emulate an environment with thousands of users concurrently using the system. Examples of performance and stress testing tools are Mercury LoadRunner, Rational Performance Tester, Segue SilkPerformer and Compuware QACenter. Rational Purify checks for memory leaks.
1.8.4 Disadvantage of Record and Playback Test Automation Tools

In Section 1.8.3, the functionality of some of the most common software testing tools were outlined. However, none of those tools automatically and intelligently create test cases based on an understanding of what the software under construction is intended to execute. Some of those available tools record and playback test scenarios produced by the navigation of testers through the product while testing the product. In this case, coverage relies on how well the tester or user goes through the different and possibly numerous system scenarios. The system responses have to be judged by the tester who determines whether the system passed the test or not, based on his understanding and knowledge of the specifications. The tester thus, also acts as the oracle who decides whether the system passed that specific test, generated by the scenario in question.

1.9 Related Literature

A number of researchers in the field of software testing emphasize the need and the advantages of having a machine readable mapping of software specifications [31, 34, 38]. Other researchers describe techniques to compare software to its formal specifications as a good means of software testing [35,39]. Other techniques, such as those found in [37,40] can be deployed to work efficiently and automatically, given a software’s specification in the form of a machine-readable format, as that described in this research. In order to automatically test database operations, Chays et. al. in [27, 28] derived the software specifications from the database schema and program code, and a tester inputting relevant heuristics. The authors noted that it is not possible to fully automate the testing and
verification process unless the formal specification is given in full, which is the focus of this research. A similar approach to testing automation was illustrated by [36], in which the authors depended on programmed embedded SQL statements in the code, assuming it to be the correct source upon which to generate test cases. The weakness of this approach and the latter, lie in the fact that the test cases are generated based on the resulting software not from the specifications, as would be the result of adopting the techniques in this research.

In [33], Hung expanded on the work of Chays et al. in [28], designing an input generator for database applications. Hung emphasized the need to compute the expected output automatically. Similar to Chays et al. in [28], Hung generated test cases based on the testers values for the system inputs in their respective equivalence classes. The test cases where not generated automatically based on the specifications. The research in [58,59,60,61] did not address the problem of automatically checking the results of the tests to verify whether the results are correct compared to the specifications. A number of researchers investigated the automation of testing web database applications, including [62]. Also regression testing of database applications was researched in [68]. Define-use path testing was used in [63] to test database driven applications. In [64] Cabal and Tuya tested the SELECT operation in database applications. Automated test case generation techniques where introduced in [32] to test the GUI of applications.

1.9.1 AGENDA: A Test Generator for Relational Database Applications

Chays, Deng and Frankl carried out a research titled “AGENDA: A Test Generator for Relational Database Applications” [27,28,54,55,56]. The tests generated in their
research were targeted towards database operations or application queries (insert, delete, update and select statements). There were no test cases targeted to test the application code itself, in which these operations are embedded. The tests generated in Chays’ research depend on database input and output states, and do not compare the input or output to those specified by the design. The research does not tackle other types of operations where the output database state is not changed, but files, printed reports, magnetic card programming, etc. are the forms of output. Real world applications that use numerous types of inputs and outputs, besides database state changes will benefit only partially from AGENDA. AGENDA depends on the tester to specify whether the test passed or not. Accordingly, the tester is the oracle for AGENDA. AGENDA generates tests based on application code not the specifications and design of the software, and thus would not recognize errors of omission (parts that are designed but never programmed). When the tests are derived from the actual code, the tool will also not recognize errors of commission where the programmers wrote parts of code that were never included in the design document. Finally, AGENDA uses two types of functional testing techniques, namely, boundary value analysis and equivalence class testing. Not all faults in the software are revealed by these two types of testing alone. Boundary value analysis and equivalence class testing are good techniques to use in combination with other techniques to best test the software.
1.9.2 Executable Software Specifications

One of the major goals of the dissertation is to represent software specification in a way that they can be reasoned with by automated software testing tools, to generate intelligent test cases. Intelligent test cases are a minimal set of useful test cases that collectively cover all the software and are capable of identifying errors. Accordingly, it was necessary to review the literature for automated software specifications. In [96], Corning describes how Microsoft internally test their systems by using a technique that automates the execution of human entered software test case specifications. Accordingly, what is automated is the generation of test cases based on the specifications of each of the test cases. In other words, the test cases are not created using a computationally intelligent technique to generate interesting and important test cases, because the software specifications are not executable (though the test cases specifications are executable). In this situation, the testers are the ones that create test cases, which can be automatically re-run with different parameters. The parameters of each test case are the test case specifications, or test specifications in short. Those parameters are executable. Accordingly, automated tools can read the specification of a test case, including the input variable names, possible values, pre-conditions, as well as other factors, and create multiple test cases that match those specifications. Those multiple test cases can then be run automatically.

Formal specifications are not readily executable [97]. Fuchs believes that formal specifications should be executable if possible [98] and argues against Hayes and Jones
in [97] who wanted to make the distinction between human-readable specifications and
machine-executable prototyping. Fuchs also indicates that the executable specifications
in his research [98] are at the same high level of abstraction, the aim of the possibly
executable specifications is for rapid prototyping and to facilitate the validation process.
In the early nineteen nineties, automated testing and automated test case generation was
not the purpose of those techniques of trying to execute the specifications, the arguments
were mainly targeted towards rapid prototyping, not automated testing.

In [99,100], the authors describe a mechanism to transform formal specifications to
executable code. The result is not however, a means of representing the specifications to
an automated tool to reason about and generate intelligently test cases from it.

1.10 Research Goals and Contributions

After surveying today’s market for available software testing tools, none were found to
generate the tests independently from specifications. The available tools are either code-
coverage tools or capture/playback tools. They all rely on the implemented software either
to create tests based on white box testing and the implemented code in question, or on the
testers to provide good scenarios or test cases and their corresponding expected output.
Accordingly, one can fairly make the assumption that currently the term “automatic testing”
is misused to refer to automatic test execution and often misapplied to techniques that only
partially automate portions of the testing process. After surveying the available tools, it was
found that none of the tools in fact automatically generates the test cases for the software;
meaning that, none of the available tools creates test cases based on its ability to interpret
what the software under construction is implemented to achieve. The only tools close to
automatically generating test cases, are those for stress testing [29], which create test cases to
see how the software will react when a huge number of users log-in at the same time for
example, or for database systems, creating a huge database with very big tables, etc. These
tools, do not test the actual functionality of the software and whether it performs its tasks
correctly or not, they just generate and execute stress tests. Basically, none of the available
tools can understand the specifications and the software design, and accordingly create its
own set of test cases that intelligently test and achieve good test coverage of the software.

The main reason why such an intelligent test case generator does not exist is that there does
not exist a means of representing the specifications, requirements, and design in such a way
that it could be interpreted, understood and used by an intelligent tool to generate test cases.
Creating this machine readable and machine-interpretable representation of a system
specifications and design is one of the main goals of this research. The research contributions
are discussed in this section. In Section 1.11, the contributions are listed together with the
location in the dissertation where each contribution is discussed.

The dissertation research goal is thus to create a unified, machine readable representation of
a software application’s unit’s design and specifications; whereby an automated testing tool
could interpret and understand the software’s units’ design and specifications and
automatically generate intelligent test cases to test the specified units of the software and
determine whether the actual output matches the expected specified output; and to use the
machine readable specifications in automated testing tools to show its use. One of those
tools is to automatically generate constraints and enforce business rules for database applications at the database level. Another one of the automated testing tools is to use the specification representation to show errors of commission using a reverse engineering approach.

An automatic test case generator using the unified specification representation (SpecDB) will guarantee good coverage of the software, since it can generate test cases for all the specified functionalities of the software, utilizing the specifications not the implementation. Such an intelligent test case generating tool can easily find errors of omission, where the programmers did not implement parts of the software. It can also spot errors of commission caused by programming unspecified functionality in the software. Both of the latter types of errors would never be spotted if the tests were not generated based on the program specifications and design. Finally, such an intelligent test-case generator can spot faults of commission caused by errors in programming a specified functionality incorrectly.

The goal of this dissertation should not be misinterpreted as a requirements verifier. Requirements verifiers test the requirements document for clarity, consistency and completeness [30], however, they do not check the actual implemented software based on these requirements. A first step before starting to represent the specifications and the design in a way interpretable by a testing tool is to use a requirements verifier to check the accuracy of the specifications and the requirements [66]. Hence, the goal of the dissertation is to represent (formal) software specifications in a machine readable format to aid in the testing and test result verification processes and to create useful testing tools that make use of the
machine readable software specifications to intelligently guide the testing process and assure better software quality.

Accordingly, the first objective is to create a standard process for producing and storing UML or formal specifications and other software specification and design constructs and detailed functionality specification. In order to create an intelligent automated test case generation tool for software testing purposes, there has to be a way for the tool to understand what the software is built for, its functionalities, inputs, outputs, processing scenarios, etc. Accordingly, the research objective is to provide that standard way of representing specifications, in a way that could be interpreted by an automated test case generator. This representation of specifications is detailed in Chapter 2, SpecDB: A Database Representation of Software Specifications.

This dissertation includes extensions to the work by Chays, Deng and Frankl on AGENDA[27,28]. One of the uses of SpecDB, is to create an automated tool to generate intelligent test cases from SpecDB based on its understanding of the functionality of the software under test, and test the application units containing both imperative-like code (programming constructs like if, while, etc. and abstract data types, etc.) and also including the database operations. This proposed solution can compare actual program outputs to the specified design, stored in SpecDB. Therefore, the tool can automatically make the decision whether or not the test passed, based on its understanding of the available querable design, represented in SpecDB. Based on the specification, automated testing tools using SpecDB can spot errors of omission and commission immediately. In their research [27], Chays and
his research team, wrote: “Unless there is a formal specification of the intended behavior of the application, including a specification of how the database state will be changed, it is not possible to fully automate checking of the application’s results.” Again, representing the formal specification they referred to, in a machine readable format, is the first objective of my research to make it possible to automate the checking of the software’s results. The second objective is to use the representation of the design and specifications to create an intelligent business rules generator for database applications. The proposed representation of the analysis and design of software will be based on a database structure that can be easily queried. Business rules and constraints will automatically be generated to enforce those constraints at the database level, not the application level.

The final part of the research is dedicated to studying the feasibility and effectiveness of structural test case generation, based on the level of detail given by the software specification and design. From the actual implementation, a reverse engineering approach is used to show all the software output, and post conditions after a test is executed. The output and post conditions are compared to the specified behavior if represented in SpecDB and accordingly, errors are automatically spotted and code segments that caused the errors are isolated. The automated structural testing techniques used in the reverse engineering automated testing approach are dataflow testing and path testing.

In Figure 1.2 the contributions of the research are represented. The design of SpecDB is detailed in Chapter 2. The software specifications are stored in SpecDB manually or automatically using the formal specification translation algorithm in Chapter 3. The
usefulness of the SpecDB representation for automatic constraint generation and testing based on reverse engineering is demonstrated using two applications described in Chapters 4 and 5, automatic constraints generator and the reverse engineering testing tool, respectively. Finally in Chapter 6, enhancements to available tools in the literature like AGENDA, using SpecDB are proposed.

1.11 Objectives and Accomplishments

In Order to achieve the research goal described in Section 1.10, the following research objectives were accomplished:
1. Represented specifications in a machine readable format. This is detailed in Chapter 2, the design of SpecDB.

2. Created a translation algorithm to transform formal specifications written in a formal language like Z, to SpecDB. This is described in Chapter 3, storing specifications in SpecDB.

3. Created useful applications that make use of the data stored in SpecDB.
   a. In Chapter 4, an automated tool is introduced which generates constraints and business rules to enforce them at the database level as opposed to the application level, for better reliability.
   b. Also in Chapter 5, a reverse engineering approach to testing database applications is described. It uses specifications to intelligently test software.

4. Suggested enhancements on published work in the field of testing database applications, using the techniques described in this research are presented. In Chapter 6, some enhancements are proposed to AGENDA [27, 28] as an example.
CHAPTER 2
SPECDB: A DATABASE REPRESENTATION OF
SOFTWARE SPECIFICATIONS

2.1 Introduction

From the market survey of existing testing tools in Chapter 1, it was concluded that the current automated tools in the market lack an understanding of the functionality of the software under construction and testing. The literature review also showed how much of the research on automated tools can benefit from machine-readable specifications, which can reason with the specifications and automatically generate intelligent test cases and automatically determine whether the software passed the test based on its understanding of the specifications and design. In this chapter, one of the main contributions of the dissertation is presented, namely SpecDB, a database that stores part of the software specifications to aid in the process of automated testing.

The design of SpecDB introduced in this dissertation focuses only on the specifications required for unit testing database operations, and is not, in its current state a full representation of the complete specifications of a software under construction. However, the techniques used in this chapter can be extended to include more of the software’s
specification for other types of testing; for example, security, performance, system or integration testing, etc. In the following section, the motivation for the research in, and the development of the specification representation database, SpecDB is emphasized.

2.2 The Advantages of a Database Representation of Specifications

Since neither UML diagrams nor written formal specifications can be used as an input to automated testing tools, another channel is required to relay the specifications to automated tools. UML is not executable, nor machine readable. On the other hand, formal specifications are a complete representation, but are difficult to write and not used frequently. A database representation is chosen because it has the benefit of being machine readable and also not as hard to use as formal languages. Once the specifications of the software under construction are expressed in a machine-readable way, the specifications may be used and reasoned about by automated tools to test or generate the code for software applications automatically.

In order to create a software tool, that automatically tests parts of another software under construction, or automatically generates parts of the code for another specified software; the former has to have a good understanding of what the latter is intended to do. Accordingly, the automated software code generating / testing tool has to be given, in a machine-readable way, the specifications of the other software being built. The best way a software’s specification and design can be given in a machine-readable format is through an adaptation of the already established formal methods. Hence the idea of representing specifications in a machine readable format evolved. In this chapter a standard methodology is introduced
which represents software specifications in a machine readable format for the purpose of automated code generation and automated software testing. A database representation of the specifications was chosen as opposed to other forms or representation for the advantage of the inherently scalable nature of database designs. The design of SpecDB is scalable and it is easy to modify or append to the design in order to accommodate more forms of specifications. It is then very simple to update the queries created for previous versions of SpecDB, to accommodate the new changes. Runtime errors for database queries are not fatal, since they do not handle memory allocations like linked list representations. Another advantage is the use of the already stable and reliable database management systems available in the market, and the ability to create constrains and runtime triggers on database tables. Such constraints can aid in discovering contradictory, incomplete or ambiguous specifications.

Another use of SpecDB is to automatically represent programmed code and as an output, show the specifications of the programmed code. With this reverse engineering approach, SpecDB can also be used with already existing software. The advantage of this approach is to automatically test programmed software, giving the actual behavior for each operation. The analysts, designers and testers can then review all the software execution results, hence showing errors of commission that might not otherwise be spotted by normal testing techniques that deem a test successful if the specified behavior occurs. However, in some cases, the software errors lie in the fact that the programmed software does more than is specified. For example, an e-commerce application might perform its task of ordering an item, and also (as an unwanted side effect) send credit card information or passwords to
another non-secure location! The latter security leak cannot be spotted by conventional black box functional testing techniques. Using SpecDB in this reverse engineering approach will spot those errors of commission as will be shown in Chapter 5.

Code translation from one programming language to another can be done using the reverse engineering tool and SpecDB. Code from one language can be fed into SpecDB, the specifications can then be used to automatically generate code in another language.

2.3 The Design of the Specification Database, SpecDB

SpecDB is a database. The entities of the SpecDB database store the (formal) specifications of a software, necessary for unit testing. SpecDB is designed to efficiently and accurately represent those parts of the specifications shown in this chapter.

In Section 2.4, the reader will see how SpecDB covers and has a means to represent different forms of specifications. SpecDB is also designed to be scalable, generic, upgradeable and modifiable. Since the model is a database, it is easy to add more entities and/or more entity attributes to represent a specification that cannot otherwise be represented in the given SpecDB design in this chapter. As an example of SpecDB’s scalability, in Chapter 4 two tables are added and an existing table in SpecDB is modified to accomplish the task of an automated constraint generator using SpecDB.
The specification representation of SpecDB is designed to be the basis for the process of automated software generation and testing. Automated software code generation and/or testing tools can use SpecDB to understand what a software is designed to do, and hence can accurately test such a software by creating intelligent test cases with good coverage, and compare actual software behavior to its corresponding specified software behavior. With the use of SpecDB, the processes of automated software code generation and automate software testing can become more intelligent, efficient and reliable. In the design of SpecDB, constructs for both formal specifications and programming were taken into consideration. These rules and constructs can be found in [8,9,11-24,65].

Figure 2.1 shows an overall diagram of SpecDB’s design. Throughout Section 2.4 are figures showing parts of the relational database schema of the design of SpecDB. The relations between the entities are shown. Also the types of the fields in each entity are shown, numeric types are identified by the symbol “789”, text or character types are identified by “A”, and date or time types are identified by a calendar card. Table keys are identified by a “#” symbol, and whether each field is optional or mandatory is identified by “O” or “*” respectively. There are some fields that are constrained to a list of valid values, for example, Is_Constant is a Boolean value that can be only T for true or F for false. On the left of the field name, if the value of the field is constrained to a list of valid values, a relation symbol is viewed. Figures 2.2 and 2.3 are focused in parts of Figure 2.1, for better readability. Throughout Section 2.4 each entity is described in detail with focused images.
Figure 2.1: An Overall Diagram of SpecDB’s Design
Figure 2.2: A Focused Diagram of the General SpecDB Design (Left Section)
Figure 2.3: A Focused Diagram of the General SpecDB Design (Right Section)
2.4 Entities in the Design and How Each Represents Specific Software Requirements

In this section, each of the entities of the SpecDB design is defined and explained in detail, including examples of values for each attribute and how to store the different components of a software specification. First, general entities are defined, among those are the operators, List_of_Tables, Table Description, Types, Types List of valid values (LOV) and restrictions. The SQL scripts for the creation of the entities of SpecDB and their corresponding constraints are provided in Appendices A and B respectively. The reader can refer to those constraints to see the allowable values of the attributes, this may aid in the understanding of the meaning of the attributes in each entity in SpecDB and what each represents.

2.4.1 Operator and Valid Operands

In this section, the entities of SpecDB that hold operators and valid operands are explained.

The Operator table holds the operators of operations used in any software (formal) specification or code. Operators differ from one language to another, and hence SpecDB identifies them with unique IDs to form a generic outcome that is not language dependent.

Figure 2.4: Representing Operators and Operator Overloading in SpecDB
Examples of operator symbols are: /, MOD, AND, NOT, etc. If an operator is overloaded, records are added to the Valid_Operands table. The subroutine that defines the operator overloading operation should be stored in the Function_Declaration table (described in Section 2.4.5), with a subroutine name, the same as the operator symbol or name.

The over_loaded attribute in both the Operator and Valid_Operands tables is a Boolean variable that can be True or False. If more than three operands are involved in the operator overloading process, another attribute to represent the Operand4_Type can be added to the Valid_Operands table. The types of the operands have to reference records in the Types table (described in Section 2.4.3).

2.4.2 List_of_Tables and Table Description

The List_of_Tables entity in SpecDB stores all the software/user-defined sets and tables. The Table_Description entity stores the names of the fields of the user-defined tables and sets, and their respective types. In Chapter 4, this table will be extended, to aid in the process of the automated creation of database level business rules and constraints.

![Figure 2.5: Representing User Tables and Sets and their Description in SpecDB](image-url)
2.4.3  Types, Types_LOV and Type Restrictions

The Types entity stores the non-standard user-defined types. This entity stores types that are not sets, arrays, linked lists, or any type designed for data storage and retrieval. SpecDB represents the latter storage types as user-defined tables in the database, and stores their descriptions in the List_of_Tables and Table_Descriptions entities (described in Section 2.4.2). The Types entity stores user-defined types like month names, days of the week, etc. Each of those user-defined types has a basic type, e.g. String, Integer, etc. Also types can have size restrictions, e.g. a String of 20 characters. SpecDB allows for user-defined restrictions on types too. User defined restrictions can be one of the following:

2.4.3.1  List of Valid Values (LOV)

In the first kind of type values restrictions, the values of a variable of a specific type are constrained to only values in a defined list of valid values. A type can be restricted to a List Of valid Values, i.e. any variables of the specified type can only take on one of the values in the LOV. For example, the type Month_Name can be either January, February, etc. In this case the Has_LOV attribute value of the Types entity for the Month_Name record is set to T for true and the valid months names are placed in the Types_LOV entity, each in a separate record.

2.4.3.2  Other Restrictions

Types can also be restricted to a set of valid values that are better specified by ranges. In this case, the ranges are stored in the Types table in their respective fields to define
the allowable range. For example, for a Month_Number type, valid values are between the range of 1 o 12. Hence the Range_From_Int in the Types Table in the record defining the months is set to 1 and Range_To_Int is set to 12. A range can also be specified, with inclusions, e.g. all even integers between 0 and 100 including 0. Also exclusions can be defined from certain defined ranges by using the Type_Excluded_Ranges Table.

Figure 2.6: Representing Types, Type Restrictions and Types Valid Values in SpecDB

The Types_LOV table contains a record for each value in the list of valid values for each type with LOV restrictions. The table has been de-normalized to include values of all basic types, based on the “type” attribute value in the Types table, the value of the LOV will be stored in the relevant attribute. For example if the type of the Month_Number Type is Integer, then there will be twelve records in the Types_LOV table, each record will have a unique value between 1 and 12 stored in the Int_Value attribute.

40
2.4.4 Variables and Restrictions on Variable Values

In SpecDB, the VAR entity stores the specifications of a used variable. Including, its type, that can be also a user-defined type already defined in the Types entity. The SpecDB design, takes into account the fact that in the same software, many different variables can have the same name, but defined in different contexts or sub-routines. This entity stores qualities each variable should possess. This aids automated testing tools in creating test cases that make sure that in all the software’s execution cycle and in all its states, each variable maintains its designed properties. If a variable is part of a class or an object, this can be specified, together with its encapsulation property; public, private or protected.

Variables can have pre-defined types, but in turn also have their own restrictions. SpecDB has the flexibility to define restrictions on both types and variables. A variable can have a LOV restriction or other restrictions as the ones previously discussed for the types restrictions.

A variable can also have a Table Restriction, i.e. the variable can only take on values that are stored in a specific field in a user-defined table. The Var_LOV table stores preset values that are preset before a software is executed, e.g. names of the months of the year. On the other hand, a variable can be restricted to take on values that are determined as the software executes, these values are stored in user-defined tables; e.g. names of people in an address book, where the names are continuously added as the address book is used.
Those restrictions are stored in the Var_Table_Restrictions entity. SpecDB gives the flexibility for a variable to be restricted to take on only values from another table, or, by exclusion, to take any value that does not currently exist in the specified table. This is done by specifying the operator in the Var_Table_Restrictions entity.

![Diagram showing Var_Table_Restrictions and SpecDB relationships]

Figure 2.7: Representing Variables, Variable Restrictions and Variable Valid Values in SpecDB

2.4.5 Procedure, Function_Declaration, and Subroutine_IP_Description

Subroutines, or units are a major part of software specifications as well as the actual program code, and the focus of this research. In the SpecDB database, procedures differ from functions in the fact that functions return a value, whereas procedures are subroutines that do not return a value, yet can change the system state. If a procedure should change the state of the system, then the value of the Changes_System_State field, is set to true. Automated testing tools can use this specification to make sure that the programmed subroutines are implemented as designed. The inputs for both procedures
and functions are placed in the Subroutine_IP_Description entity. Inputs are represented with their respective orders using the IP_Order field.

For a function that takes two inputs like, Calculate_Interest_Only_Monthly_Mortgage (Loan_Amount, Rate), the record in the Function_Declaration table will be: (Calculate_Interest_Only_Monthly_Mortgage, 2, Real, 68, Monthly_Mortgage,F,F,F,, ) where F stands for False. There should be two records in the Sub_Routine_IP_Description table:

(Calculate_Interest_Only_Monthly_Mortgage,1,Real, Loan_Amount, F,F,T) and (Calculate_Interest_Only_Monthly_Mortgage, 2,Real, Rate, F,F,T). The two inputs can have a restriction to be greater than zero. In this case two records have to be placed in the Var_Restrictions table to describe those restrictions.

The output value returned by a function (OP) can also have restrictions. SpecDB provides a place to store these pre-determined restrictions on the valid allowable values a function can return.
2.4.6 Subroutine_IP_Instance, Designed_SR_Post_Conditions and Runtime_SR_Post_Conditions

Each time a sub-routine is invoked during the life of the software runtime, it is given different parameters. The Subroutine_IP_Instance is where the values, or the sources for the values of the parameters are specified for each one of those instances. For example, if on the seventh call to Calculate_Interest_Only_Monthly_Mortgage in the program specifications or program, the input is taken from the GUI text fields (:HouseValue, :NegotiatedRate), in the Interest Only window. Accordingly, the records in the Subroutine_IP_Instance table will be (7,1, Calculate_Interest_Only_Monthly_Mortgage, GUI, , :HouseValue, Interest Only,,,,,,,,)

The Designed_SR_Post_Conditions, holds the designed post conditions for subroutines, functions and procedures. Those will be compared to the actual post conditions during the testing phase to spot out software errors if there exists discrepancies between the designed subroutine post conditions in this table and the actual runtime subroutine post conditions, declared in the Runtime_SR_Post_Conditions table.

The Type of the post condition could be a modification to or execution of one of the following: GUI, DB, File, Report, Sequence, Message, Database Privilege or Role, Global Variable, Commit command, Savepoint command, Rollback, Run another program, Open a window or terminate the program execution. The source and the destination of the post condition are specified in this table.

If the subroutine should result in a database operation being executed, the unique ID for
that operation is placed in the DB_Operation_Num field, and the actual specifications for that operation is placed in the DB_operations and DB_Operation_Details tables.

The Runtime Subroutine Post Conditions Table is a duplicate of the Designed_SR_Post_Conditions, with the difference that actual post conditions that occur due to the execution of a function or procedure are stored in the latter. Each time a sub-routine is instantiated during runtime, an instance number is generated, and the input parameters used to invoke the sub-routine, is stored in the Sub_routine_IP_Description table.

Figure 2.9: Representing Sub-routines Instances, and Post Conditions in SpecDB

The post conditions of that instance are stored in the Runtime_SR_Post_Conditions table, using that same instance number. The fields in this table are exactly the same as those of the Designed_SR_Post_Conditions table, with the difference that the field Instance number is added, and is part of the key to this table. Also the data in the records
indicate the actual runtime post_condition behavior after a sub-routine is run, as opposed to the designed or the expected, detailed in the Designed_SR_Post_Conditions table. This table will be populated after the software is implemented and executed. An automated testing tool can make use of this table.

2.4.7 Dataflow, Triggers and Input_Output_Definition

In order to specify the order of operations of a unit in an application, the operations have to be given serial numbers. Accordingly, as a preparation stage for representing the specifications in SpecDB, unique dataflow numbers are given to operations in the (formal) specifications. Accordingly, SpecDB assumes that the program is numbered. Every line of the (formal) specification of a software as well as every line of code generated is given a unique serial number. SpecDB assumes that every line of code is a simple program statement, unnecessary bundling of code or specification on the same line is forbidden in this representation, to simplify the representation and enhance readability from SpecDB by automated tools. Accordingly, a preparation phase to simplify the specifications or code is necessary. The dataflow table stores what each line of the (formal) specification presents. SpecDB can also be used to represent actual lines of programmed code and automatically compare the actual behavior to the specified behavior using this reverse engineering approach where the Dataflow table would contain the actual programmed lines of code.

Each position is identified by a number, a successor number, and a type. The position type can be one of the following: Initial, Begin, Input, Assignment, Decision, While
Loop, For Loop, Run exe, Output, Subroutine Call, DB, open window, construct object, destroy object, call object method, end, end loop, end for, or terminal. According to the type of the position, the successor number can change. A position number can have many successor numbers and vice versa. For example, a decision point can have two results (true or false), and hence two successors, depending on the result.

Database management systems have structures similar to subroutines called triggers. The difference between triggers and other subroutines is that procedures or functions are called explicitly, sometimes with arguments, whereas triggers are executed implicitly when a pre-specified event occurs, also there are no arguments passed to triggers. For our design specification purpose, the trigger table stores the events that cause triggers to execute. Those cannot be part of the normal dataflow table, since they do not follow the same dataflow process as the rest of the software specification or code. However, this table will refer to the dataflow table once a trigger is defined, to adopt the same technique used to describe a normal subroutine body, to describe the trigger body. Also what is named smart trigger is described in this table. Smart triggers execute once a pre-specified event occurs to a GUI object. For example, when the mouse pointer enters a specific area in a window or when a button right click occurs, etc. If the LHS var is assigned to a GUI object value, the var_id is the place to identify that GUI object, since all GUI used vars are listed in the VAR table.
The Input_Output_Definition table describes how inputs and outputs to and from the software are represented. Inputs and outputs can be represented in many forms, file, database, through the GUI or to output devices like a printer (where a report is printed), or from other input devices. If the input or output is a database operation, then in the dataflow table, the line of code will be of type database operation, not input or output, and hence this type of input or output will not be represented in this Input_output_definition table.

2.4.8 Assignment and Calculation

Assignment operations and calculations are discussed in this section. A variable can be assigned to a constant or another variable, e.g. \( a := 65; \ b := c \). A variable can also be assigned to a calculation or a predicate statement resulting in a true or false value, e.g. \( a := 5 + b; \ e := (a \ \text{AND} \ b) \ \text{OR} \ (c \ \text{XOR} \ d) \). If a variable is assigned to a calculation, then
there might be a number of records in the Calculation table describing this calculation. A calculation identifier is provided in the Cal_Num field, and then at least one record is inserted in the Calculations table.

In the Calculation table, complex calculations are stored in the form of simple components. Each component involves only two operands and an operator. An operand can be simple like a variable or a constant or can be complex, and consists in itself of components. An example of a Simple Calculation is \( a+b \). The LHS is a variable, and thus “\( a \)” will be stored in the Left_Var_ID field. “\( b \)” will be stored in the Right_Var_ID field, and the operator ID for “\(+\)” will be stored in the Operator_ID field. Also \( a+109; \ a<\ b; \ a - \ mod(b); \) are all simple calculations. Examples of a complex calculations are: 

\[
\frac{a}{b} + c; \quad (U \ OR \ V) \ AND \ (Not \ W),
\]

are all complex calculations.

In the first example, \( \frac{a}{b} + c \), \( \frac{a}{b} \) is one component, and \( c \) is another simple component. Accordingly, there will be two records for this calculation in the Calculation table. One record with Component_Num =1, where Left_Operand_Type=“Var”, Left_var_ID is that of “\( a \)”, and Right_Operand_Type = “Var”, and Right_Var_ID is that
of “b” and operator_ID is that of the “/” operator sign. The other record with the same Cal_Num, but Component_Num = 2, will have Left_Operand_Type = “Comp”, the Left_Component_Num = 1, and the Right_Operand_Type = “Var” and the Right_Var_ID = that of “c”.

2.4.9 Predicate

The Predicate table described in this section stores complex predicate statements or database conditions. Each line of specification or code i.e. position, has a type that is stored in the Dataflow table. If the Pos_Type is either a Decision, While Loop, For Loop, or Database Operation, the statement details at this line of code are stored in this Predicate table.

In case the Pos_Type is a decision, a decision statement can have a simple or a complex predicate. An example of a simple predicate is:

\[
\text{If } a = b \text{ then …}
\]

In this case, there are no calculations, or several components. There is only one component, and the left and right operands are variables, or a variable and a constant.

An example of a complex predicate is

\[
\text{If } (a+b) < (d+11) \text{ then…}
\]

In this case there are several components. The components are themselves calculations. Records are inserted in this table starting with the lowest level components, i.e. the ones that contain only simple predicates, with just variables and constants, i.e. from the inside out. Finally the predicate that represents the relationship among the already described
components is inserted, using their respective component numbers. The technique is very similar to that used in the calculation table.

In case the Pos_Type is a loop, the beginning of a loop is a decision statement, determining whether another iteration through the loop should take place or to exit. The same discussion above for decisions is followed to describe a while loop predicate.

In case the Pos_Type is a database operation, the predicate table is where the WHERE clause of a database operation is stored. Similar to the above examples, the operands are database fields compared to one another, for example: S.ID = Emp.ID. In this example, both the left and the right operands are database fields in the S and Emp tables (or table alias). This example will be treated as a simple predicate, as discussed above. However, if the WHERE clause had many such simple predicates joined with AND or other operators, they will again be treated each as a component. The components will be
stored in this predicate table as in the above examples involving only variables, and/or constants.

Finally, if the Pos_Type is a set operation, the predicate table is also used to express operations on sets. For example,

\[ N \notin \text{Known}. \]

In this case, the Component_Type is a DB_Table. Database tables are sets. And the records are the tuples belonging to the sets. Left and/or Right_Table_Name are the fields from the predicate table used in this case.

### 2.4.10 Database Operations, Tables Involved in Database Operations

In this subsection the DB_Operations, DB_Operation_Tables and DB_Operation_Details tables of SpecDB are discussed. The DB_Operation_Tables table contains a list of tables involved in each DB operation defined in the DB_Operations table. The DB_Operations table stores database SQL Statements. Good references for SQL standards can be found in [73,74,75,76]. Since a database select, insert, delete, update, etc, each is considered one statement; this table is dedicated to describe those operations. There is one record for each SQL statement in the software specification or code.

The ‘Where’ clause is stored in the Predicate table, in this case the DB_Condition_Num is the same as the Pos_Num in the predicate table. This table is a representation of some of the database operations that can be programmed in a PL-SQL environment.
Following the same technique all the SQL Language functions can be represented in this table. The functions listed and accounted for by fields are just a few examples to show the feasibility of the approach. The DB_Operation_Details table contains the details of the Database operations described in the DB_Operations table.

For SELECT Statements, the variables to select are inserted in the Field_Name attribute. If there are n fields to select, there will be n records in this table with the same DB_Operation_Num. If the database operation was "Select *", then there will be only one record, and the field_name will be *. The INTO portion of the select statement, i.e. the variable where the selected value should be stored is indicated in the INTO_Var_ID attribute. The FROM clause is listed in the DB_Operation_Tables table, with the same DB_Operation_Num. Finally, The WHERE clause is stored in the Predicate table, with the POS_NUM = the DB_Condition_Num. If a cursor is defined, the Operation_Type is in the DB_Operations table is 'Cursor', it is treated the same as a select statement, with the addition of identifying the Cursor Name in the Cursor_Name field in the DB_Operations table. Also if a Fetch operation on a cursor is the DB Operation type, again the Cursor_Name field is populated, and all the details of which values to put from the cursor SELECT statement into variables are all stored in the DB_Operation_Details table.
For Insert Into Statements, the table name will be in the DB_Operation_Tables table, with the same DB_Operation_Num. If specific attributes (sequence) is specified, there will be a record in this table for each attribute. The attributes will be stored in the Field_name field. The corresponding value that should be inserted in this attribute, will be placed in the appropriate ‘type’_value field, according to its type.

If the DB statement is an Insert statement of the following type:

\[
\text{Insert into Table\_name}
\]

\[
\text{Select...}
\]

then the Insert_Select attribute will be set to true. Also the Insert_Select_DB_Oper_Num will be filled out and the Select operation will be defined in the DB_Operations table, under its Pos_Num.

For Delete DB operations, the table name will be in the DB_Operation_Tables table,
with the same DB_Operation_Num. If there is no where clause, then all the rows in the table will be deleted, and thus the Whole_Table field should be set to True. If there are some specific rows to be deleted, identified by a where clause, the where clause will be specified in the Predicate Table, under a POS_Num equal to the position number of the current Delete Statement in the code. This number will be also stored in the Delete_Where_Pos_Num field.

For Update DB operations, the table name will be in the DB_Operation_Tables table, with the same DB_Operation_Num. The attributes will be stored in the Field_name field. The corresponding value that should be inserted in this attribute, will be placed in the appropriate ‘type’_value field, according to its type. (if there are n fields to update, there will be n records in this table with the same DB_Operation_Num). The WHERE clause is stored in the Predicate table, with the POS_NUM = the DB_Conition_Num. In a query, the same table can be used twice, with different aliases. Accordingly, the key to the DB_Operation_Tables table is a numeric ID.

2.4.11 Classes, Objects and Class Relations

SpecDB can represent object oriented constructs; classes, objects, and relationships between classes. In this section, the entities of SpecDB that hold object oriented constructs are explained.
The Classes table holds the names of the classes, the rest of the information of the class are stores in their respective locations, with an identifier pointing to their respective classes. For example, class variables, methods and encapsulations rules are stored in the Var table, in Section 2.4.4 and either the Procedure or the Function_Declaration tables respectively, explained in Section 2.4.5. Relationships between classes, like inheritance, association, aggregation, etc, are stored in the Class_Relationships table. In the Class_Relationships table, two classes from the Classes table are referenced in Class_Name1 and Class_Name2. The relationship type between those two classes is stored in the relationship attribute in the Class_Relationships table. This attribute has a list of valid values representing the types of relationships among classes, e.g. inheritance, encapsulation, etc.

![Diagram](image)

Figure 2.14: Representing Object Oriented Classes and Objects in SpecDB

### 2.5 Conclusion

Chapter 2 contains the design of SpecDB, a database designed to hold parts of a software’s specification for the purpose of unit testing. Through the use of SpecDB, automated tools can become more intelligent in generating and testing programs, based
on an understanding of what the software under construction is supposed to do, by design. The database design of SpecDB is scalable and thus, allows for any additions to accommodate more programming constructs or requirements, new forms of input or output, etc., as they evolve in the future. The motivation behind building SpecDB has been expressed in Chapter 1, mainly to aid the process of automated intelligent test case generation based on a machine readable specifications. In order to create these machine-readable specifications to allow for automated reasoning, SpecDB was created. A scalable representation of the specification was chosen for this task, namely the use of a database specification representation. The standard technique of representation for portions of the software specifications to aid in the process of unit testing has been detailed in this chapter. Each of the entities of the SpecDB design has been explained in detail. Representing software specifications in this manner aids the process of automated testing and automated code generation. In Chapter 3, a translation algorithm is described, to translate formal specifications to the SpecDB representation, showing how a formal software specification can be automatically represented in SpecDB.
In this chapter, an algorithm that translates formal software specifications to the SpecDB representation is described. The design of the underlying database that stores the formal specifications and design, SpecDB was detailed in Chapter 2. SpecDB is a database created to represent and store parts of the formal specifications of software, in a queryable / machine-readable format. The Z formal specification language has been chosen as an example for the translation algorithm. Similar algorithms to that shown in this chapter can be created to translate other formal specification languages such as VDM to SpecDB.

The outcome of the translation algorithm is a machine readable representation of the original formal specifications. It is worth noting that the users of SpecDB do not have to learn formal languages. The specifications can be entered directly in SpecDB as easily as populating any database with the correct data. However, this chapter gives an algorithm to transform formal specifications to the SpecDB representation, if formal specifications were used in the specification and design phases of the software development. The representation of SpecDB for formal specifications, aids in the processes of both automated software code generation and automated software unit testing, based on the actual software specifications.
3.1  The Translation Algorithm From Formal Specifications to SpecDB

The entities of the SpecDB database store the (formal) specifications of a program. SpecDB is designed to efficiently and accurately represent portions of the software formal specifications required to carry out the testing techniques. In Section 2.4, the methodology of how SpecDB covers and represents different forms of formal specifications is shown. In this section, formal specification fragments will be given together with their corresponding algorithm fragments to store these specifications in SpecDB. SpecDB was also designed to be generic, upgradeable and modifiable as described in Chapter 1.

The examples given from the Z formal specification language in Section 3.4 are from [18,19,23]. In each subsection in Section 3.4 the Z specification syntax is shown together with the algorithm steps. The algorithm is numbered to identify it from the explanation text.

Not all the syntax of the Z specification language is discussed in this chapter. Examples are given for some of the algorithms. Numbers corresponding to the step in the translation algorithm in that specific case are shown to show the reader how each step is executed. Examples of the most commonly used syntax to document specifications are discussed. These examples are demonstrated to prove the feasibility and show the reader how an automated translation tool can execute this algorithm and transform formal specifications into the machine-readable SpecDB representation. The main idea of the algorithm can be understood from the steps of the algorithm in the following subsections. Similar algorithms can be created to translate other formal specification syntax to the SpecDB representation as needed. The understanding of the SpecDB design detailed in Chapter 2 is needed to generate extensions to the algorithm explained in this chapter.
3.2 Assumptions and Restrictions

All operators are prefixed with the “&” sign to distinguish them from variables, user defined types and schemas. User defined variables and types should not have dashes, spaces or Z symbols.

Var_name? is an input.

VAR_name! is an output.

No user defined variable can end with 1. Variables ending with 1 are temporary variable names given by the algorithm, until the variable is given a name in the specifications to follow its type definition.

3.3 Preparing the Formal Specifications for Translation

Before translating the formal specifications into the representation of SpecDB, the following preparation phase tasks have to be performed:

1. Going through the formal specifications, expand complex operations to simple operations.
2. Number the specifications, giving the main control (main menu) the number 1, and increments of 10 to the rest of the specifications in order to allow room for future enhancements, and add on specifications where needed.
3. Put either the Δ or Ξ symbol before a schema name that is the first line after another schema definition, according to the meaning of the schema.
4. Add necessary brackets to complex calculations and predicates to accurately
describe the required output. A couple of brackets should enclose one operator and one LHS operand or component within brackets and one RHS operand or component within brackets.

3.4 Storing Formal Specification Statements into SpecDB

In the following subsections of Section 3.4, a number of formal specification statement syntax is provided together with its corresponding translation into the SpecDB representation. The most commonly used statements are discussed. Other formal specification statements can also be translated into the SpecDB representation by expanding on the given algorithm with the same scheme after understanding the entities of the SpecDB design detailed in Chapter 2.

3.4.1 Defining Schema Names and Operations

Program units in the Z formal specifications language are given the name schemas. In Figure 3.1, the unit or schema name is AddBirthday.

![AddBirthday Schema](image)

Figure 3.1: AddBirthday Schema

To represent Z schemas in SpecDB, the following steps in the algorithm have to be executed.
1. Set Current_SR to the Schema Name. (SR stands for Sub-Routine.)

2. Insert the name of the Schema in the Procedure table, setting the num_of_inputs field to 0 and Changes_System_State to F for False. The operation remains a procedure until proven to be a function, then it is removed from this table and placed in the function_declaration table instead.

Example: in the “AddBirthday” schema in Figure 3.1, executing the above algorithm, will result in the following state:

2. Insert record in the Procedure table (AddBirthday,0,F)

3.4.2 Types of Schemas

After specifying the name of a schema, as in Section 3.4.1, the nature of the schema is specified. For example, in Figure 3.1, in line 70 of the AddBirthday schema:

\[ \Delta \text{Schema1 or \Delta BirthdayBook} \]

This means that AddBirthday is part of another schema (program in this case) i.e. AddBirthday is a subroutine definition. It indicates that after the execution of the AddBirthday schema, a change will occur in the state of the system, defined by Schema1, or BirthdayBook in this case. In this case, i.e. a \( \Delta \) or change in the state, the following should be executed:

1. Update the Function_Declaration or Procedure record for the Current_SR, set Changes_System_State to T for True.

Example from Figure 3.1: Update the record of the Current_SR (AddBirthday) in the
Procedure table from (AddBirthday,0,F) to (AddBirthday,0,T)

Another type of schema exists, one that does not change the state of the system, identified in Z by:

$$\Xi \text{ Schema1}$$

This symbol indicates that after the execution of the schema, No change should occur in the state of the system, defined by Schema1. In this case:

1. Update the Function_Declaration or Procedure record for the Current_SR, set Changes_System_State to F for False, if not already F.

3.4.3 User-Defined Types

User-defined types are built from the basic system-defined types, possibly with some constraints. In the Z syntax below, the basic system-defined types for these user-defined types are not specified:

$$[\text{Type1}, \text{Type2}..]$$

1. Insert a record in the Types table for every type until the closing bracket is reached.

Example: $$[\text{NAME}, \text{DATE}]$$

Two records should be inserted in the Types table as follows:

$$\{\text{NAME, , , N}\} \text{ and } \{\text{DATE, , , N}\}$$
3.4.4 Specifying Values for a User-Defined Type

A variable type can be a pre-defined system type, or as part of the specification, other higher level types can be defined. The Z syntax for values of user-defined types is as follows:

\[
\text{FreeType1} ::= \text{val1} | \text{val2} | \ldots.
\]

In this case all the values after the ' ::= ' sign form a list of valid values that FreeType1 can take, and implicitly specifies, that FreeType1 cannot be of any other value. In order to represent this list of values for a user-defined type in SpecDB, the following algorithm should be executed:

1. Insert a record in the Types table (FreeType1,,L ) where L is the value of the LRN field that specifies that the type has a list of valid values.
2. Set LOV_Count = 0
3. Increment LOV_Count. Also each time a “|” symbol is observed LOV_Count is incremented indicating that another valid value is specified next, then repeat step number 4.
4. Insert a record in the Types_LOV table (FreeType1,LOV_Count,,Val,.. ) Val is placed in the text_value field since the type is not specified. Repeat steps 3 and 4 if the next symbol read is a “|”

For example, line 290 below in Figure 3.2, the REPORT free type definition, has three possible values: \{ok, already_known, not_known\}.

\[
\text{REPORT ::= ok | already_known | not_known.}
\]

Figure 3.2: Report, User-defined Type Specification
3.4.5 Defining Sets or Tables

Sets can be defined to describe entities in the software design. An example of a set is the set of known people in the birthday book. Sets are defined in formal languages as follows:

\[ A: \mathcal{P}B \]

Where \( A \) is a set, and \( B \) is a type. In order to represent the set definition above in SpecDB the following algorithm should be executed:

1. If not already inserted, insert \( B \) in Types: type_name (leave all the rest of the attributes empty)
2. Create a new table, give it the name \( A \), with one attribute (key) of type \( B \) and name this attribute: \( A \).
3. Add \( A \) to the List_of_Tables table.
4. Add a record in the Table_Description table (\( A,A,B \))

Example: known: \( \mathcal{P} \) NAME

1. Output: record added to Types table: (known, , , )
2. Table ‘known’ created, with one attribute, known, of type NAME
3. Record added to List_of_Tables table: (Known)
4. Record added to Table_Description table: (Known, known, NAME)
3.4.6 Function Declaration

System functions are higher level processes or subroutines that collectively define the system operations. A function return a value, unlike a procedure. Functions are defined as follows in the Z formal specification language syntax:

\[
\text{Func1: } B \rightarrow C
\]

Where Func1 is a function or set, B is an input type and C is an output type. In order to declare a function in SpecDB the following algorithm should be executed:

1. Put all types (B and C) in the Types table (type_name), if not already inserted.
2. Consider variables B1 and C1 of types B and C respectively. Insert in the VAR table two records:
   (#,B1, B, Func1, ,T,....), where T is true for the is_input attribute and # is the next sequence number in the Var table
   (#,C1, C, Func1, , , T,....), where T is true for the is_output attribute
3. Insert in the Function_Declaration table (Func1,1,C,# ,C1,F,F,F). Where # is the VarID for C1. All restrictions are false by default till restrictions are declared.
4. Insert in the Sub_Routine_IP_Description table (Func1, 1,B,B1 ,F ,F ,F)
5. Add Func1 to the List_of_Tables table.
6. Create a new table, give it the name Func1, with two fields of type B and C, and give them the name B1 and C1.
7. Add a record in the Table_Description table (Func1,B1,B)
8. Add a record in the Table_Description table (Func1,C1,C)
3.4.7 Subroutine Input Restriction

The input of a subroutine (procedure or function) can have restrictions, based on the business rules. Subroutine Input restrictions are defined as follows in Z:

\[ A = \text{dom } \text{Func1} \]

In order to represent that input restriction above in SpecDB the following algorithm should be executed:

1. Current_Var_ID := the input var_ID for Func1
2. In the Sub_Routine_IP_Description table, update the fields of the Func1 function to reflect this restriction: (Func1, 1, B, , F, T, F) i.e. the input B has a table restriction.
3. Insert a record in the Var_Table_Restrictions table (#, Current_Var_ID, Func1, , I, BelongsTo, A, ) where # is the current serial number for this table.

Example: known = dom birthday

In this case the specifications state that the set of people already in the birthday book are the set of known individuals. New people added to the birthday book should not thus be already in it, to avoid multiple entries for the same person. Given the above specification, the algorithm trace should look as follows:

1. Current_Var_ID := 1;
2. Update the Birthday record in the Sub_Routine_IP_Description table: (Birthday, 1, NAME, Name1, F, T, F)
3. Add record to the Var_Table_Restrictions table: (1,1, Birthday, , 1, BelongsTo, Known, Known, )
3.4.8 Variable Definition and Subroutine Inputs

Variables can be defined as inputs to a subroutine, or in the main operation, they are defined in the Z formal specification language as follows:

\[ \text{Var1? : Type1} \]

In order to introduce the subroutine inputs above in SpecDB the following algorithm should be executed:

1. Insert in table Var (#, var1, Type1, Current_SR, ,T,,...) Current_SR is inserted in that record if the variable is defined in the context of an operation definition (Schema). Set the value of the Is_Input field to true.
2. Set Current_Var_ID = serial number given above (#).
3. If the variable is defined in the context of an operation, update the Procedure table, increment num_of_inputs, where proc_name = Current_SR.

For example, line 80 in Figure 3.1:

\[ \text{name?: NAME} \]

In this case a variable named: name is of the type NAME. This is a general variable definition statement. However in line 80 in Figure 3.1, it is defined within the scope of a subroutine definition, accordingly, it is an input variable to that subroutine. The trace of the algorithm for the above example should be as follows:

1. Add record in the Var table: (3, name, NAME, AddBirthday, ,T,....)
2. Current_Var_ID := 3
3. AddBirthday record updated in the Procedure table (AddBirthday, 1,F)
3.4.9 Output Definition

Variables can also be defined as an output of some subroutine. In Z output variables are defined by appending an exclamation mark to the right of the variable name as follows:

\[ \text{Var3}!: \text{Type2} \]

In order to define output variables in SpecDB the following algorithm should be executed:

1. Insert in table Var (\#,\text{var3},\text{Type2},\text{Current_SR} ,,T,,\ldots). \text{Current_SR} is inserted in that record if the variable is defined in the context of an operation definition (Schema). Set the value of the Is_Output field to true (T).

2. Set \text{Current_Var_ID} = \text{serial number given above} (\#).

3. If the variable is defined in the context of an operation, then this operation is a function, not a procedure. Add \text{Current_SR} to the Function_Declaration table, with the same data in its record in the procedure table. Set Op_type = \text{Type2}, and Op_Var_ID = \text{Current_Var_ID}. Set all restrictions to False in this record.

4. Remove \text{Current_SR} from procedure table

For example in line 180 in Figure 3.3, below:

\[ \text{date}!: \text{DATE} \]

Figure 3.3: FindBirthday Schema
In this case, it is understood from the specifications that FindBirthday is a function that returns an output of type DATE. The algorithm trace should result in the following:

1. Add a record to the Var table: (6, date, DATE, FindBirthday, , ,T,...)
2. Current_Var_ID := 6;
3. Record added to the Function_Declaration table (FindBirthday, 1, DATE, 6, date, F,F,F,F,..)
4. Delete the FindBirthday record from the Procedure table.

3.4.10 Setting Variable Restrictions

A variable can be restricted based on the business rules or software specifications. In Z, variables restrictions are defined as follows:

$$\text{Var1}? \notin \text{Set1}$$

The symbol '∉' can be substituted with any other operator. Any set operators can be expressed using the same method. Also, the type of the variable, it can be an input with the symbol '?' or an output variable with the symbol '!' appended to the right of the variable name. In order to represent the variable restrictions defined above in SpecDB the following algorithm should be executed:

1. Set Current_Var_ID = Var_ID of Var1 in the Var table, for the Current_SR if any.
2. Update the Var table for Current Var_ID, Set has_table_restrictions to true.
3. Insert record in the Var_Table_Restrictions table (#,Current_Var_ID, Current_SR, Var1, ,NotBelongTo, Set1, ). Current_SR is inserted in that record if the variable
is defined in the context of an operation definition (Schema). (Substitute NotBelongTo with any other operator used.)

For example in line 100 in Figure 3.1:

\[ \text{name?} \not\in \text{known} \]

The trace of the above algorithm should yield the following results:

1. \text{Current_Var_ID} := 3;
2. The record for \text{name} in the \text{Var} table is modified: (3, \text{name}, \text{Name}, AddBirthday, ,T,,T..)
3. Add a record to the \text{Var_Table_Restrictions} table: (2, 3, AddBirthday, Name, , NotBelongTo, Known,..)

### 3.4.11 Assignment Operation

The syntax for assignment operations in Z is simple, and defined as follows:

\[ A = B \]

Where \( A \) is a variables. \( B \) can be a variable or a constant. However, the same assignment syntax can be used for more complex constructs than simple variables. The following algorithm should be executed to represent an assignment operation in SpecDB:

1. Insert a record in the \text{DateFlow} table (Pos#, Suc#, Assign, ..) If this statement is the end of a schema definition (operation) then set Suc# to 1, i.e. main menu control or the Suc# from the calling subroutine.
2. Set \text{Current_Var_ID} = \text{Var ID} for \text{A} in its subroutine context, if this statement is defined in a subroutine.
3. If B is a variable, set Current_Var_ID2 = Var_ID for B in its subroutine context, if this statement is defined in a subroutine.

4. Insert record in the Assignment table (Pos#, Current_Var_ID,Var,..., Current_Var_ID2,..). If B is a constant then replace Var for Const in the RHS_Type field, and put the value in the correct location according to its type.

For example, in line 320 in Figure 3.4, result! = ok:

```
<table>
<thead>
<tr>
<th>Pos#</th>
<th>Success</th>
<th>result!</th>
<th>REPORT</th>
</tr>
</thead>
<tbody>
<tr>
<td>320</td>
<td></td>
<td>result!</td>
<td>ok</td>
</tr>
</tbody>
</table>
```

Figure 3.4: Success Schema

Below is the trace of the algorithm for the above example:

1. Add record to the DataFlow table (320,1,Assign,...)

2. Current_Var_ID := 9;

3. Add record to the Assignment table (320,9,Const,Text,....,ok,..)

### 3.4.12 Assignment Operation and Function Call

The following formal specification syntax is more complex than those discussed above. In formal specifications, we can define both a function call and an assignment at the same time, where the output of the function call is directly assigned to a variable, as in the below example:

```
Var2! = func1(var1?)
```
In order to define function calls and assign the output to a variable in SpecDB the following algorithm should be executed:

1. Insert a record in the DataFlow table (Pos#, Suc#, Assign, ..) If this statement is the end of a schema definition (operation) then set Suc# to 1, i.e. main menu control or the Suc# from the calling position.

Set Current_Var_ID = Var ID for Var1? in its subroutine context, if this statement is defined in a subroutine.

Insert record in the Sub_Routine_IP_Instance table (#,1,func1,Var, Current_Var_ID,.. ). Repeat this for all inputs to the function in order, incrementing the IP_Order field every time.

Inst_Num = # used in the above record inserted in Sub_Routine_IP_Instance .

Set Current_Var_ID = Var ID for Var2! in its subroutine context, if this statement is defined in a subroutine.

Insert record in the Assignment table (Pos#, Current_Var_ID, Func, , ,func1, Inst_Num, ….), where the RHS is set to Func, and FuncName to func1.

For example, in line 200 in Figure 3.3: date! = birthday (name?), the function birthday is called, given the variable name? as an input, the output of the function is stored in the output variable date!. Below is the trace of the algorithm for this example:

1. Add record to the DataFlow table (200, 1 , Assign,..)

Current_Var_ID = 5; where 5 is the Var_ID for date in its subroutine.

Add record to the Sub_Routine_IP_Instance table: (1,1,Birthday,Var,5)
3.4.13 Assignment Operation for Sets

As mentioned in Subsection 3.4.11, assignment operations can be more complex, in this section the syntax of assigning a set to the value of another set is discussed.

Set1 = Set2

This is a complex operation that will be broken down at the preparation phase. The resulting operations will be as follows:

1. If Set1 already exists, drop table, and continue. Add record in the DataFlow table (Pos#, Pos#+1, DB,......). Add a record in the DB_Operations table (Pos#, Drop_Table, ,......,Set1 ) Where Set1 is the Table_Name field.

2. Create a new table, give it the name Set1, and with fields equal to Set2

3. Add record in the DataFlow table (Pos#, Suc#, DB,......). If step 1 was executed, then Pos#+1 should replace Pos#. also in the next step. If this statement is the end of a schema definition (operation) then set Suc# to 1, i.e. main menu control or the Suc# from the calling subroutine.
4. Add a record in the DB_Operations table (Pos#(+1), Create_Table, „……,Set1,Set2,) Where Set1 and Set2 are the Table_Name and As_Table_name fields.

3.4.14 Assignment Operation for a Set

The formal specification syntax allows for resetting set values, as follows: Set1 = Φ

In this case, the assignment operation on the set deletes all the elements in the set, i.e. all the table records and changes the state of the system as follows:

1. Add record in the DataFlow table (Pos#, Suc#, DB,,,,,,). If this statement is the end of a schema definition (operation) then set Suc# to 1, i.e. main menu control or the Suc# from the calling subroutine.
2. Add a record in the DB_Operations table (Pos#, Delete,…)
3. Add a record in the DB_Operation_Tables table (Pos#, Set1,)
4. Add a record in the DB_Operation_Details table (Pos#, 1, …..,T, …) Where the Whole_Table field is set to T for True

An example is shown in line 280 in the InitBirthdayBook schema in Figure 3.5 below.

![Figure 3.5: InitBirthdayBook Schema](image)

3.4.15 Inserting Records in the Database

Elements can be added to sets, in other words, records can be added to tables, the syntax in Z is as follows:
In order to represent this complex operation in SpecDB, the following algorithm should be executed:

1. Add a record in the DataFlow table (Pos#+2, Suc#, DB, , , , ,). If this statement is the end of a schema definition (operation) then set Suc# to 1, i.e. main control or the Suc# from the calling subroutine.
2. Add a record in the DB_Operations table (Pos#+2, Insert, , , , , ).
3. Add a record in the DB_Operation_Tables Table (Pos#+2, Set3, ..)
4. Set Current_Var_ID = Var ID for Var1 in its specific subroutine if any.
5. Add a record in the DB_Operation_Details Table (Pos#+2, 1, ,, , , Current_Var_ID)
6. Set Current_Var_ID = Var ID for Var2 for its specific subroutine if any.
7. Add a record in the DB_Operation_Details Table (Pos#+2, 2, ,, , , Current_Var_ID)

(Current_Var_ID is inserted in the From_Var_ID field for steps 5 & 7)

For example, in line 110 in Figure 3.1.

\[
\text{birthday}^* = \text{birthday} \cup \{ \text{name?} \rightarrow \text{date?} \}
\]

This is a complex operation that is broken down in the preparation phase. The resulting operations will be as follows:

Line 110: \( \text{birthday}_* = \text{birthday} \) (creating a new table)

Line 112: \( \text{birthday}_* = \text{birthday}_* \cup \text{name?} \rightarrow \text{date?} \)

For line 112, add record in the Data flow table:

\[
(112,1,\text{DB}.). \quad \text{Add record to DB_Operations (112, Insert,.). Add record to DB_Operation_Tables (112, Birthday_, .). Set Current_Var_ID := Var_ID of name?,}
\]
e.g. 3, add a record to DB_Operation_Details (112,1, ...,3,..) and set Current_Var_ID := Var_ID of date?, e.g. 4. Add record to DB_Operation_Details (112,2, ...,4,..).

3.4.16 Output Table Definition

A system may give various forms of outputs, sets can be one of those output types. In Z, Output sets are defined as follows:

Var1!: P Type1

Var1 is an output table or set in this case, this is known from its type. In order to represent this definition in SpecDB, the following algorithm should be executed:

1. If not already inserted, insert Type1 in types: type_name (all the rest of the attributes will be empty)

2. If table Var1 exists, drop it, add record in dataflow table (Pos#,Suc#,DB,...) and add record in DB_Operations table (Pos#, Drop_Table,.....,Var1,..) where Var1 is the table name. if not, add Var1 to the List_of_Tables table.

3. Create a new table, give it the name Var1, with one attribute (key) of type Type1 and name this attribute: Var1.

4. Add a record in the Table_Description table (Var1,Var1,Type1)

5. Insert a record in the Var table (#,var1,DB_Table,Current_SR , ,T,,...). Current_SR is inserted in that record if the variable is defined in the context of an operation definition (Schema). The true (T) is the value of the Is_Output field.
If Var1 is the only output for a subroutine, then it cannot be considered as a function, since tables cannot be passed as parameters. The output table will be available for access outside the scope of the current subroutine, and when this subroutine or any other utilizing the same table is invoked, this table will be dropped, and recreated, and the data redefined.

For example, in line 240 in Figure 3.6.

```
Remind
BirthdayBook
\today? : DATE
cards! : \text{NAME}
cards! = \{ n : known \mid birthday(n) = \today? \}
```

Figure 3.6: Remind Schema

In this case the algorithm will add cards to the list of tables. Create table Cards, with one attribute (key), named Cards of type NAME. Add record in the Table_Description table (Cards, Cards, NAME). Add record in the Var table (8,Cards, Remind, , T,...) Where the Is_Output attribute is set to T.

### 3.4.17 Assigning a Variable to a Calculation

The Z syntax of a calculation assigned to a variable is given below.

```
Var1 = ((Func2 \land Func3) \lor Func4)
```

Any other right hand side calculation can be represented using the same method. The operators can be of any type that match the results of the operands. In order to represent this complex calculation and assignment operation in SpecDB the following algorithm should be executed:
1. Add record in the DataFlow table (Pos#, Suc#, Assign,,,). If this statement is the end of a schema definition (operation) then set Suc# to 1, i.e. main menu control or the Suc# from the calling subroutine.

Set Current_Var_ID = Var ID for Var1 in its subroutine context, if this statement is defined in a subroutine.

Insert record in the Assignment table (Pos#, Current_Var_ID, Calc, ,Pos#,....), where Cal_Num = Pos#;
Reset Component_Num = 0
Increment Component_Num.

Follow the rule for inserting components in the calculation or predicate table: Insert record in the Calculation table (Pos#, Component_Num, func,…,func2, , func,...func3,,AND) If the Calculation is a complex one like the one shown, then repeat steps 5 and 6 until all the components of the calculation are specified.

```
390       RAddBirthday
400       ∆BirthdayBook
410       name? : NAME
420       date? : DATE
430       result! : REPORT
440       result! = (AddBirthday ∧ Success ) ∨ AlreadyKnown
```

Line 440 in Figure 3.7 is an example of assigning the variable result to either the value “Success” or “AlreadyKnown” depending on whether the input name and date already exist in the Birthday Book or not.
3.4.18 Constraining the Values Populating a Table or Set

The syntax of the formal specification language Z allows for specifying how a set or a table can be populated. In the below example, a table is populated from specific results of a query on another table:

$$\text{Var1!} = \{n: \text{Set1} \mid \text{func1}(n) = \text{var2}\}$$

Var1 is an output table set, func1 is also a table or a database operation on a table.

The first part of this statement: “Var1!={” populates a table (set). Assigning a variable to a set means that this variable is really a set of values, which may include 0 or more values.

The second part, “{n:Set1 |” introduces a new variable, of type Set1, also it indicates that n is in itself a set (table) that is a subset of Set1. The third part of the statement “|” indicates that there is a restriction on n, it does not include all the values of set1, just those that satisfy the condition on the right hand side. The final part of this statement is the restriction “func1(n)=var2”.

These four steps will be defined in SpecDB as in this translation of this SQL statement:

Insert into var1

(Select Field_Name1 from Set1
Where Field_Name1 in
(Select Field_Name2 from func1
Where Field_Name3 = Var2))
The statement will be specified as above, using Pos# and Pos#+1.

The field names and the quantities will be known by inquiring the already specified design.

1. Insert into DataFlow table (Pos#, Suc#, DB,..) If this statement is the end of a schema definition (operation) then set Suc# to 1, i.e. main menu control or the Suc# from the calling subroutine. In this case Suc# should not = Pos#+1, since this value will be used to define this statement.

2. Insert record in DB_Operations table (Pos#, Insert,, Pos#,…,Var1,,T) where the value of DB_Condition_Num = Pos#, Table_Name = Var1 and AS_Select = T for true.

3. Set Field_Name = Field_Name from the Table_Description table where Table_Name = Set1

4. Insert record in DB_Operation_Details (Pos#,1,,Field_Name,…..)

5. Insert record in DB_Operation_Tables (Pos#, Set1,)

6. Reset Component_Num = 1 for the predicate table, with each insert in the Predicate table with the same Pos#, increment the Component #.

7. Insert record in Predicate (Pos#, Component_Num, DB_Field, …, Field_Name, …, Pos#+1,…) where Left_DB_Field = Field_Name and In_DB_Oper_Num = Pos#+1.

8. Insert record in DB_Operations table (Pos#+1, Select, ,Pos#+1,…) 

9. Set Field_Name = SR_Var_name from the SubRoutine_IP_Description table where Subroutine_Name = func1 and type = that of Set1 

10. Insert record in DB_Operation_Details (Pos#+1,1,,Field_Name,…..)
11. Insert record in DB_Operation_Tables (Pos#+1, func1,)

12. Set Field_Name = OP_Var_name from the Function_Declaration table where Func_Name = func1.

13. Set Current_Var_ID to Var_ID of var2.

14. Reset Component_Num = 1 for the predicate table, with each insert in the Predicate table with the same Pos#, increment the Component #.

15. Insert record in Predicate (Pos#, Component_Num, DB_Field, ..., Field_Name, VAR, , Current_Var_ID, ..., Equal) where Left_DB_Field = Field_Name and Right_Component_Type = Var and Right_Var_ID = Current_Var_ID and Operator_ID = Equal.

Figure 3.6, line 250 is an example of such an operation.

The steps for this complex operation will be defined in SpecDB as in the following translation of the statement at Line 250 cards!={n: known | birthday(n) = today?}:

Insert into cards

(Select known from KNOWN

Where known in

(Select Name1 from Birthday

Where Date1 = today))

The statement should be specified as above, using Pos# 250 and Pos# 251.
3.5 Conclusion

In Chapter 3, some examples were given to show how software specifications written in a formal specification language like Z, can be represented in SpecDB. The formal specification language Z was chosen because it is one of the more common formal specification languages. Representing formal specifications in SpecDB aids the process of automated testing and automated code generation. Through the use of SpecDB, automated tools can become more intelligent in generating and testing programs, based on an understanding of what the software under construction is supposed to do, by design. The scalable database design of SpecDB allows for any additions to accommodate more programming constructs, new forms of input and output, etc., as they evolve in the future, as well as formal specification syntax. In this chapter, a method to translate some examples of formal specifications written in Z into the unified SpecDB representation was illustrated. Automated tools can easily be developed based on the algorithm in this chapter to carry out the translation from well formulated and complete specifications written in Z, or other formal methods, into the SpecDB representation. Once the specifications have been stored in SpecDB, automated tools can reason about the specifications and thus can automatically and intelligently create test cases to test the software.

In Chapter 4 and Chapter 5 two examples of such intelligent testing applications are discussed, namely an automated constraint generator to enforce business rules at the database level of applications and a reverse engineering automated testing tool.
CHAPTER 4

AUTOMATED CONSTRAINTS GENERATOR:

AN APPLICATION OF SPECDB

In this chapter, one of the tools that uses SpecDB as a foundation, is introduced; namely, an automated database constraints generator that creates constraints from represented business rules in SpecDB. The main advantage of database level constraints creation is the enforcement of some of the business rules at the database level, and not only at the application level. If business rules are enforced only within the code of the application running on a database, with no database level constraints on the tables, the possibility exists for inconsistent, unreliable data. Errors in the data may result from not enforcing and coding the business rules every time a change is made to the state of the database, or when the data is altered at the database level, and not the application level.

The algorithm used to automatically generate database-level constraints is explained. The tables from SpecDB used in this process are also highlighted, together with an explanation of how the individual entities collectively describe business rules represented by the specifications expressed in SpecDB.
The use of SpecDB illustrated in this chapter focuses only on the specifications required to generate database-level constraints for the purpose of testing the validity of the data in a database. In some production databases, database administrators prefer to drop constraints in order to offer more speed during system use. This strategy may result in faster system response, but can threaten data consistency, completeness, correctness and integrity. However, if the technique explained in this chapter is used to generate constraints, data errors will be expelled, giving a chance to clean the production data. The tool described in this chapter can also be used in the data cleaning phase of building a data warehouse. The algorithm in this chapter automatically generates a script for all the database constraints representing data-level business rules. When the script is run, all the constraints are created in the database. Once the constraints are created and the database integrity is checked, all the data that does not satisfy the constraints will be identified.

The algorithm also creates another script for dropping all the constraints, if the system administrators would like to drop them again. As business rules are added to or changed in SpecDB, the algorithm in this chapter will generate the new or modified constraints. A new table is introduced for SpecDB, in addition to the original design. also an extension to one of the tables in SpecDB is made. The addition and extension of these tables was necessary to store some business rules to aid the process of automatic constraint generation. The new table is Constraints. Table_Description was extended to include some required business rules for the fields. From SpecDB, Type_Restrictions, Type_LOV, Predicate and DB_Operation_Tables are also used.
The extensions of the above mentioned tables highlight the scalability of the SpecDB implementation. Other tools can be created, like the one presented in this chapter to test other sections of the software applications. If an extension of SpecDB is required to represent the specifications necessary for the completion of the new automated tool's function, a similar technique as the one used in this chapter can be adopted to extend the SpecDB representation.

4.1 The Design of the Additional Tables in SpecDB

In this section, each of the new entities of the SpecDB design is defined and explained in more detail.

SpecDB is a database. The entities of the SpecDB database store the specifications of a software application. SpecDB is designed to efficiently and accurately represent the specifications necessary for unit testing database applications. In the representation described in Chapter 2, it was stated that the same concept used to create SpecDB can be used to add more tables to represent other aspects of a software application, as needed. In this chapter, a new table is added to SpecDB, in addition to the original design. Also Table_Description was extended to become Table_Fields_Description. The addition and extension of these tables was necessary to store some business rules. The storage of business rules in SpecDB provided the basis for automatic constraint generation for the purpose of data validity testing. The new table added is Constraints. From SpecDB the List_Of_Tables, Type_Restrictions, Type_LOV, Predicate and DB_Operation_Tables are also used.
Throughout the subsections of Section 4.1 are figures showing parts of the relational database schema of the design of SpecDB. The relations between the entities are shown. Also the types of the fields in each entity are shown, numeric types are identified by the symbol “789”, text or character types are identified by “A”, and date or time types are identified by a calendar card. Table keys are identified by a “#” symbol, and whether each field is optional or mandatory is identified by “O” or “ * ” respectively. There are some fields that are constrained to a list of valid values, for example Is Unique is a Boolean value that can be only T for true or F for false. On the left of the field name, if the value of the field is constrained to a list of valid values, a relation symbol is viewed. All the tables specifications and field types are written in an SQL script in Appendix A.

### 4.1.1 Using List_of_Tables and Extending Table_Description

The List_of_Tables entity stores all the software/user-defined sets and tables. The Table_Description entity stores the names of the fields of the user-defined tables and sets, and their respective types.

![Figure 4.1: SpecDB Original Table_Description](image)

The SpecDB Table_Description entity is extended to become Table_Fields_Description as shown in Figure 4.2. This table stores the fields of the user defined tables and the features of each field. The Field_ID is the Table_Name concatenated with the
Field_Name, separated by an underscore symbol. Accordingly, Field_ID is a unique value in this table for every record (since a table cannot have two fields with the same name). The Field_ID is used to identify a specific field in the rest of the tables joined to Table.Fields_Description. Is_Primary_Key is a Boolean value indicating whether the field is a primary key to its entity. If Is_Primary_Key is true, then the algorithm in Section 4.2 will create a Primary Key constraint for this field on its table. Is.Unique is also a Boolean value indicating whether the field is unique for each record. If Is.Unique is true, then the algorithm in Section 4.2 will create a UNIQUE constraint for this field on its table. Optional_Mandatory can be either O for optional or M for mandatory; indicating whether the field can be left blank or not for each record. If the value of Optional_Mandatory is “M”, then the algorithm in Section 4.2 will create a NOT NULL constraint for this field on its table. Has_LOV_Restrictions is also a Boolean value used to identify whether there is a list of valid values that the values of this field have to take.

![Figure 4.2: Table Field Description Table](image-url)
Any other value not in the valid list of values is considered invalid. If the value of Has_LOV_Restrictions is true, the algorithm in Section 4.2 creates a table for the valid values referencing the Types_LOV table. It also creates a REFERENCES constraint on this field. Has_Table_Restrictions is similar to Has_LOV_Restrictions. If the value of this field is true then Which_Table will store the table that this field references, and Which_Field will store the specific field referenced in the specified table. The algorithm in Section 4.2 will create a REFERENCES constraint if the value of Has_Table_Restrictions is true. Finally, Has_Range_Restrictions indicates whether there are other restrictions on the range of valid values for the specified field. For example, the value of “Age” for employees should be any value between 18 and 74. A list of values can be created, however, for some situations as the latter, it is best to specify a valid range of values. In this case, the algorithm in Section 4.2 creates a check constraint. For the latter example, the range is specified in the Types table, and a check constraint is created to enforce that range of valid values.

Other restrictions like specifying a range excluding one or more values are stored in the Type_Excluded_Ranges table. Also the Constraints table stores some other more complex restrictions as shown in Section 4.1.2. If there are other types of constraints, the Has_Constraint_Restrictions attribute is set to true. Multiple constraints can exist for the same field, accordingly, the number of the other constraints is specified in the Num_of_Constraint_Restrictions attribute.
4.1.2 Adding the Constraints Table to SpecDB and Using the Predicate Table

The constraints table was added to specify certain business rules that are not represented in the Table_Fields_Description table and dynamic business rules. The constraints specified in Section 4.1.1 in table Table_Fields_Description are static, meaning that the requirement that a SSN is unique is a fact that needs to be enforced on the tables, regardless of the data in the tables. However, there are some business rules that are dynamic. For example, the salary of an employee should not exceed that of his/her manager. The salary of the manager is not pre-set, i.e. not static. Also different employees have different managers. In order to enforce such business rules on the database level, the Constraints table is added to SpecDB and use other tables like DB_Operation_Tables and Predicate to store those business rules. The Constraints table is defined in Figure 4.3. The tables used by the constraint are stored in DB_Operation_Tables. The conditions and rules are stored in the Predicate table. On_Field is a Boolean value indicating whether the constraint is on a specific field. If On_Field is True, then there is a constraint for the value of the specified field. For example the employee age has to be greater than 18. In this case the Field_Name, DB_Name and Table_Name will store the name of the field (age in this example), name of the database where the table is, and the name of the table containing the specified field, respectively. Field_Name and Table_Name are required to link between the Table_Field_Description table and the Constraint table. A snapshot of the data in the SpecDB tables to represent constraint 1 is shown in Tables 4.1, 4.2 and 4.3.

Constraint 1: employee age \( \geq 18 \)
Table 4.1: Snapshot of SpecDB Table Field Description for Constraint 1

<table>
<thead>
<tr>
<th>Table Name</th>
<th>Field Name</th>
<th>Is_unique</th>
<th>Has_range_restrictions</th>
<th>Has_constraint_restrictions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Employee</td>
<td>Age</td>
<td>F</td>
<td>F</td>
<td>T</td>
</tr>
</tbody>
</table>

Table 4.2: Snapshot of SpecDB Constraint Table for Constraint 1

<table>
<thead>
<tr>
<th>Constraint Name</th>
<th>Constraint Num</th>
<th>On_Field</th>
<th>Field Name</th>
<th>Table Name</th>
<th>Constraint Clause</th>
<th>Pred Num</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAgeGT18</td>
<td>27</td>
<td>T</td>
<td>Age</td>
<td>Employee</td>
<td></td>
<td>79</td>
</tr>
</tbody>
</table>

Table 4.3: Snapshot of SpecDB Predicate Table for Constraint 1

<table>
<thead>
<tr>
<th>Pos_num</th>
<th>Component _Num</th>
<th>Left_operand_type</th>
<th>Lt_table_Name</th>
<th>Lt_DB_Field</th>
<th>Right_operand_type</th>
<th>Right_num(Const)</th>
<th>Operator_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>79</td>
<td>1</td>
<td>DB_field</td>
<td>employee</td>
<td>Age</td>
<td>Const</td>
<td>18</td>
<td>&gt;=</td>
</tr>
</tbody>
</table>

If On_Field is false, this means that the constraint is not defined on only one field, there is a number of tables involved, or a number of fields involved in the constraint, i.e. a complex business rule or constraint. In this case the relation between the values of some fields in specified tables has to be governed by the constraint.

Figure 4.3: Constraint Table

If for example, the salary of an employee should not exceed that of his/her manager, the employee entity is involved twice in this constraint, once as a manager and once as an
employee. There are also other fields involved, the salary, manager's ID, and the employee ID. The employee table will be used twice in this case, once to reference the employee and once to reference the manager. For each case, the alias is different, for example, alias 'e' is used for employee and alias 'm' is used for manager. The rule itself is placed in the Predicate table. To illustrate how this constraint is represented in SpecDB, snapshots of the tables of SpecDB needed to represent this constraint number are shown:

Constraint 2: employee's salary ≤ employee's manager's salary.

Table 4.4: Snapshot of SpecDB Table Field_Description for Constraint 2

<table>
<thead>
<tr>
<th>Table Name</th>
<th>Field Name</th>
<th>..</th>
<th>Is Unique</th>
<th>Has_range_restrictions</th>
<th>Has_constraint_restrictions</th>
<th>..</th>
</tr>
</thead>
<tbody>
<tr>
<td>..</td>
<td>..</td>
<td>..</td>
<td>..</td>
<td>..</td>
<td>..</td>
<td>..</td>
</tr>
<tr>
<td>Employee</td>
<td>Salary</td>
<td>..</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>..</td>
</tr>
</tbody>
</table>

Table 4.5: Snapshot of SpecDB Constraint Table for Constraint 2

<table>
<thead>
<tr>
<th>Constraint Name</th>
<th>Constraint Num</th>
<th>On Field</th>
<th>Field Name</th>
<th>Table Name</th>
<th>Constraint Clause_Pred_Num</th>
<th>Condition_Pred_Pos_Num</th>
<th>..</th>
</tr>
</thead>
<tbody>
<tr>
<td>..</td>
<td>..</td>
<td>..</td>
<td>..</td>
<td>..</td>
<td></td>
<td></td>
<td>..</td>
</tr>
<tr>
<td>ESalLTMSal</td>
<td>28</td>
<td>F</td>
<td>salary</td>
<td>Employee</td>
<td>80</td>
<td>81</td>
<td>..</td>
</tr>
</tbody>
</table>

Table 4.6: Snapshot of SpecDB Predicate Table for Constraint 2

<table>
<thead>
<tr>
<th>pos_num</th>
<th>Component_num</th>
<th>left_operand_type</th>
<th>.</th>
<th>Left_table_name</th>
<th>Left_table_alias</th>
<th>Left_DB_field</th>
<th>Rt_op_type</th>
<th>Rt_table_name</th>
<th>Rt_table_alias</th>
<th>Rt_db_field</th>
<th>Operator_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>..</td>
<td>..</td>
<td>..</td>
<td>.</td>
<td>..</td>
<td>..</td>
<td>..</td>
<td>..</td>
<td>..</td>
<td>..</td>
<td>..</td>
<td>..</td>
</tr>
<tr>
<td>80</td>
<td>1</td>
<td>DB_field</td>
<td>.</td>
<td>Employee</td>
<td>E</td>
<td>salary</td>
<td>DB_field</td>
<td>employee</td>
<td>M</td>
<td>salary</td>
<td>&lt;=</td>
</tr>
<tr>
<td>81</td>
<td>1</td>
<td>DB_field</td>
<td>.</td>
<td>Employee</td>
<td>E</td>
<td>manage</td>
<td>DB_field</td>
<td>employee</td>
<td>M</td>
<td>emp_ID</td>
<td>=</td>
</tr>
</tbody>
</table>

The constraint in Table 4.5 states that there exists a restriction on the employee salary. The constraint itself (employee salary < manager salary) is placed in the Predicate table at Pos_num 80. However, this is not the complete constraint. The relation between the
two employees is also stated in the Predicate table at Pos_Num 81. Using the Predicate table in combination with the Constraint table allows for a wide variety of constraints and business rules that can be represented in SpecDB.

Among the complex business rules for example, the age of an employee should not exceed 125% of his manager, or the increase in an employee's salary should not exceed 125% of his previous salary. The latter example is a comparison between the value of a field before and after a modification. To identify old and new values, :old.field_name and :new.field_name are used respectively in the Left or Right_DB_Field field in the Predicate table. This complex business rule requires a constraint clause in the Predicate table with two components. The first component is the :old.salary * 1.25, in this case the left operand is a database field, the right operand is a constant and the operator is the
multiplication operator. The second component is constraining :new.salary <= component 1, i.e. :new.salary <= :old.salary*1.25.

4.1.3 Other Tables Used from SpecDB, Type_Excluded_Ranges and Types_LOV

SpecDB allows for user-defined restrictions on the fields of the user-defined entities. Restrictions can be one of the following:

4.1.3.1 Representing List of Values (LOV) Restrictions

A List Of valid Values, i.e. any variables of the specified type can only take on one of the values in the LOV. In this case, the valid values for a specified field are placed in the Types_LOV entity, each in a separate record.

4.1.3.2 Representing Other Field Restrictions

If a type is restricted to a set of valid values that are better specified by ranges, the ranges are specified in the Types table. If there are some values to exclude from the range of valid values, or even a range to exclude, the rules are stored in the Type_Excluded_Ranges entity to define the allowable range. For example, for a Month_Number type, valid values are between the range of 1 to 12. Hence the value of the “Has_Range” field in the Types table is set to True, Range_From_Int is set to 1 and Range_To_Int is set to 12. A range can also be specified, with exclusions, e.g. integers from -150 to 150 excluding 0. In this case the range -150 to 150 is specified in the Types table, and the exclusion of the value 0 is specified in the int_value in a record in the Types_Excluded_Ranges table.
4.2 Automatically Generating Database-Level Constraints: The Algorithm

After describing the tables used by the automatic constraint generator tool in Section 4.1, in this section the algorithm required to generate the different types of constraints, outlined in the previous section, is described. There are two main modules for the automated constraint generator. The first is a straightforward constraint generator with little computation intelligence, that generates static constraints on the values of the fields. This first module creates database level constraints. The second is a more complex constraint generator that handles complex business rules that cannot be expressed by simple database constraint operations, but need more complex stored procedures for triggers to enforce the business rules on the database level. Accordingly, the second module automatically generates stored procedure code for triggers, as opposed to simple database constraints as in the first module.
In Section 4.2.1 simple constraints are generated. In Section 4.2.2, the algorithm for the stored procedure code generator is described.

### 4.2.1 Generating Simple Static Constraints – The Algorithm

In this section, the simple algorithm for the automatic constraint creator is described. The actual code is provided in Appendix C. The constraints that are automatically created by the algorithm in this section handle primary key constraints, uniqueness, not-null (mandatory) constraints, simple check constraints for static field values, list of values and relation constraints.

1. Open a file to create the SQL script to create the constraints, and another to drop all the constraints.

2. For every constraint created and appended to the SQL constraint script file, append a drop constraint command to the drop constraints script, so as to create two SQL script files, one for creating all the constraints, and one for dropping them.

3. For every record in the Table_Field_Description table do
   
   a. if the field is a primary key (Primary_Key = T)
      
      then append to the script file the Alter Table command to add a primary key constraint on the current table (Table_Name) for the current field (Field_Name).

   b. if the field value is unique (Is_Unique = T)
      
      then append to the script file the Alter Table command to add a unique constraint on the current table for the current field.

   c. if the field is mandatory (Mandatory = T)
then append to the script file the Alter Table command to add a Not Null constraint on the current table for the current field.

d. if the field should have a value from a list of valid values
then append to the script file the following commands:

- Create table command, give it the name of the table appended to the field, for example Employee_MonthHired.
- Insert into the newly created table all the values from Types_LOV where the Type_Name is the current field ID.
- Alter the current table, add a foreign key constraint referencing the newly created table.

e. if the field has table restrictions, i.e. it references another field in the same or another table, this is a referential integrity constraint that is enforced by the RDBMS once the constraint is created. Accordingly, there is no need to create triggers to check whether the constraint is satisfied upon the deletion of a record in the referenced table, since this is enforced by the RDBMS. However, we need to specify the referential integrity constraint as done below:
then append to the script file, a foreign key constraint for the current field and table, referencing the specified table and field in the which_table and which_field attributes.
f. if the field has a range of valid values
then query the Types table for the range boundaries (range_from, range_to values).

• Append a Check constraint to the script file to check that the value of the field is between the two boundary values obtained.

• Since there can be one or more exclusions from the range of valid values, query the Type_Restrictions table for exclusions on the valid range specified.

• For each exclusion, determine whether it is a single value excluded from the range, or a range to exclude.

• Append to the script another Check constraint to check that the value of the field is not between the specified range in the exclusion, or is not the value of the specific value to exclude.

g. if the field has constraint restrictions
then query the Constraint table for the current table and field.

if a constraint exists in the Constraint table where the On_Field attribute is set to true
then using the Constraint_Clause_Pre_Num attribute for that constraint, query the Predicate table for the record(s) where Pos_Num = Constraint_Clause_Pre_Num.

If there is only one record (i.e. on component) in the Predicate table then append to the script file a check constraint on the current field
and table to satisfy the predicate, i.e. left operand followed by the operator ID, followed by the right operand.

Else if there are multiple components,

- Use a temporary empty text variable or database text fields to build the constraints from the predicate components
- Starting with component 1, append to the text variable, or DB field the left operand, operator ID, right operand
- repeat with the following components until all components are built
- Append all components to one another with the correct operators into one predicate.
- Append to the script file the Check constraint for the current field and table with the built predicate.

4. Commit the changes to the open files and close the two script files.

Case 3.g is for representing a constraint similar to constraint 1 discussed in Section 4.1.2.

In this case, a constraint can have a simple one component predicate e.g. Age >=18, or it can have a complex multiple component predicate, e.g. Annual Salary + Annual Bonus <=125% Annual Salary. For the latter example, the constraint can be on the Annual Bonus field. Three components in the Predicate table represent this constraint

1. Annual Salary * 1.25 .......Component 1
2. Annual Salary + Annual Bonus.......Component 2
3. Component 2 <= Component 1
The algorithm specifies the use of a temporary database table containing one text field or a text variable to form the complete constraint predicate statement before appending the constraint to the script file.

4.2.2 Generating Complex Dynamic Business Rules Enforcement Code

More complex business rules involving dynamic checking of the data cannot be enforced using simple database constraints as those described in Section 4.2.1. Complex operations similar to constraint 2 discussed in Section 4.1.2 can only be created using database assertions or stored procedure code for a database level trigger. Accordingly, in this section we look at how the automated constraint generator automatically generates procedure code for triggers to enforce those complex dynamic business rules.

The algorithm is better understood following an example. Constraint 2 is used for this purpose in this section:

Constraint 2: employee's salary ≤ employee's manager's salary.

There are two main criteria for specifying such a constraint. First, the constraint that needs to be satisfied, in this example the constraint that needs to be satisfied is:

\[ e.\text{salary} \leq m.\text{salary} \]

The second criterion is the relationship(s) between the entities that are involved in this constraint. For Constraint 2, the relationship is the specification: 'Employee's Manager', translated into PL/SQL code renders a where clause: \( e.\text{manager\_id} = m.\text{employee\_id} \).
The first criterion, the constraint that needs to be satisfied, can be a simple one component constraint, or it can be more complex, and more than one component is needed in the Predicate table to represent it. The same applies for the second criterion, the relationship between the entities. Again there can be none, one or more components in the predicate to specify the relationship between the entities involved in the constraint. The data needed for the automatic creation of procedure code to enforce complex dynamic constraints is represented in the design of the SpecDB tables discussed in Section 4.1. From Tables 4.4, 4.5 and 4.6, the algorithm in Section 4.2.2.1 automatically generates the procedure code for a trigger to enforce Constraint 2.

The full algorithm to include all cases is decision intensive, one of the cases are given here to illustrate the idea. However, there are other paths in the algorithm where the operands are not database fields, but constants for example. Handling components in the predicate table with constant operands is simpler than handling database field operands, as shown in Section 4.2.2.1. An example of handling constants was shown in Section 4.2.1. There is also paths through the algorithm where the key to the table being modified is a composite key, meaning that there are multiple fields that collectively form the database table key. Before the below algorithm is executed, the database table key is identified from the Table_Field_Description table. In the algorithm shown in Section 4.2.2.1, a simple database table key is handled. The same technique can be used to handle composite keys. Finally, there are cases in the algorithm involving multiple components for a predicate. The technique used in the algorithm in Subsection 4.2.1 to handle complex multiple component predicates is also used here for the same purpose.
4.2.2.1 Complex Business Rules Generation Algorithm

The algorithm described in this section is a part of the whole decision intensive algorithm to automatically create procedure code for database level triggers, as discussed in the previous section. A file is created for the procedure code of the trigger, every statement in the algorithm is appended to the file. Also the value of the fields involved in the update operation that triggers the execution of the procedure code created by the algorithm below, are referred to as 'New' values, in the 'Current' record.

1. Open a file to create the procedure code to create the constraint, and another to drop all the triggers created.
2. Determine the field name that is being updated from the Constraint table.
3. Create a Trigger to be executed before update on Constraint.Table_Name, Constraint.Field_Name.
4. Determine the type of the field stored in right_DB_field in the Predicate table.
5. Create a variable of the type identified in step 4, append the alias to the right_DB_field and use the result as the name of the variable (e.g. m_salary).
6. Determine the key to the table being modified.
7. Query the Constraint table to identify the pos_num for the predicate representing the constraint clause (a), and that representing the condition (b).
8. Append a select statement as follows:

- Select right_table_alias.right_db_field into variable (created in step 5) (the data used here is from (a) in step 7).
- From Right_table_name Right_table_alias , left_table_name, left_table_alias (the data used here is from (a) in step 7).
- Where left_table_alias.Key = CurrentKey and
  left_table_alias.left_db_field operator_id right_table_alias.right_db_field
  (the data used here is from (b) in step 7).

9. From step 2, the field being updated has been determined, now determine the new value of that field (Current_Field & New_Field_Value).

10. Append to the file the decision statement:  If Not (New_Field_Value operator_id right_table_alias.right_db_field) (the data used here is from (a) in step 7) then raise and handle exception.

From Constraint 2 the following Select statement is created from the above algorithm:

```
Select m.salary into m_salary
from employee m, employee e
where e.employee_id = Current_Emp_ID and
  m.employee_id = e.manager_id;
if not(Current_Salary <= m_salary) then raise_exception;
```
4.3 Conclusion

In this chapter, an automated tool to generate business rules is described. It is also shown how to enforce them on the database level by automatically creating constraints and assertions using the data stored in SpecDB. The design of a new table, Constraints is explained, and an extension on the design of the Table_Description table of SpecDB, renamed Table_Fields_Description is shown. Those modifications were made in order to allow for the representation of business rules. The automated tool introduced in this chapter creates a script to create all the constraints and another one for dropping them. System administrators can choose to run production systems with fewer constraints or no constraints at all in order to enhance the speed of production systems. They can run the tool described in this chapter to create and automatically test the validity of the data during certain times of the week to maintain the accuracy, completeness and integrity of the system. Through the use of SpecDB, automated tools can become more intelligent in generating and testing programs, based on an understanding of what the software under construction is supposed to do, by design. The design of SpecDB allows for future enhancement and additions to accommodate more programming constructs or requirements, as they evolve in the future.
A lot of testers and available automated testing tools decide that a software application has passed a specific test case, if the expected outputs match the actual outputs, based on black box testing. This is, however, not an absolute proof of the quality of the software for that test scenario. The fact that the software application can commit implicit errors that are not directly reflected in the output values still exists. For example, in a database application, if a social security number of an employee is given as an input, and the system outputs the correct required information about that employee, as specified for an operation; this is not enough for a tester or a testing tool to decide that the software passed for this operation. There is a possibility that some other records in the database have been mistakenly deleted, or modified, during or after the output is displayed. In another example from a system security perspective, an e-commerce tool takes credit card information, and customer confidential data, carries out the transaction, sends a sales order to the correct department, and accomplishes all the tasks to complete the sale transaction. The system could also send the credit card information to an external non-secure site. Such a security leak cannot be identified by black box testing techniques.
The motivation to think of a testing tool solution that would spot such implicit errors of commission was aroused from the above discussion where the software application carries out operations that are not part of the specifications. White box testing techniques, where the actual implementation code is examined is a good basis for such a solution. If a testing tool can trace an implemented program and generate the specifications of the actual implementation, and present it in a non-technical form that can be understood by testers and end users, it will then be very easy to spot such errors of commission. In this chapter the operation of this tool is explained. It is called a reverse engineering testing tool, or RE tool, since it generates the specification of the implemented software from its implementation. Such specifications can then be fed into SpecDB as an instance of a software specifications and can be easily automatically compared to another set of specifications, the required specifications of that software, i.e. compare actual to specified behavior. The determination whether the implemented software matches the required specifications can then be easily made, with the guarantee that no errors of commission have been unnoticed.

In this chapter, a reverse engineering approach is used to log the actual execution of a program, due to some input, using white box testing techniques. Once the program actions have been logged, they are presented in a way that non-technical testers and other members of the software development team can understand. The actual outcome can then be easily matched and compared to the expected output based on the program specifications and oracles that determine what the expected program output should be, based on the specified inputs. The decision-to-decision path (also known as DD-paths) white box testing technique, is modified and used to implement this tool, but for a different purpose; namely to determine
the actual program behavior given specific input, and not as a coverage matrix to determine which parts of the software are covered by some test cases, or as a guide to generate test cases to cover all DD-paths in a program. Instead, the technique for DD-paths testing is used to determine actual program behavior.

5.1 Technique

The main idea of the reverse engineering (RE) testing tool is to follow the program execution and report the inputs it receives, the actual outputs it generates, and the process it follows. These three perspectives, input, actual output and the process are all represented in a simple format for non-programmers to be able to read and interpret how the program executed and whether there are specifications that it did not execute. This outcome will aid in the process of identifying and isolating faults. Faults are identified when the tester (or automated tool, using SpecDB) compares the expected output to the actual output. Faults are isolated by following the process execution behavior provided as an outcome of the approach explained in this chapter and comparing it to the specifications. For the purpose of the RE tool described in this chapter, as an outcome of the tool, for a specific test case on specified inputs, the tool will show how the executed portion of the software got its inputs, and how it generated its outputs and finally it presents all the outputs of the executed software. Those three outcomes of the tool (inputs, computation, and output) are presented in an easily understandable format described in the following sub-sections.

Triggers are programmed units that execute if a pre-specified event occurs. For example, a database trigger can be implemented to check the constraint on a table when a certain field is
modified. GUI triggers execute when a certain event happens in the GUI, for example, a button click, a menu item click, a modification to the value of a field, etc. In order to make the RE tool more effective, it is designed not only to test the specified unit without noting changes that occur due to some triggers that execute as a result of the state changes, or output results of the unit being tested, but also to consider all the modifications that happen to the system, including those of triggers. A domino effect might result from following triggers, since a state change or an output of one trigger may trigger another to execute. The specifications of the RE testing tool include that it should query all triggers, and note the possible domino effect, displaying as part of its own output all the system and state changes as well as all the outputs that result from the execution of the unit being tested.

5.1.1 Input and Pre Conditions Classification and Categorization

The first outcome of the RE testing tool is identifying all the inputs used by the software in its execution of the test case. Inputs can be both explicit and implicit. Explicit inputs are those given to the software using a GUI component or a command line. Implicit inputs are values that the software implicitly acquires during execution, for example when the application queries a database table for the value of a specific field in a specific record. Such values are used in the computations and the outcome of the software is affected by those implicitly acquired inputs too. The format of how the RE tool identifies what kinds of input the program received, are described in this section. Database applications are dynamic by nature depending on the input state of the database. This means that when a test case is executed on a database application with the
same input values several times, the output can be different for each test case, if the data stored in the database changed, and yet, it can be decided that the application passed that test. Database applications are thus unlike a function, which depend solely on explicit parameters. For example, a calculator application is a function, each time you enter $4 \times 7$ to a calculator, it should output the same value, 28. However, each time you query the social security database for the number of living citizens, it might give you a different number each time. It is thus important to testers or automated testing tools on such data-centric applications to know the inputs, both explicit and implicit, to be able to judge whether the software being tested has passed a test case.

Inputs can be presented to a program in many forms. Among the forms are GUI objects, data from a database or files. It is assumed that other forms of input like barcode readers can be considered as data from GUI components, since a barcode reader replaces manual keyboard entry of data in a text field for example. For inputs from the database, a specific database table, field and row identifier are required to determine a single value, as an input. For example the Employee table, has a Salary field. There are as many values for this field as there are records, i.e. employees in this table. An employee identifier is also required to specify a single input salary value. Once the value is extracted from the database, it can be stored in a variable to be used in the computation process. For queries that return multiple values of the same field, i.e. no record identifier is chosen, during run time the results of those queries are placed in what Oracle calls a cursor. Oracle uses cursors to handle computations on input values, one record at a time. The end result is that only one value per field is handled for computation at a time in a
loop, i.e. one record at a time. Accordingly, the above assumption that one field and one row identifier provide a single input is also valid in the situation of using a cursor in a loop for computation involving a query that returns multiple rows. In this case, in each iteration through the loop, the value of the input variable will change because the row identifier changes.

Inputs to a program can also be presented through a GUI component, like a text box, choice from a drop down menu or a value from a combo box containing a list of possible values, etc. If input is taken from a GUI component, several identifiers are needed to specify which GUI component the input is taken from, unless every component has a unique identification number or name. For example, if an employee first name is entered in a text field, the window identifier where that text field is located should be given as part of the component definition, together with an identifier of that specific text field, since there can be many text fields in the same window, a text field for last name, for example. The type of the component should also be specified for computation purposes; in this example the component is a text field.

If input is taken from a file, the file name and location are specified. The type of file is necessary for computation purposes, for example text or binary file, etc. In Table 5.1, the specifications for the different forms of input are summarized, i.e. how each input to the software being tested is identified as an outcome of the RE testing tool. The RE testing tool, determines all the kinds of inputs given or acquired by a software application during execution of a scenario (test case). Each input is given a number, and described
depending on its type, i.e. obtained from a database (DB), GUI or file. Types of GUI components can include “Button”, “Text”, “Radio” for a Radio Button and “Check” for a check box, etc.

<table>
<thead>
<tr>
<th>Category</th>
<th>Classification</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Database(DB)</td>
<td>Table</td>
<td>Field</td>
</tr>
<tr>
<td>GUI</td>
<td>Window</td>
<td>Component</td>
</tr>
<tr>
<td>File</td>
<td>Name/location</td>
<td>Type</td>
</tr>
</tbody>
</table>

In the next subsections examples of the first set of outputs provided by the RE testing tool are given, namely identified inputs of a software execution. The examples will show how the tool communicates the implicit and explicit inputs to a software in a non-technical form, following the categorization in table 5.1.

5.1.1.1 Examples of Inputs from the Database Displayed by the RE Testing Tool

Table 5.1 summarizes how the RE testing tool represents a database input to the program. In table 5.2 are some examples of database inputs. Table 5.2 specifies that the fifth input given to/acquired by the program is from the database. Specifically from the salary field in the employee table, at row #14. The value of Salary at row 14 when read was 56000 and this value was assigned to the internal variable "Salary".
Table 5.2: Examples of Inputs from a Database

<table>
<thead>
<tr>
<th>Input #</th>
<th>Category</th>
<th>Table</th>
<th>Field</th>
<th>Row #</th>
<th>Value</th>
<th>Into variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>“DB”</td>
<td>Employee</td>
<td>Salary</td>
<td>14</td>
<td>56000</td>
<td>Salary</td>
</tr>
<tr>
<td>6</td>
<td>“DB”</td>
<td>Employee</td>
<td>Emp LName</td>
<td>14</td>
<td>Gibson</td>
<td>Last Name</td>
</tr>
</tbody>
</table>

5.1.1.2 Examples of Inputs from the GUI Displayed by the RE Testing Tool

Table 5.1 also summarizes how the RE testing tool represents a GUI input to the program. In table 5.3 show some examples of how the testing tool identifies GUI inputs.

Table 5.3: Examples of GUI Inputs

<table>
<thead>
<tr>
<th>Input #</th>
<th>Category</th>
<th>Window</th>
<th>Component</th>
<th>Type</th>
<th>Value</th>
<th>Into variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>“GUI”</td>
<td>Employee Entry</td>
<td>Save</td>
<td>“Button”</td>
<td>“Click”</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>“GUI”</td>
<td>Employee Entry</td>
<td>SSN</td>
<td>“TextBox”</td>
<td>123456789</td>
<td>E_SSN</td>
</tr>
<tr>
<td>3</td>
<td>“GUI”</td>
<td>Employee Entry</td>
<td>Female</td>
<td>“Radio”</td>
<td>“Y”</td>
<td>Female</td>
</tr>
<tr>
<td>4</td>
<td>“GUI”</td>
<td>Employee Entry</td>
<td>Speak_Spanish</td>
<td>“Check”</td>
<td>“N”</td>
<td>Spanish</td>
</tr>
</tbody>
</table>

Table 5.3 indicates that the first input to the program was a click on the Save button in the Employee_Entry window. This might have triggered the operation or computation that is taking place. The second input was the social security number, taken from the SSN field, the value taken was 123456789 and it was saved in the E_SSN variable to be used in the computation. The outcome of the RE tool is represented in this simple form to aid the process of specification comparison and isolating the sources of errors. The outcome can also be stored directly in SpecDB as will be shown in Section 5.3.

5.1.2 Output and Post Conditions Classification and Categorization

The second and most important outcome of the RE testing tool is the full set of outputs, and system state modifications that resulted from the software during execution of a
scenario or test case. There is a great advantage to displaying to the development and quality assurance team all the outputs, and changes to the system state, both explicit and implicit so that hidden errors of commission and security flaws can be identified easily. Similar to the inputs, there are explicit and implicit outputs or post-conditions of the execution of the software application being tested. Explicit outputs are those that can be observed by functional black box testing techniques. Examples are the display of data in GUI components, a printed report, etc. However, there also can be some other implicit outputs like exporting information to another non-secure location. This kind of implicit output that cannot be spotted by black box testing techniques, and the post conditions of the software execution of a particular test case scenario are all displayed as an outcome of the RE testing tool.

In the remaining part of this section, an explanation is given to show how the RE tool describes different forms of system outputs and post conditions including system state modifications, and both implicit and explicit system outputs. In table 5.4, the RE testing tool description of twelve different forms of outputs or post conditions is given, namely, database (DB), GUI, file, sequence, message, report, variable, role, privilege, commit, savepoint and finally rollback. A modification in the database state could either be a deleted or updated row, or a newly inserted row. If the change in the database is a deletion or insertion, only the row number is presented. If the database change resulted from a row or field in a row that has been updated, the row number, field name, value stored and from which variable the value was taken are all presented by the RE testing tool. In case a database sequence has been modified, the new value is displayed by the
RE tool. Also if a message is displayed, the message text is given. If a report is one of the outputs of the software, the RE testing tool displays the destination of the report, whether it is to a printer, file, or on the screen. If a global (persistent) variable has been modified after an operation is completed, the RE testing tool also displays its new value. This can help isolate the source of some errors depending on calculations based on global variables. If the software application created or changed a database user's role or gave or revoked database privileges, the RE tool also indicates the role or privilege, specified by the displayed identifier, has been granted or revoked to the specified user. In database applications if modifications have been made but not committed (saved), and a rollback operation has been triggered, the data in the database could become corrupt, or the operation that triggered the rollback command, would reverse the operation of another unit previously executed.

System testers can have a hard time to isolate where the resulting database state got its values, after testing a previous operation, and affirming that it operates correctly. When the RE testing tool displays all the operations of the system and all the implicit outputs, the software development and testing team can immediately recognize that the execution of the scenario being tested committed the changes to the database or not, and accordingly, spot errors of omission, namely, forgetting to commit the changes to the database, so that the executed operations will not be rolled back accidentally.
Table 5.4: Output and Post Conditions Classification and Categorization

<table>
<thead>
<tr>
<th>Output #</th>
<th>Category</th>
<th>Classification</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Database(DB)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>GUI</td>
<td></td>
</tr>
<tr>
<td></td>
<td>File</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Sequence</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Report</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Variable</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Role</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Privilege</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Commit</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SavePoint</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Rollback</td>
<td></td>
</tr>
</tbody>
</table>

The RE testing tool also displays the operation of the unit or software being tested, displaying step by step how the outputs got their value. In Section 5.3, an example of the execution of the RE testing tool is given and also how it displays the inputs, outputs and operation of the software being tested.

### 5.1.2.1 Examples of Outputs Displayed by the RE Testing Tool

As described in the previous section, the RE testing tool displays several types of outputs or post conditions resulting from the execution of a scenario. In table 5.5 an example of how the RE testing tool displays a GUI output of the application being tested is shown. In this case, the RE testing tool identifies that the application running the scenario, has displayed the employee picture from a file located in C:/images/gib1.jpg, in the Emp_Picture GUI component on the Employee_Data window or screen.

<table>
<thead>
<tr>
<th>Output#</th>
<th>Category</th>
<th>Window</th>
<th>Component</th>
<th>Type</th>
<th>Value</th>
<th>From variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>“GUI”</td>
<td>Employee Date</td>
<td>Emp_Picture</td>
<td>“Image”</td>
<td>C:/images/gib1.jpg</td>
<td></td>
</tr>
</tbody>
</table>
5.2 Case Study

After discussing the benefits and motivation for the creation of the RE testing tool, and explaining how it displays different forms of inputs and outputs of the software being tested, in this section a case study to show the functionality of the tool is described. In order to demonstrate the RE testing tool's ability to spot errors, three examples are shown in Section 5.3. The examples are all different scenarios for the same case study or program. The program is an implementation of the vacation salesman commission problem, adopted from [5]. In Section 5.2.1 the specifications, or problem statement is explained. In Section 5.2.2 the implementation of the program is shown, to allow for white box testing, and to show how the RE testing tool traces the computation of the software and how it displays its outputs. As noted previously in this chapter, a technique similar to DD-paths testing is adopted by the RE testing tool. Accordingly, the DD-paths and the DD-path graph are shown in Section 5.2.3 and 5.2.4. In Section 5.3 three different examples using the problem described in Section 5.2 are given, to show how the RE testing tool can spot help spot errors.

5.2.1 Problem Statement for the Vacation Salesman Commission Program

The problem of the vacation salesman commission is adopted from [5]. Since it a reasonably small problem, it serves to show the functionality of the RE testing tool. In this section the problem statement of the vacation salesman commission program is given.

A vacation planner sells cruises, airline tickets and books shore excursions, to a certain destination. A one week Cruise costs $450 per person, airline tickets from his city to the cruise port of departure cost $300 per person and the shore excursions and activities
package costs $250 per person. The vacation planner hires a salesmen. A salesperson has to sell at least one cruise ticket, one airline ticket and one shore activities package per month to stay in the job! The vacation planner has a maximum of 70 cruise tickets to book per month, 80 airline tickets and 90 shore excursions packages. Occasionally, the salesman should report to his employer what vacation items he sold and how many of each. The employer enters this in the database, under the salesman name, and the date of sale. At the end of the month, the employer computes the salesman’s commission, by simply clicking on a button. The salesman gets a 10% on all sales up to and including $10000, 15% on the next $8000 and 20% on any sales in excess of $18000. An on-screen report is issued for the salesman showing the total number of cruises, airline tickets and shore excursions sold. The report also shows the salesperson’s total dollar sales and his commission.

5.2.2 Implementation

In order for the RE testing tool to accomplish its task of identifying faults of commission or omission, a technique similar to DD-path testing is used in the RE testing tool to determine how the software application being tested computed the resulting output, and what output values it gave, and how it modified the system state, if it did. A detailed explanation of DD-Path testing can be found in [5]. White box testing techniques require the actual implementation code. Accordingly, the implementation of the vacation salesman commission program is presented in this section. The implementation will be used in the examples in Section 5.3, and also to generate the DD-paths for this program. Each line of the implementation of the Vacation Salesman Commission program shown
below, is numbered, to help in the DD-paths generation in the subsequent sections and the examples following the case study. The code is written in PL-SQL, it is triggered when the calculate commission GUI button is clicked, to calculate the commission of the given salesman, in the given month and year. The salesman ID, month and year are three text fields in the GUI window, the implementation uses these values as inputs. After querying the database for all the sales of the month, and calculating the commission, the program displays total cruises, air tickets and activities sold, total sales, and the commission in five text fields on the same GUI window. Accordingly, the system takes three GUI inputs, other information from the database, and displays five outputs. All exception handling and error checking code has been taken out for simplicity. The code given is executed after checking that all input is given and in the correct format. Below is the implementation:
1 - - Salesman Commission Calculate Button WHEN-BUTTON-PRESSED Trigger

2   Declare cruise, air_ticket, activity integer;

3   Cruise_Price, Air_Price, Activity_Price ,Total_Sales, commission  real;

4   Number_Cruises_Sold, Number_Air_Tickets_Sold, Number_Activities_Sold integer;

5   Dollar_Cruise_sales, Dollar_Air_sales, Dollar_Activity_sales  real;

6   cursor all_month_sales is

7      select s_cruise, s_air_ticket, s_activity   from sales

8      where s_salesman = :salesman and s_month = :month and s_year = :year;

9 Begin

10   Cruise_Price = 450;

11   Air_Price = 300;

12   Activity_Price = 250;

13   Number_Cruises_Sold=0;

14   Number_Air_Tickets_Sold=0;

15   Number_Activities_Sold=0;

16   open all_month_sales;

17   loop

18      fetch all_month_sales into cruise, air_ticket, activity;

19      exit when all_month_sales %notfound;

20      Number_Cruises_Sold = Number_Cruises_Sold + cruise;

21      Number_Air_Tickets_Sold = Number_Air_Tickets_Sold + air_ticket;

22      Number_Activities_Sold = Number_Activities_Sold + activity;

23   end loop

24   close all_month_sales;

25   :Total_Cruises_Sold := Number_Cruises_Sold;

26   :Total_Air_Ticket_Sold := Number_Air_Tickets_Sold;
:Total_Activities_Sold := Number_Activities_Sold;
Dollar_Cruise_sales = Cruise_Price * Number_Cruises_Sold;
Dollar_Air_sales = Air_Price * Number_Air_Tickets_Sold;
Dollar_Activity_sales = Activity_Price * Number_Activities_Sold;
Total_Sales = Dollar_Cruise_sales + Dollar_Air_sales + Dollar_Activity_sales;
:Total_Sales = Total_Sales;

if Total_Sales > 18000
ten
commission = 0.10 * 10000;
commission = commission + (0.15 * 8000);
commission = commission + (0.20*(Total_Sales - 18000))
else if Total_Sales > 10000
ten
commission = 0.10 * 10000;
commission = commission + 0.15 * (Total_Sales - 10000)
else commission = 0.10 * Total_Sales;
end if;
end if;
:Salesman_Commission := commission;
end;
5.2.3  Program Graph for the Vacation Salesman Commission Problem

From the implementation in Section 5.2.2, one could come up with the program graph shown in Figure 5.1. A detailed description of how to obtain this program graph from the implementation is available in [5]. The program graph is stage needed to be created to complete extract the program's DD-paths. The DD-paths are used to trace program behavior on specific input.

Figure 5.1: Program Graph for the Vacation Salesman Commission Problem
5.2.4 DD-Path Graph for the Vacation Salesman Commission Problem

From the program graph in Figure 5.1 in Section 5.2.3, the DD-Paths graph in Figure 5.2 can be automatically derived, from the DD-paths in table 5.6. Jorgensen in [5] explains how to derive the DD-Paths for an implementation. The DD-paths graph and the DD-paths are used in the RE tool. Figure 5.3 shows a higher level module diagram for the RE tool. The unit implementation and test case scenario are taken as inputs. The different process of the RE tool are shown, followed by the RE tool output, representing the inputs, and outputs to and from the system, both implicit and explicit, and also the process that the software executed to reach the results shown.

Table 5.6: DD-Path Graph for the Vacation Salesman Commission Problem

<table>
<thead>
<tr>
<th>DD-path</th>
<th>Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>9</td>
</tr>
<tr>
<td>B</td>
<td>10-16</td>
</tr>
<tr>
<td>C</td>
<td>17</td>
</tr>
<tr>
<td>D</td>
<td>18</td>
</tr>
<tr>
<td>E</td>
<td>19</td>
</tr>
<tr>
<td>F</td>
<td>20-23</td>
</tr>
<tr>
<td>G</td>
<td>24-32</td>
</tr>
<tr>
<td>H</td>
<td>33</td>
</tr>
<tr>
<td>I</td>
<td>34-37</td>
</tr>
<tr>
<td>J</td>
<td>38</td>
</tr>
<tr>
<td>K</td>
<td>39-41</td>
</tr>
<tr>
<td>L</td>
<td>42</td>
</tr>
<tr>
<td>M</td>
<td>43</td>
</tr>
<tr>
<td>N</td>
<td>44</td>
</tr>
<tr>
<td>O</td>
<td>45</td>
</tr>
<tr>
<td>P</td>
<td>46</td>
</tr>
</tbody>
</table>

Figure 5.2: DD-Path Graph for the Vacation Salesman Commission Problem
In the following Section 5.3 three examples to test the vacation salesman commission program is demonstrated, based on the implementation, DD-paths, and DD-paths graph generated in this section.

![Figure 5.3: RE Tool Model Diagram](image)

### 5.3 Examples

From the Case study described in Section 5.2, three examples are discussed in the following subsections. Each of the examples demonstrates a different state showing how the RE tool can spot errors of commission that are not identified by other tools, discussed in the literature review in Chapter 1.
5.3.1 Test Case 1

5.3.1.1 Scenario

This example demonstrates a simple scenario for the purpose of showing how the RE tool technique works. The salesman reports twice to his employer during a certain month. The first time he reports that he sold 8 cruise vacations, 12 airline tickets and 10 shore excursion packages. The second time he reports that he sold 12 cruise vacations, 8 airline tickets and 4 shore excursion packages. The employer enters the data as sent each time. Therefore the database will have two records in the sales table for that specific salesman for that specific month. At the end of the month the employer enters the employee name, and the required month and year, then hits the Calculate Commission Button to view the report on the screen.

5.3.1.2 Expected Output

According to the calculations, the total sales amount sums up to $18500. Since the sales are greater than $18000, the salesperson should get a 10% on all sales up to and including $10000, 15% on the next $8000 and 20% on the remaining $500, adding up to $2300 in commission.

5.3.1.3 Expected GUI Output

The expected GUI output on a system monitor for the above scenario is shown in Figure 5.4. There is no database state change expected from this scenario, it is only an inquiry, not a data entry scenario. Accordingly, the input and output database states
should be identical. Also, there should not be any other output or post condition other than the expected GUI output shown in Figure 5.4.

<table>
<thead>
<tr>
<th>Employee</th>
<th>Gibson</th>
</tr>
</thead>
<tbody>
<tr>
<td>Month</td>
<td>9</td>
</tr>
<tr>
<td>Year</td>
<td>2003</td>
</tr>
<tr>
<td>Total Cruises Sold this month</td>
<td>20</td>
</tr>
<tr>
<td>Total Air Tickets Sold this month</td>
<td>20</td>
</tr>
<tr>
<td>Total Shore Excursions Sold this month</td>
<td>14</td>
</tr>
<tr>
<td>Total Sales</td>
<td>$18500</td>
</tr>
<tr>
<td>The Salesman Commission for this month</td>
<td>$2300</td>
</tr>
</tbody>
</table>

Figure 5.4: The Expected GUI Output for Test Case 1

### 5.3.1.4 The RE Tool Display of the Inputs for Test Case 1

When the RE testing tool runs on the vacation salesman commission program implementation and for the specified scenario in test case 1, it will produce three main outcomes. First, it will give all the inputs, implicit and explicit, those inputs are shown in this section. It will also show which DD-paths have been traversed and the operation of the software, and how it computed each value of the outputs. The third outcome of the RE testing tool is a list of all the outputs produced by the software during execution of the test case. The operation and outputs are shown in the next section.

Tables 5.7 and 5.8 show how the RE testing tool displays the inputs of the vacation salesman commission program, given from the GUI and acquired from the database. These two tables are the first outcomes of the RE testing tool for test case 1.
5.3.1.5 Process and Actual Output

The RE testing tool also displays two other outcomes, the outputs of the program after executing the test case, and how the values of the output where obtained. The RE testing tool displays the following for test case 1:

At execution time the program traversed the following nodes in the DD-Path graph: ABCDEF CDEF CDE GHI NOP. The following statements where executed: 9, 10-16, 17, 18, 19, 20-23, 17, 18, 19, 20-23, 17, 18, 19, 24-32, 33, 34-37, 44, 45, 46

The RE testing tool also displays the outputs and the calculations. The development or testing team can refer to the calculations if errors where found in the output. The calculations or the operation of the program unit resulting in the values of the outputs will thus aid in fault isolation. The following is what the RE testing tool displays for test case 1, outputs are tabulated.

### Table 5.7: The RE Testing Tool Displayed Database Inputs for Test Case 1

<table>
<thead>
<tr>
<th>Input #</th>
<th>Category</th>
<th>Table</th>
<th>Field</th>
<th>Row #</th>
<th>Value</th>
<th>Into variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>DB</td>
<td>Sales</td>
<td>s_cruise</td>
<td>15</td>
<td>8</td>
<td>cruise</td>
</tr>
<tr>
<td>2</td>
<td>DB</td>
<td>Sales</td>
<td>s_air_ticket</td>
<td>15</td>
<td>12</td>
<td>air_ticket</td>
</tr>
<tr>
<td>3</td>
<td>DB</td>
<td>Sales</td>
<td>s_activity</td>
<td>15</td>
<td>10</td>
<td>activity</td>
</tr>
<tr>
<td>4</td>
<td>DB</td>
<td>Sales</td>
<td>s_cruise</td>
<td>17</td>
<td>12</td>
<td>cruise</td>
</tr>
<tr>
<td>5</td>
<td>DB</td>
<td>Sales</td>
<td>s_air_ticket</td>
<td>17</td>
<td>8</td>
<td>air_ticket</td>
</tr>
<tr>
<td>6</td>
<td>DB</td>
<td>Sales</td>
<td>s_activity</td>
<td>17</td>
<td>4</td>
<td>activity</td>
</tr>
</tbody>
</table>

### Table 5.8: The RE Testing Tool Displayed GUI Input for Test Case 1

<table>
<thead>
<tr>
<th>Input #</th>
<th>Category</th>
<th>Window</th>
<th>Component</th>
<th>Type</th>
<th>Value</th>
<th>Into variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>GUI</td>
<td>Commission_Calc</td>
<td>Salesman</td>
<td>TextBox</td>
<td>Gibson</td>
<td>NULL</td>
</tr>
<tr>
<td>8</td>
<td>GUI</td>
<td>Commission_Calc</td>
<td>Month</td>
<td>TextBox</td>
<td>9</td>
<td>NULL</td>
</tr>
<tr>
<td>9</td>
<td>GUI</td>
<td>Commission_Calc</td>
<td>Year</td>
<td>TextBox</td>
<td>2003</td>
<td>NULL</td>
</tr>
</tbody>
</table>

...
Cruise_Price = 450
Air_Price = 300
Activity_Price = 250
Number_Cruises_Sold=0
Number_Air_Tickets_Sold=0
Number_Activities_Sold=0
Number_Cruises_Sold = 0 + 8
Number_Air_Tickets_Sold = 0 + 12
Number_Activities_Sold = 0 + 10
Number_Cruises_Sold = 8 + 12
Number_Air_Tickets_Sold = 12 + 8
Number_Activities_Sold = 10 + 4

Table 5.9: GUI Output for Test Case 1 – Part1

<table>
<thead>
<tr>
<th>Output #</th>
<th>Category</th>
<th>Window</th>
<th>Component</th>
<th>Type</th>
<th>Value</th>
<th>From Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>GUI</td>
<td>CommissionCalc</td>
<td>Total_Cruises_Sold</td>
<td>Textbox</td>
<td>20</td>
<td>Number_Cruises_Sold</td>
</tr>
<tr>
<td>2</td>
<td>GUI</td>
<td>CommissionCalc</td>
<td>Total_Air_Ticket_Sold</td>
<td>Textbox</td>
<td>20</td>
<td>Number_Air_Tickets_Sold</td>
</tr>
<tr>
<td>3</td>
<td>GUI</td>
<td>CommissionCalc</td>
<td>Total_Activities_Sold</td>
<td>Textbox</td>
<td>14</td>
<td>Number_Activities_Sold</td>
</tr>
</tbody>
</table>

Dollar_Cruise_sales = 450 * 20
Dollar_Air_sales = 300 * 20
Dollar_Activity_sales = 250 * 14
Total_Sales = 9000 + 6000 + 3500

Table 5.10: GUI Output for Test Case 1 – Part2

<table>
<thead>
<tr>
<th>Output #</th>
<th>Category</th>
<th>Window</th>
<th>Component</th>
<th>Type</th>
<th>Value</th>
<th>From Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>GUI</td>
<td>CommissionCalc</td>
<td>Total Sales</td>
<td>TextBox</td>
<td>18500</td>
<td>Total Sales</td>
</tr>
</tbody>
</table>

Since Total_Sales > 18000
Then commission = 0.10 * 10000;
commission = 1000 + (0.15 * 8000);
commission = 2200 + (0.20*(18500 - 18000))

Table 5.11: GUI Output for Test Case 1 – Part3

<table>
<thead>
<tr>
<th>Output#</th>
<th>Category</th>
<th>Window</th>
<th>Component</th>
<th>Type</th>
<th>Value</th>
<th>From Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>GUI</td>
<td>CommissionCalc</td>
<td>Salesman Commission</td>
<td>Textbox</td>
<td>2300</td>
<td>Commission</td>
</tr>
</tbody>
</table>
The RE testing tool thus displays and confirms that there were only five outputs to the program for test case 1. No other database outputs, or state changes have been executed by the unit.

5.3.1.6 Comparing Expected to Actual Output

In this example it is clear that the expected output in Figure 5.4 matches the actual output in tables 5.9, 5.10 and 5.11, and therefore, there is little or no need to look at the execution process. The specifications of the implemented program can be fed in SpecDB, together with the actual outputs and the calculation formulas. Those implementation specifications can then be compared to the required specifications and an automated tool can make the correct decision whether the implementation matches the required specifications or not. Also if there are discrepancies between the actual and the required specifications, the outcome of the RE tool helps in isolating the source of the errors. The RE tool numbers the implementation statements, accordingly, it indicates which statement caused the difference between the required specifications and actual the actual code.

SpecDB is used to host the specifications resulting from the RE testing tool. An example is given in this section to clarify how an automatic comparison between the required specifications and those resulting from the RE testing tool, i.e. the specifications of the implemented code, can be made using SpecDB. A snapshot of a portion of the specifications will be used to demonstrate the principal technique. The specifications of the output for the vacation salesman commission problem are then represented in SpecDB. In test case 1, both the required and actual specifications
match. However, to show how a discrepancy can be automatically identified, a software error is injected. Software fault injection was researched in [102]. The software error injected is not in the code, but in the GUI, assuming that in the vacation salesman commission program, after the employer entered the correct salesman identification, month and date, and pressed the calculate commission button, the program did not display anything in the five fields. The same program unit implementation in test case 1 is used again. By viewing the result of the RE testing tool in the previous sections, it is noticed that the program unit executed correctly. For the purpose of comparing the actual specifications to the faulty ones, the scenario is changed, so that the output is not displayed in the correct window, it is displayed in another screen. From the user's point of view, no output is generated when he clicks the button, although, the output is computed correctly, is it displayed in the wrong location in the GUI. Given the required specifications and the observed specification of the fault injected test case 1, resulting from the RE tool, it is a trivial task to note where the error occurred. After injecting this fault the results in tables 5.9, 5.10 and 5.11 should be slightly different, the window name should change and/or the component name should change. The RE testing tool outputs in this case, display the values in table 5.12.
For the purpose of this example, only a snapshot of the original required specifications represented in SpecDB is shown in the Var, Dataflow and the Input_Output_Definition tables.

Table 5.12: GUI Output for Test Case 1 After Fault Injection

<table>
<thead>
<tr>
<th>Output #</th>
<th>category</th>
<th>Window</th>
<th>Component</th>
<th>Type</th>
<th>Value</th>
<th>From Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>GUI</td>
<td>Sale_Report</td>
<td>Total_Cruises_Sold</td>
<td>Textbox</td>
<td>20</td>
<td>Number_Cruises_Sold</td>
</tr>
<tr>
<td>2</td>
<td>GUI</td>
<td>Sale_Report</td>
<td>Total_Air_Ticket_Sold</td>
<td>Textbox</td>
<td>20</td>
<td>Number_Air_Tickets_Sold</td>
</tr>
<tr>
<td>3</td>
<td>GUI</td>
<td>Sale_Report</td>
<td>Total_Activities_Sold</td>
<td>Textbox</td>
<td>14</td>
<td>Number_Activities_Sold</td>
</tr>
<tr>
<td>4</td>
<td>GUI</td>
<td>Sale_Report</td>
<td>Total_Sales</td>
<td>Textbox</td>
<td>18500</td>
<td>Total_Sales</td>
</tr>
<tr>
<td>5</td>
<td>GUI</td>
<td>Sale_Report</td>
<td>Salesman_Commission</td>
<td>Textbox</td>
<td>2300</td>
<td>Commission</td>
</tr>
</tbody>
</table>

Tables 5.13, 5.14 and 5.15 contain a snapshot of the data in Var, Dataflow and the Input_Output_Definition tables in SpecDB, reflecting the original required specifications. The snapshots of only the fields in the SpecDB tables that are used are shown here, for simplification. Note that the program line numbers in Section 5.2.2 are not the same as the POS_Num in the specifications. Line numbers used in the implementation in Section 5.2.2 was for the purpose of extracting the DD-paths graph. However, when the specifications are entered in SpecDB directly or through the RE tool, different POS_Num numbers are given, to reflect actual program actions. For example, all variable or type declarations are inserted in the Var or Type tables prior to inserting the dataflow data in the Dataflow table.
The above specifications in the Dataflow table in SpecDB, specify that the sixteenth operation that should take place in the implemented vacation salesman program should be an output. The output is defined in the Input_Output_Definition table, shown in Table 5.15 below. But before we examine the snapshot of the specifications in the Input_Output_Definition table, we need to look at the snapshot of the SpecDB Var table, in Table 5.14.

The Var table snapshot shown in Table 5.14 above, indicates that the variable Total_cruise_sold is an output, specifically, a GUI numeric output, that should be displayed in the CommissionCalc window or screen. Finally, from the above two SpecDB tables and the snapshot of the SpecDB Input_Output_Definition table in Table 5.15, it is specified that the output in the sixteenth operation is a display of the
value of the Number_Cruises_Sold variable (Var_ID = 9) in the Total_cruise_sold GUI output (Var_ID = 18) in the CommissionCalc window or screen.

Table 5.15: SpecDB Input_Output_Definition Table Snapshot – Original Specifications

<table>
<thead>
<tr>
<th>Pos_Num</th>
<th>Var_ID</th>
<th>Input_Output</th>
<th>Source_Destination_Var_ID</th>
<th>…</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>18</td>
<td>Output</td>
<td>9</td>
<td>..</td>
</tr>
<tr>
<td>17</td>
<td>19</td>
<td>Output</td>
<td>10</td>
<td>..</td>
</tr>
<tr>
<td>18</td>
<td>20</td>
<td>Output</td>
<td>11</td>
<td>..</td>
</tr>
<tr>
<td>23</td>
<td>21</td>
<td>Output</td>
<td>7</td>
<td>..</td>
</tr>
<tr>
<td>34</td>
<td>22</td>
<td>Output</td>
<td>8</td>
<td>..</td>
</tr>
<tr>
<td>..</td>
<td>..</td>
<td>..</td>
<td>..</td>
<td>..</td>
</tr>
</tbody>
</table>

After examining the snapshot of the required specifications stored in SpecDB above, we will now examine a snapshot of the specifications of the program after fault injection. After examining how the specifications of the faulty program are represented in SpecDB as a result of the RE testing tool, the errors can then be automatically isolated and reported; as shown below.

In Table 5.12, the RE testing tool display of the GUI Output for test case 1 after fault injection was demonstrated. The outcome of the RE tool is then stored in SpecDB, in the same three tables, Var, Dataflow and Input_Output_Definition. The RE testing tool specifications stored in the Dataflow, and input_Output_Definition tables can be identical to that of Tables 5.13 and 5.15 for test case 1 after fault injection. Table 5.16, however, shows that the five output variables (text boxes) are in another GUI window.
Comparing the two instances of the SpecDB tables to one another, those of the required specifications and the specifications of the implemented code resulting from the RE tool, is a trivial task that is easy to automate. However, much attention should be dedicated to identifying variable names and matching the implemented variable names to those of the specified design. If the implementation uses the exact variable names as those in the specifications and design, then comparing the RE testing tool specifications of the implementations to those of the required specifications is an easy task. The same consideration should be given to the operation position numbers in the dataflow table, to facilitate the comparison of the two sets of specifications and the process of fault isolation. The next example illustrates how RE Testing tool helps spot faults of commission, when either the program generates an unspecified output, or generates the wrong output, due to an incorrect computation.

Table 5.16: SpecDB Var Table Snapshot – RE Tool Implementation Specifications

<table>
<thead>
<tr>
<th>var_ID</th>
<th>Var_Name</th>
<th>Type</th>
<th>Is_Input</th>
<th>Is_output</th>
<th>is-GUI-object</th>
<th>GUI_window</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>Total_Sales</td>
<td>Real</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Commission</td>
<td>Real</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Number_Cruises_Sold</td>
<td>Int</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>Number_air_tickets_Sold</td>
<td>Int</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>Number_Actions_Sold</td>
<td>Int</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>Total_cruise_sold</td>
<td>Int</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>Sale_Report</td>
</tr>
<tr>
<td>19</td>
<td>Total_air_sold</td>
<td>Int</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>Sale_Report</td>
</tr>
<tr>
<td>20</td>
<td>Total_activities_sold</td>
<td>Int</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>Sale_Report</td>
</tr>
<tr>
<td>21</td>
<td>Total_Sales</td>
<td>Real</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>Sale_Report</td>
</tr>
<tr>
<td>22</td>
<td>Salesman_Commission</td>
<td>Real</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>Sale_Report</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
5.3.2 Test Case 2

In test case 1, the correct implementation in Section 5.2.2 was used. In order to demonstrate how the RE testing tools spots errors of commission, a slight modification of the code will be used. If the program was generated using the wrong implementation, it will result in errors. Accordingly, in test case 2, only one line of code will be modified to show how the RE testing tool identifies errors of commission. Per the specifications, the data entered by the employer in salesman identification, month and year fields should not change, when the user clicks on the calculate commission button. The five other fields should display the required information. In test case 2, we will assume that the program in line 25 has the following statement:

```pascal
25    : month := Number_Cruises_Sold;
```

instead of

```pascal
25    : Total_Cruises_Sold := Number_Cruises_Sold;
```

In this case, the programmer made an error while writing the statement at line 25. Consequently, the actual output displayed on the GUI screen, will appear as in Figure 5.5. The GUI output clearly indicates an error, since the total sales and commission amounts are based on more than just 20 air tickets and 14 shore excursions. The value in the month field has also changed to 20. Since an error is obvious in the GUI output, the report displayed by the RE testing tool helps to isolate where the error occurred. In test case 2, the actual input is identical to that in test case 1, detailed in Tables 5.7 and 5.8.
5.3.2.1 Process and Actual Output

For test case 2, the DD-paths and all calculations will be identical to the process shown in test case 1. Hence, the calculation for the total number of cruises sold is correct. The program just does not display it in its position. The only difference between the correct program implementation in test case 1 and that of test case 2 is in the first set of outputs shown in Table 5.17. Output number 1 in Table 5.17 highlights the error.

<table>
<thead>
<tr>
<th>Output #</th>
<th>Category</th>
<th>Window</th>
<th>Component</th>
<th>Type</th>
<th>Value</th>
<th>From Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>GUI</td>
<td>CommissionCalc</td>
<td>Month</td>
<td>Textbox</td>
<td>20</td>
<td>Number_Cruises_Sold</td>
</tr>
<tr>
<td>2</td>
<td>GUI</td>
<td>CommissionCalc</td>
<td>Total_Air_Ticket_Sold</td>
<td>Textbox</td>
<td>20</td>
<td>Number_Air_Tickets_Sold</td>
</tr>
<tr>
<td>3</td>
<td>GUI</td>
<td>CommissionCalc</td>
<td>Total_Activities_Sold</td>
<td>Textbox</td>
<td>14</td>
<td>Number_Activities_Sold</td>
</tr>
</tbody>
</table>
From the GUI output, the tester can see that the text box Total_Cruises_Sold was not modified by the program. Whereas the text box Month was modified. This is a fault of commission since the program should not have modified that Month field. In this example it was given the value 20. If there was a restriction on the integer values that are allowed to be put in the Month text box from 1 to 12, this might have cause a run time error as well. There is also a fault of omission since the program did not modify the Total_Cruises_Sold text box, when it should have displayed in it the value of the variable Number_Cruises_Sold. These faults could be easily spotted by the testers, once the RE testing tool displays the program output shown in Table 5.17 and the process. The RE testing tool technique thus far helped to identify the faults by showing the source of the implementation's actual output. It also helped isolate where the problem occurred.

As in test case 1, the results of the RE testing tool in SpecDB are shown. The required specifications represented in SpecDB remain unchanged for all the test cases. In test case 2, the specifications from the RE testing tool help to automatically isolate the source of the error and show that the specifications of the implementation are different, in only one location. Table 5.18 shows that the information in the Var table resulting from the RE tool specifications are correct, and match the original specifications.

Table 5.18: SpecDB Var Table Snapshot – RE Tool Specifications – Test Case 2

<table>
<thead>
<tr>
<th>Var_ID</th>
<th>Var_Name</th>
<th>Type</th>
<th>Is_Input</th>
<th>Is_output</th>
<th>Is_GUI_object</th>
<th>GUI_window</th>
</tr>
</thead>
<tbody>
<tr>
<td>18</td>
<td>Total_cruise_sold</td>
<td>Int</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>CommissionCalc</td>
</tr>
<tr>
<td>..</td>
<td>..</td>
<td>..</td>
<td>..</td>
<td>..</td>
<td>..</td>
<td>..</td>
</tr>
</tbody>
</table>

136
Table 5.19 shows the specifications of the implemented code, resulting from the RE testing tool. Compared to table 5.15, the error lies in the Var_ID, it should have been 18(Total_cruise_sold) not 16(Month). By a trivial automated comparison of the two sets of specifications, the error can be easily isolated.

| Pos_Num | Var_ID | Input_Output | Source_Destination_Var_ID | ...
|---------|-------|--------------|---------------------------|---
| 16      | 16    | Output       | 9                         | ..
| ..      | ..    | ..           | ..                        | ..

### 5.3.3 Test Case 3

The third example shows errors that most probably are missed by the testing team. These involve errors that happen in the database and are not immediately visible to the tester. This happens when all the items in the GUI return the expected output, however, more happens in the background that the user/tester cannot immediately see, but that also leaves the database in a possibly unstable or wrong state, i.e. containing erroneous data. Such implicit errors, can be easily identified when the tester examines the outcome of the RE tool containing the list of program outputs given in the actual output table. The following example demonstrates those invisible errors of commission and/or omission. The advantage of this technique lies in its ability to recognize errors as early as on the first run of a specific unit. To demonstrate this, the following scenario is used.

The expected output for the *Calculate Commission* unit is only to fill out five fields in the GUI and not to change the underlying database in any way. In this example the code in line 46 is changed to show how the RE testing tool identifies hidden errors of commission. The original code in line 46 of the implementation was:
Instead, we will assume that the programmer put the following lines of code:

```sql
update sales
set cruise = Total_Sales
where s_salesman = :salesman and s_month = :month and s_year = :year;
end;
```

This updates the entire salesman’s cruise sales in that specific month and year (taken from the GUI text fields) to the total dollar sales amount for the cruises, air_tickets and activities. This is of course an error. However, it is not visible to the tester if she uses ordinary testing procedures to compare the actual output to the expected output. The outputs are correct for the first time, but then the data in the database changes, and when the unit is executed again with the same inputs, it will generate different values each time! Such an error may not be visible during the testing phase when the database is not fully populated, but after a year of usage, if the employer wants a report of each salesman’s sales and commissions earned, the report will give him totally different numbers than those the salesmen actually received during the year.

In order to spot such an error early one needs to either compare the database state before and after the execution of the unit code or use the proposed technique in this chapter. The output process will reveal the following, making it very easy for the tester to identify that there is an error of commission where the unit was programmed to execute more than it was intended to execute. The actual input will be identical to that in test case 1.
The process will have statements 46-48 extra. The output will be identical to that in Example 1; five GUI outputs will be displayed by the RE testing tool in addition to the database modification in Table 5.20, which is displayed by the RE testing tool at the end of the program operation.

Table 5.20: Database Output for Example 3

<table>
<thead>
<tr>
<th>Output#</th>
<th>Category</th>
<th>Table</th>
<th>Change</th>
<th>Field</th>
<th>Row #</th>
<th>Value</th>
<th>From Var</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>DB</td>
<td>Sales</td>
<td>Updated</td>
<td>s_cruise</td>
<td>15</td>
<td>18500</td>
<td>Total_Sales</td>
</tr>
<tr>
<td>7</td>
<td>DB</td>
<td>Sales</td>
<td>Updated</td>
<td>s_cruise</td>
<td>17</td>
<td>18500</td>
<td>Total_Sales</td>
</tr>
</tbody>
</table>

Similar errors can be spotted when executing the Commit database command after an error is introduced to the database. Also other errors that can be spotted with this technique are those resulting from deleting correct information that was stored in the database earlier but was not yet committed and a Rollback command is executed. When the results of the RE tool are fed into SpecDB, it will be shown that there are an extra set of specifications resulting from the implementation that do not exist in the original specifications, thus isolating an error of commission.

5.4 Conclusion

In this chapter one of the applications of SpecDB is introduced, namely a reverse engineering approach is used to log the actual execution of the program from the code due to some input. Once the program actions have been logged, the specifications of the actual executed code are fed into SpecDB to allow for automatically matching them with the expected specifications. In this chapter, the actual program outputs and post conditions are presented as an output of the RE testing tool, in a way that non-technical testers and other members of the software development team can understand. As a result of the RE testing
tool, the development team will have two documents, the required specifications, and the actual implemented specifications. The observed output can then be easily compared to the expected output based on the program specifications and oracles that determine what the expected program output should be. If the specifications of the software were already logged in SpecDB, then both specifications can be automatically compared, and errors automatically reported.

Specifications can be stored in SpecDB from the RE tool without a specific input for a test case. The RE testing tool as shown in the case study in this chapter automates the process of program DD-paths graph creation. After traversing the code with the white box testing technique shown using DD-paths testing, the specifications of the implemented code can be fed directly into SpecDB following the dataflow of the implementation. Accordingly, the RE tool can feed the specifications of the implemented units into SpecDB without the input data from a test case.
A survey of related literature, and the current available tools in the market was given in Chapter 1. In Chapter 2, the motivation for the creation of SpecDB was discussed together with its detailed design. In Chapters 4 and 5 two testing tools using SpecDB were introduced. The focus of this chapter is on using the machine-readable specifications represented in SpecDB to enhance other tools developed or designed by other researchers. As an example, AGENDA, developed by Chays et. al. in [27,28,54,55,56,57] is chosen. The research involving AGENDA was summarized in Chapter 1. After introducing the machine interpretable specification of SpecDB, the research on AGENDA and other testing tools or techniques, can be enhanced. One of the motivations for the creation of SpecDB was to enhance already existing automated testing systems and complete current recommendations in some of the published literature. This chapter focuses on improvements that can be made to those testing tools using the SpecDB representation and tools described in the previous chapters of this dissertation. In each section a limitation of other tools is emphasized, a solution using SpecDB is proposed and examples of the solution are provided.
6.1 Using the Database Representation of SpecDB to Expand the Testing Domain

Some testing tools, including AGENDA use linked lists and files to store the information needed about rules, constraints and the schema of the application being testing. Tools implemented using linked lists are not as easy to modify as database queries, and database schema design. Accordingly, scalability is an advantage of using a database design over a linked list representation. Using the database of SpecDB is a more effective approach, since much more software specifications can be stored in the SpecDB database. Storing more specifications result in an ability to automatically create better test cases and automate the process of test case result verification. Also enforcing the constraints on the SpecDB database is a means of testing the specifications for completeness and accuracy to a great extent. The authors of AGENDA investigated memory capacity and had concerns about size limits for the linked lists used. However, the use of tables in SpecDB to store and access the information as opposed to linked lists, eliminates the concern about memory overflow.

To demonstrate how storing more specifications result in an ability to automatically create better test cases, examples are given to show what test cases can be generated by AGENDA, compared to those created by a tool using SpecDB. In the vacation salesman program described in Chapter 5, the only portion of the unit implementation in Section 5.2.2 that can be tested using AGENDA, is line 7 and 8 (the Select statement) of the 46 lines of code. Tools built on SpecDB, however, can test imperative code as well. The RE testing tool demonstrated in Chapter 5 traversed the entire unit implementation, as shown by the three
examples in Chapter 5. Accordingly, better test coverage is possible using the SpecDB. However, in order to benefit from the results of other research, it is shown how it is an uncomplicated task to expand SpecDB, to represent the information used by other tools. Again this is demonstrated with an example from AGENDA. Translating the information needed and used by AGENDA in linked lists to the SpecDB representation is shown in Figure 6.1. In this figure it is shown how the data used to generate test cases and database states in AGENDA can be stored in SpecDB. Each database operation to test in AGENDA is given a unique query ID. This ID is stored in SpecDB in the DB_Test_Operation table (operation ID). The host variables used or updated in the query or database operation are stored in the Host_Var table. The descriptions of those variables are also stored in this table.

In AGENDA only one attribute is observed for modifications per operation. The SpecDB representation offers an expansion to this simple query testing, to include multiple attributes per operation. This is achieved by storing those attributes in the Attributes_Changed table. Also multiple pre and post-conditions can be observed and tested using the SpecDB representation.

The Pre_Post_Conditions table in SpecDB stores the operation number together with the corresponding condition number. The conditions are stored in the Predicate table, using the Condition_Num to reference the Pos_Num. The authors of AGENDA explained that only simple mathematical numeric calculations for post-conditions can be carried out using their tool. In the SpecDB representation, very complex operations can be specified using the generality of the Predicate table and the complicated operations that it can represent. The Predicate table can also represent operations on other types besides numbers, text, date, etc.
Data groups used by AGENDA to populate the tables with valid data and generate test cases, are stored in the Data_Group table in SpecDB, together with the probability of each group. The probability was used by AGENDA as a heuristic to guide the processes of data and test case generation. Boundary values are stored in the Type_Restrictions table in SpecDB. In SpecDB every attribute in every table can have a distinct type with specified allowable values. Using this representation in SpecDB allows for more complex type restrictions. For example specifying a range (from -5 to 5) excluding one or more values (like 0). If the attribute is better described by a list of values (e.g. month name), the values are represented in the Types_LOV table.

Using AGENDA, the tester specifies some valid values for each of the data groups to guide the database state and test case generation processes. The list of valid values for each data
group is stored in the Equivalence_Class table in SpecDB. The attribute can be of any basic type. Other more complex types can also be used by expanding this table in SpecDB, giving the tester more flexibility.

The List_of_Tables table in SpecDB hosts the required information to be used by AGENDA. The attributes and their respective constraints are stored in the Table_Field_Description table in SpecDB. In AGENDA, information about the constraints on each attribute is gathered, but the testing tool does not check that these constraints are enforced. The automated constraint generator tool created using SpecDB creates the constraints and triggers for more complex business rules, to enforce them at the database level as opposed to the application level, hence producing a better database state.

6.2 Enhancing State Validation Tools Using SpecDB

In order to test the operation of parts of a software application, some tools, including AGENDA, check the state of the resulting system after test case execution. Another enhancement to AGENDA using the SpecDB representation is that of the state validator. AGENDA creates log tables to log all modifications after update, delete, or insert operations. Temporary constraints are created on the log tables to check the database state changes after test execution. AGENDA also checks whether the number of affected rows by the test execution equals that expected. If not, AGENDA reports an error, indicating that either too many or too few records have been affected. It is then the tester’s task to go through the database and the log tables to identify the error.
There are some errors that can occur and not be identified by AGENDA, in its current technique. AGENDA checks that the post-conditions have been fulfilled using the temporary constraints on the log tables. However, if the specified post-conditions have been fulfilled but also other modifications have been made to the database too, i.e. errors of commission, that would be a software error that would not be spotted by AGENDA. For example, if the operation being tested, gives an annual raise to all employees. AGENDA will make sure that the salaries of all the employees have been raised by the specified corresponding rates. It would not however, spot an error of a change in the department made to all employees. Also AGENDA would not spot other operations in the database that are not linked to the database table being updated, for example, the operation grants privileges to unauthorized users to access the database. Using SpecDB the latter problem is solved by using the reverse engineering approach to testing the database system, where all of the modifications made to a system are logged for each test operation. Those errors of commission can be spotted very easily using this technique, as shown in the examples of Chapter 5.

If the number of rows modified equals that expected, and the post-conditions are satisfied by the temporary constraints on the log tables, but other attributes have been modified too, AGENDA does not report an error. In order to report such software errors, using SpecDB more constraints can be automatically generated on the log tables to check if all the other attributes, (other than the ones in the Attributes_Changed table for the specified operation) have not changed. If the post-conditions have not been satisfied or satisfied but other attributes have also been modified, the SpecDB tool will report an error.
6.3 Expanding the Testing Scope Beyond Variables and Database States

Testing tools that are not based on the specifications are limited in the areas of the software that can be tested. AGENDA for example, tackles only variables and database states. It does not tackle operations where the variables or database states are not changed, but files, printed reports, magnetic card programming, etc. Many Applications use numerous types of inputs and outputs, besides database state changes. SpecDB’s reverse engineering testing tool handles all these different types of inputs and outputs, as seen in Chapter 5.

6.4 Testing Different Operation Types

Another limitation of testing tools not based on software specifications, is the types of operations tested. AGENDA tests four types of database operations, namely, insert, update, delete and select. Using the reverse engineering testing tool of SpecDB, other database operations can also be tested, and their results logged, together with imperative code, i.e. programming constructs that are not related to database operations. Using SpecDB the following database operations can be tested: Grant, Revoke, Create and Drop(view, table, constraint, sequence, etc.), Roles, Access Sequence, Commit, Rollback and Savepoint. Other operations can be easily added to SpecDB. The reverse engineering SpecDB testing tool observes all modifications that happen in the database state, the application variables, and the input/output domains. Each of the above mentioned database operations are logged with all its details, as described in the reverse engineering SpecDB testing tool in Chapter 5.
6.5 Conclusion

In this chapter four areas of enhancements offered by the SpecDB representation of the software specifications over other tools, were proposed. As an example from the literature, Chays et. al. AGENDA research [27,28,54,55,56,57] using SpecDB was chosen. Additional enhancements can be made using the SpecDB representation to other tools in the market and the literature using the machine interpretable specifications stored in SpecDB. Many researchers confirmed that having such a queryable formal specification that can be used in their tools will automate and enhance the process of test case generation and result verification [31, 34, 38]. As a future work, many of those proposals can be implemented using the SpecDB technique, to test systems using the proposed automated ideas published, based on the software's machine readable specifications. In Chapter 7 a comparison between different testing tools discussed in Chapter 1, and SpecDB, is given to show some of the limitations and strengths of each tool. Also a conclusion of all the dissertation contributions and recommend future work is summarized in Chapter 7.
In this dissertation the design of SpecDB, a database created to represent and host software specifications in a machine-readable format is introduced. This scalable database representation was shown to aid in the processes of both automated software code generation and automated software testing, based on the actual software specifications. One of the main contributions of the dissertation is the creation of SpecDB; its design is detailed in Chapter 2. SpecDB is the database that can hold the specifications required for unit testing database software. An algorithm was created to show how to translate formal specifications into the SpecDB representation, in Chapter 3. Some examples of automated testing applications benefiting from the machine-retable specifications represented in SpecDB were created and discussed in Chapters 4 and 5.

This chapter concludes with a summary of the research contributions and accomplishments. In Section 7.1 the contributions are summarized in points. In Section 7.2 a more general discussion of the research accomplishments is summarized together with some proposals for future work and research continuation ideas.
7.1 **A Comparison Between Testing Tools**

In this section six test case scenarios are described. With these scenarios, the ability of different tools to identify errors in the software being tested, is highlighted. The testing tools used in this comparison are AGENDA, SpecDB, RE testing tool, Rational, Mercury and Segue set of tools. In the next section the test cases and scenarios are describe first. Then a comparison follows denoting the strength and limitations of each of the tools mentioned above.

7.1.1 **Test Cases and Scenarios**

7.1.1.1 **Test Case 1**

The case study and implementation in Chapter 5 is also use for this test case. The salesman reports twice to his employer during a certain month. The first time he reports that he sold 8 cruise vacations, 12 airline tickets and 10 shore excursion packages. The second time he reports that he sold 12 cruise vacations, 8 airline tickets and 4 shore excursion packages. The employer enters the data as sent each time. Therefore the database will have two records in the sales table for that specific salesman for that specific month. At the end of the month the employer enters the employee name, and the required month and year, then hits the *Calculate Commission* Button to view the report on the screen. In this scenario, it is assumed that the code is all error free and matches the specifications. No errors should be identified.
7.1.1.2  Test Case 2

In test case 2 the same implementation as in test case 1 is used, changing only one line of code. Per the specifications, the data entered by the employer in salesman identification, month and year fields should not change, when the user clicks on the calculate commission button. The five other fields should display the required information. In test case 2, we will assume that the program in line 25 has the following statement:

\[ \text{25 : month := Number\_Cruises\_Sold;} \]

Instead of

\[ \text{25 : Total\_Cruises\_Sold := Number\_Cruises\_Sold;} \]

Accordingly, the testing tool should identify 2 errors, changing the month value, and not displaying the total cruises sold.

7.1.1.3  Test Case 3

In this test case, the same code used in test case 1 is adopted, however, the code in line 46 is changed. The original code in line 46 of the implementation was:

\[ \text{46 end;} \]

Instead, we will assume that the programmer put the following lines of code:

\[ \text{46 update sales} \]
\[ \text{47 set cruise = Total\_Sales} \]
\[ \text{48 where s\_salesman = salesman and s\_month = month and s\_year = year;} \]
\[ \text{49 end;} \]

As discussed in Chapter 5, this results in modifying the data in the database, and will generate different values for the same salesman for the same month, each time the test case is run. One Error of commission, resulting in two database modifications should
be identified as an error. If this case is run only once, the error will not be identified by any functional testing tool. If run several times on the same inputs, functional testing techniques might spot the difference in the output data each time.

7.1.1.4 Test Case 4

Again in test case 4, the implementation is the same as that used in test case 1, with the exception of displaying the output in another window. From the user's point of view, no output is generated when he clicks the button, although, the output is computed correctly, it is displayed in the wrong location in the GUI. Ten errors should be identified, since the five outputs are misplaced.

7.1.1.5 Test Case 5

In this test case, the same code used in test case 1 is adopted here too, changing only the code in line 46. The original code in line 46 of the implementation was:

46   end;

Instead, we will assume that the programmer put code that grants privileges to all users to enter sales information, a privilege that should only be given to the employer. The same test case can also include other database operations including, Revoke, Create and Drop(view, table, constraint, sequence, etc.), Roles, Access Sequence, Commit, Rollback and Savepoint. All functional testing tools will not identify this error, no matter how many times the test is run. Although this test case involves a database operation, yet AGENDA cannot test it, because it is a Grant operation.
7.1.1.6 Test Case 6

Test case 6 is similar to test case 5, but assuming that code is written before the end to export employee names, social security number, addresses and dates of birth to another remote location. No functional testing tools will identify this error even if the test is run many times. Also high volume testing techniques [77,79] cannot spot this kind of error. High volume testing is when the test case is executed many times, even if the result is correct, to identify whether the system will react differently over time, for example from a slight system clock difference that accumulates over time.

7.1.2 Comparing the Ability of Testing Tools to Identify Errors

In this section, a comparison is given between different testing tools to show how each reacts to the errors in each scenario of the test cases in Section 7.1.1. A tabulation of the results is presented in Table 7.1. The test cases selected in Section 7.1.1, were chosen to show only the differences between the tools. There are a lot of test cases that show that all or most of the tools react similarly. In Table 7.1 the instances where the tools react similarly are removed, and only the areas where they differ in capability are included. The functionality of each testing tool is observed and compared based on a variety of factors. Among the comparison criteria is whether the tool can test imperative code, i.e. performs Structural testing; also coverage of the test cases that can be handled by the tool and the percentage of code in the implementation that can be tested by the tool.
Table 7.1: A Comparison Between How Testing Tools/Techniques Handle Test Cases 1-6

<table>
<thead>
<tr>
<th>Test case #</th>
<th>Tool</th>
<th>Can run test</th>
<th>Can spot error</th>
<th>Spot errors (different inputs for same scenario)</th>
<th>Can automatically verify results</th>
<th>Isolates error sources</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SpecDB Based Tools</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>2</td>
<td>SpecDB Based Tools</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>3</td>
<td>SpecDB Based Tools</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>4</td>
<td>SpecDB Based Tools</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>5</td>
<td>SpecDB Based Tools</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>6</td>
<td>SpecDB Based Tools</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>1</td>
<td>AGENDA</td>
<td>×</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>AGENDA</td>
<td>×</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>AGENDA</td>
<td>✓</td>
<td>User</td>
<td>×</td>
<td>To an extent</td>
<td>×</td>
</tr>
<tr>
<td>4</td>
<td>AGENDA</td>
<td>×</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>5</td>
<td>AGENDA</td>
<td>×</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>6</td>
<td>AGENDA</td>
<td>×</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>WinRunner, Robot, SilkTest5</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td>2</td>
<td>WinRunner, Robot, SilkTest5</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td>3</td>
<td>WinRunner, Robot, SilkTest5</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td>4</td>
<td>WinRunner, Robot, SilkTest5</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td>5</td>
<td>WinRunner, Robot, SilkTest5</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td>6</td>
<td>WinRunner, Robot, SilkTest5</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td>1</td>
<td>QuickTest, Functional SilkTest5</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td>2</td>
<td>QuickTest, Functional SilkTest5</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td>3</td>
<td>QuickTest, Functional SilkTest5</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td>4</td>
<td>QuickTest, Functional SilkTest5</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td>5</td>
<td>QuickTest, Functional SilkTest5</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td>6</td>
<td>QuickTest, Functional SilkTest5</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td>1</td>
<td>LoadRunner, Performance Tester, Silk Performer</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>2</td>
<td>LoadRunner, Performance Tester, Silk Performer</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>3</td>
<td>LoadRunner, Performance Tester, Silk Performer</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>4</td>
<td>LoadRunner, Performance Tester, Silk Performer</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>5</td>
<td>LoadRunner, Performance Tester, Silk Performer</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>6</td>
<td>LoadRunner, Performance Tester, Silk Performer</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
</tbody>
</table>

As seen from Table 7.2, there is some functionality that can be performed by other tools, namely memory management, stress testing, etc, that cannot be performed SpecDB tools using the current design of SpecDB. However, as proved previously, the SpecDB
representation is readily scalable. The current design of SpecDB detailed in Chapter 2, is for the purpose of unit testing. However, SpecDB can be expanded easily, as previously shown in Chapter 4, to accommodate more specifications to test other parts of the software including security, memory usage, etc.

There are unique capabilities associated with SpecDB based tools that other testing tools cannot perform, as shown in Figure 7.1; namely the ability of the tools to automatically identify errors in execution if the inputs to a test case changed, and the ability to isolate the source of errors in the code. Functional testing tools like regression testing, stress testing, scenario testing, record and play back tools, need the testers to verify the result of the test case. Since the tools have no specifications, the testers need to identify the expected outputs of each test case. The latter tools can run the same test scenarios with different input and automatically verify the output, unless for each of those test cases the expected output is given to the tool.

For SpecDB based tools however, since the specifications are available, tools based on SpecDB can automatically generate the expected output and verify whether the test case results match the specifications. The dissertation research is concluded with this comparing of some of the mostly used testing tools with the capabilities of SpecDB based tools. In the next sections the contributions are summarized, and recommendations for future work on SpecDB and SpecDB based tools are suggested.
Table 7.2: A Comparison Between the Capabilities of Testing Tools and Techniques

<table>
<thead>
<tr>
<th>Capability Tool</th>
<th>Can automatically generate tests</th>
<th>Readily scalable</th>
<th>Can automatically verify results</th>
<th>Unit Coverage</th>
<th>Memory / performance testing</th>
<th>Stress testing</th>
<th>Tests DB operations</th>
<th>Structural testing</th>
<th>Functional testing</th>
<th>Handles complex operations</th>
<th>Can isolate error sources</th>
</tr>
</thead>
<tbody>
<tr>
<td>SpecDB Based Tools</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>100%</td>
<td>×</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>AGENDA</td>
<td>✓</td>
<td>×</td>
<td>To an extent</td>
<td>Only DB operations</td>
<td>×</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td>WinRunner Robot, SilkTest</td>
<td>✓ (only test scripts from user scenarios)</td>
<td>×</td>
<td>✓</td>
<td>Based on scenarios</td>
<td>×</td>
<td>×</td>
<td>✓ (DB operations only)</td>
<td>✓</td>
<td>×</td>
<td>observed only</td>
<td>×</td>
</tr>
<tr>
<td>QuickTest, Functional SilkTest5</td>
<td>✓ (only user scenario test scripts)</td>
<td>×</td>
<td>✓</td>
<td>Based on scenarios</td>
<td>×</td>
<td>×</td>
<td>considered functions</td>
<td>×</td>
<td>✓</td>
<td>observed only</td>
<td>×</td>
</tr>
<tr>
<td>LoadRunner Performance Tester, Silk Performer</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>observed only</td>
<td>×</td>
</tr>
<tr>
<td>Purify</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>✓</td>
<td>observed only</td>
<td>×</td>
</tr>
</tbody>
</table>
7.2 Research Accomplishments at a Glance

In this section the research contributions are outlined. The main contributions are in italics. In Section 7.3 the contributions are discussed in more detail.

In this dissertation, the following was accomplished:

1. *Software specifications are represented in a machine readable format.* This was detailed in Chapter 2, the design of SpecDB.

2. *A translation algorithm to transform formal specifications written in a formal language like Z, to the machine-readable SpecDB representation is created.* This was shown in Chapter 3, storing specifications in SpecDB.

3. Useful applications that make use of the data stored in SpecDB are developed.
   a. In Chapter 4 an *automated tool that generates constraints and business rules* to enforce them at the database level as opposed to the application level, for better reliability, was introduced.
   b. Also in Chapter 5 a *reverse engineering approach to testing database applications* was introduced. It uses specifications to intelligently test software.

4. Enhancements on published work in the field of testing database applications, were suggested using the techniques described in this research in Chapter 6; some enhancements on AGENDA, created by Chays et. al. were proposed.
7.3 Conclusion and Recommendations

In this dissertation, the design of SpecDB, a database to hold parts of a software’s specification is being investigated. Representing software specifications in this manner aids the process of automated testing and automated code generation. Through the use of machine-readable specifications represented in SpecDB, automated tools can become more intelligent in generating and testing programs, based on an understanding of what the software under construction is supposed to do, by design. The design of SpecDB, discussed in Chapter 2, is scalable, allowing for any enhancement and additions to accommodate more forms of specifications or business requirements, as they evolve in the future, or other parts of the software application, e.g. security or networking, etc. In this dissertation, the idea of a unified standard to represent portions of the software specifications necessary to aid in the process of unit testing was explained.

In Chapter 3, an algorithm is implemented to translate examples of formal specification statements to the SpecDB representation. In this chapter, it was shown how a formal software specification can be automatically represented in SpecDB. Additional future work is to implement this algorithm, and also create an automated intelligent software testing tool to test software represented by SpecDB.

As an application of SpecDB and to illustrate its purpose, an automated tool was developed to generate business rules and enforce them on the database level by automatically creating constrains and assertions using the data stored in SpecDB, in Chapter 4. the design of a new table in SpecDB was discussed, Constraints, and the design of the Table_Description table of
SpecDB was extended, and renamed Table_Fields_Description. Those modifications were made in order to allow for the representation of business rules, and to show how the main idea of SpecDB is scalable and easily modifiable to accommodate the representation of different specifications, proving that the design of SpecDB allows for future enhancement and additions to accommodate more types of requirements, as they evolve in the future. The automated tool introduced in Chapter 4 creates a script to create all the constraints and another to drop them. System administrators can choose to run the production system with fewer constraints or no constraints at all in order to enhance the speed of production systems. They can run the automatic constraint generating tool demonstrated in Chapter 4 to create and automatically test the validity of the data during certain times of the week to maintain the accuracy, completeness and integrity of the system. Through the use of SpecDB, automated tools can become more intelligent in generating and testing programs, based on an understanding of what the software under construction is supposed to do, by design.

The use of SpecDB in a reverse engineering testing tool is discussed in Chapter 5. The reverse engineering approach was used to log the actual execution of the program due to some input, extracting the specifications of the implemented code. Once the program actions have been logged, they are presented in simple explanations which non-technical testers and other members of the software development team can understand. The actual outcome can also be easily matched and automatically compared to the expected output based on the program specifications in SpecDB that determine what the expected program output should be based on the specified inputs. Finally, in Chapter 6, some enhancements to published
literature, using SpecDB was suggested; as an example AGENDA [27,28,54] was used to show the suggested enhancements.

The representation of specifications in SpecDB can be used as a completion of AGENDA, the work done by Chays et. al. in [27,28,54,55,56,57]. AGENDA uses the database schema as a specification of the relations between the different entities in the software, and to extract the constraints on the relations or the values of the attributes. AGENDA depends on the tester’s user scenarios to test a database system, as well as the testers input files describing attribute valid values and their respective distribution probabilities. It generates tests by parsing application code not the specifications and design of the software. AGENDA then generates test data to populate the database and use the populated database as the input state for the testing operations. The data generated by AGENDA are only valid data that satisfy the constraints imposed by the implemented schema.

The tester suggests inputs for test cases and finally AGENDA provides a log file of the output database state after each test operation has executed, providing the modifications that happened in the database and output variables as a result of the test execution. The tester then uses those log files to determine whether the database output state is correct given the database input state before the test was executed with the given input variable values. The authors repeatedly highlight the fact that the process of verifying the test results cannot be automated since there is no formal specification available (i.e. available for automated to tool to interpret and reason with).
SpecDB is the machine-readable specifications that can be used with tools like AGENDA to automate the process of verifying the output state of a database after a test has been executed. In production level databases when there are thousands and maybe millions of records, it will be difficult and time consuming for the testers to verify the output and determine whether or not a test failed. SpecDB can also be used as an input to systems like AGENDA, instead of testers log files. When relying on testers to supply values for attributes or values for test inputs, are leaving the tester with the task of attaining good system coverage with the values he/she provides. Automating this process too, will provide the coverage needed without any coverage gaps.

Given SpecDB as one of the inputs to AGENDA, better data can be generated to populate the database. Also better test cases can be generated, with better system coverage. As another future work suggestion, the specifications stored in SpecDB can be used to automate the process of mathematically inducing rules to determine the validity of the output state after a test has been performed, and thus fully automate the process of output validation.

7.4 Future Work at a Glance

From the above description of the contributions introduced in this dissertation, and the ideas to continue on the work summarized in Section 7.2, the following future work ideas are proposed:

1. Expand on the design of SpecDB to allow for testing other areas of the software, including system security, networking, etc. An example of this was presented in Chapter 4, to show the scalability of the SpecDB representation.
2. Expand on the translation algorithm using the same idea in Chapter 3 to include other forms of the formal specifications syntax, as needed for the testing process.

3. Implement the algorithm in Chapter 3, to automatically read formal specifications and translate them to the SpecDB representation. This is a simple coding task that requires no further contribution.

4. Create a code generating tool that automatically implements software from the specifications in SpecDB. This is mostly a coding task, and requires a good understanding of the design of SpecDB and where and how specifications are represented and the interactions between the different entities of SpecDB.

5. Create other intelligent automated testing tools that benefit from the machine-readable software specifications in SpecDB and automatically reason with the specifications and create test cases based on an understanding of the specifications. This task requires the deployment of some artificial intelligence techniques to automatically generate intelligent and useful test cases to test software based on its specifications detailed in SpecDB. The test cases generated should attain good system coverage based on a white box system coverage matrix, maybe as part of the reverse engineering tool explained in Chapter 5. The result is a very useful intelligent automated testing tool that crowns the work established in this research. All the requirements to build this tool are detailed in this dissertation. A lot of simple test cases can be directly generated from the basic data in SpecDB, including
boundary analysis test cases, and type restrictions, as seen in Chapter 4. However, since the SpecDB representation handles more complex specifications and business rules, the proposed automated testing tool for future work can create much more intelligent, important and useful test cases, reasoning with the specifications in SpecDB, as well as generic requirements test cases including memory and exception handling, etc.

6. Implement the automatic comparison tool, which compares actual system behavior reported by the reverse engineering tool in Chapter 5, to that specified in SpecDB and report system errors or any discrepancies.

7. Automate the process of output state validation by mathematically inducing rules to determine the validity of the output state after a test has been performed. For database applications with thousands of records in production databases, it is very hard to manually validate the output state of the database after an operation has been executed. Accordingly, the proposed automated database output state validation tool can make the decision whether the output state matches the post conditions specified.

8. Create a GUI – SpecDB populating tool to populate SpecDB with the software specifications. Instead of writing the formal specifications, or manually populating SpecDB with the specifications, the human language specifications can be fed into SpecDB using this GUI tool. The GUI tool can ask questions and give choices for answers. Based on the input, the specifications are stored in the correct location in
SpecDB. The GUI tool can also make sure that the specifications are complete, and non-contracting to one another, i.e. test the specifications.

All the proposed future work tasks outlined in this section can be completed following the contributions achieved in this research. The introduction of the machine-readable specifications of SpecDB, is certainly the most important step and the basis for creating the intelligent automated testing tools that can aid the testing task immensely, if not promise to fully automate the process of testing software applications. After the completion of the recommendations for future work in this dissertation, there is no doubt that, a fully automated intelligent test case generator and output validator tool, based on the contributions of this dissertation, will be available in the near future.
REFERENCES


168


64. Cabal, M and J. Tuya, Using an SQL coverage measurement for testing database applications. FSE, October 2004.


APPENDICES
Appendix A: SQL Scripts for SpecDB Entities

The following SQL provides the creation scripts for the entities of the SpecDB database

Prompt Creating Table 'Valid_Operands'
Create Table Valid_Operands
(Operator_Id Number(2) Not Null
,Operand1_Type Varchar2(10)
,Operand2_Type Varchar2(10)
,Operand3_Type Varchar2(10)
,Over_Loaded Varchar2(1) Default 'F' )/

Prompt Creating Table 'Designed_Sr_Post_Conditions'
Create Table Designed_Sr_Post_Conditions
(Sub_Routine_Name Varchar2(50) Not Null
,Post_Cond_Num Integer Not Null
,Source Varchar2(5)
,Post_Cond_Type Varchar2(10)
,From_Var_Id Integer
,From_Gui_Obj_Name Varchar2(30)
,From_Gui_Window Varchar2(30)
,From_Seq_Name Varchar2(20)
,From_Char_Const_Value Varchar2(1)
,From_Text_Const_Value Varchar2(100)
,From_Int_Const_Value Integer
,From_Real_Const_Value Real
,From_Bool_Const_Value Varchar2(1) Default 'T'
,From_Date_Const_Value Date
,From_Time_Const_Value Date
,To_Gui_Obj_Name Varchar2(30)
,To_Gui_Window Varchar2(30)
,To_Var_Id Integer
,To_File_Name Varchar2(100)
,To_File_Type Varchar2(10)
,To_File_Change Varchar2(10)
,To_File_New_Location Varchar2(100)
,To_Message_Text Varchar2(240)
,To_Report_Name Varchar2(100)
,To_Report_Destination Varchar2(10)
,To_Report_File_Name Varchar2(100)
,To_Inc_Seq_Name Varchar2(30)
,Run_Exe_Name Varchar2(100)
,Open_Window_Name Varchar2(40)
,Savepoint_Rollback_Name Varchar2(30)
,Db_Operation_Num Integer )/
Prompt Creating Table 'Actual_Input'
Create Table Actual_Input
(Ai_Ip# Number(7) Not Null
, Ai_Test_Case# Number(7) Not Null
, Ai_Category Varchar2(4)
, Ai_Db_Table Varchar2(20)
, Ai_Db_Field Varchar2(25)
, Ai_Db_Row# Integer
, Ai_Value Varchar2(240)
, Ai_Into_Var Varchar2(240)
, Ai_Gui_Window Varchar2(240)
, Ai_Gui_Component Varchar2(240)
, Ai_Type Varchar2(240)
, Ai_File_Name Varchar2(240) )/

Prompt Creating Table 'Var'
Create Table Var
(Var_Id Integer Not Null
, Var_Name Varchar2(15) Not Null
, Type Varchar2(10)
, Sub_Routine_Name Varchar2(30)
, Length Integer
, Is_Input Varchar2(1) Default 'T'
, Is_Output Varchar2(1) Default 'T'
, Is_Constant Varchar2(1) Default 'T'
, Is_Gui_Object Varchar2(1) Default 'T'
, Class_Name Varchar2(240)
, Object_Name Varchar2(240)
, Encapsulation Varchar2(10)
, Gui_Window Varchar2(240)
, Has_Lov_Restrictions Varchar2(1) Default 'T'
, Has_Table_Restrictions Varchar2(1) Default 'T'
, Has_Other_Restrictions Varchar2(1) Default 'T'
, Num_Of_Other_Restrictions Integer
, Physical_Logical Varchar2(1)
, Mandatory Varchar2(1) Default 'T'
, Default_Char_Value Varchar2(1)
, Default_Text_Value Varchar2(100)
, Default_Int_Value Integer
, Default_Real_Value Real
, Default_Bool_Value Varchar2(1) Default 'T'
, Default_Date_Value Date
, Default_Time_Value Date
Appendix A: (Continued)

,Current_Char_Value Varchar2(1)
,Current_Real_Value Real
,Current_Text_Value Varchar2(100)
,Current_Int_Value Integer
,Current_Bool_Value Varchar2(1) Default 'T'
,Current_Date_Value Date
,Current_Time_Value Date )/

Prompt Creating Table 'List_Of_Tables'
Create Table List_Of_Tables
  (Table_Name Varchar2(240)
   ,Composite_Primary_Key Varchar2(1) Default 'T'
   ,Order_To_Fill Number )/

Prompt Creating Table 'Types_Lov'
Create Table Types_Lov
  (Type_Name Varchar2(20) Not Null
   ,Lov_Id Number Not Null
   ,Int_Value Integer
   ,Char_Value Varchar2(1)
   ,Text_Value Varchar2(240)
   ,Real_Value Real
   ,Date_Value Date
   ,Time_Value Date
   ,Bool_Value Smallint Default 0 )/

Prompt Creating Table 'Assignment'
Create Table Assignment
  (Pos_Num Integer Not Null
   ,Assign_To_Var_Id Number(3) Not Null
   ,Rhs_Type Varchar2(5)
   ,Const_Type Varchar2(10)
   ,Cal_Num Integer
   ,Func_Name Varchar2(100)
   ,Instance_Num Integer
   ,Var_Id Number(3)
   ,Const_Int_Value Integer
   ,Const_Char_Value Varchar2(1)
   ,Const_Real_Value Real
   ,Const_Date_Value Date
   ,Const_Text_Value Varchar2(100)
   ,Const_Boolean_Value Varchar2(1) Default 'T'
   ,Const_Time_Value Date

176
Appendix A: (Continued)

```
,Sequence_Name Varchar2(30)) /

Prompt Creating Table 'Sub_Routine_Ip_Description'
Create Table Sub_Routine_Ip_Description
(Subroutine_Name Varchar2(30) Not Null
,Ip_Order Integer Not Null
,Type Varchar2(240) Not Null
,Sr_Var_Name Varchar2(240)
,Has_Lov_Restrictions Char
,Has_Table_Restrictions Char
,Has_Other_Restrictions Char ) /

Prompt Creating Table 'Predicate'
Create Table Predicate
(Pos_Num Number(7) Not Null
,Component_Num Number(7) Not Null
,Left_Operand_Type Varchar2(5) Default 'Comp'
,Left_Component_Num Integer
,Left_Var_Id Integer
,Left_Text_Const Varchar2(240)
,Left_Num_Const Float(12)
,Left_Table_Name Varchar2(240)
,Left_Table_Alias Varchar2(240)
,Left_Db_Field Varchar2(240)
,Left_Func_Name Varchar2(240)
,Left_Func_Instance_Num Integer
,In_Db_Oper_Num Integer
,Right_Operand_Type Varchar2(5) Default 'Comp'
,Right_Component_Num Integer
,Right_Var_Id Integer
,Right_Text_Const Varchar2(240)
,Right_Num_Const Float(12)
,Right_Func_Name Varchar2(240)
,Right_Func_Instance_Num Integer
,Right_Table_Name Varchar2(240)
,Right_Table_Alias Varchar2(240)
,Right_Db_Field Varchar2(240)
,Operator_Id Integer Not Null ) /

Prompt Creating Table 'Pre_Post_Conditions'
Create Table Pre_Post_Conditions
(Db_Operation_Id Number Not Null
,Condition_Num Number Not Null
```
Appendix A: (Continued)

Prompt Creating Table 'Actual_Output'
Create Table Actual_Output
(Ao_Op# Integer Not Null
, Ao_Test_Case# Integer Not Null
, Ao_Category Varchar2(240)
, Ao_Db_Table Varchar2(240)
, Ao_Change Varchar2(240)
, Ao_Field Varchar2(240)
, Ao Row# Varchar2(240)
, Ao_Value Varchar2(240)
, Ao_From_Var Varchar2(240)
, Ao_Window Varchar2(240)
, Ao_Component Varchar2(240)
, Ao_Type Varchar2(240)
, Ao_Name Varchar2(240)
, Ao_Text Varchar2(240)
, Ao_Destination Varchar2(240)
, Ao_User Varchar2(240)
, Ao_Role_Id Varchar2(240)
, Ao_Privilege_Id Varchar2(240)
, Ao_Database Varchar2(240)
, Ao_Savepoint Varchar2(240) )/

Prompt Creating Table 'Class_Relationships'
Create Table Class_Relationships
( Relationship_Id Number Not Null
, Class_Name1 Varchar2(240) Not Null
, Relationship Varchar2(30) Not Null
, Class_Name2 Varchar2(240) Not Null )/

Prompt Creating Table 'Input_Output_Definition'
Create Table Input_Output_Definition
(Pos_Num Integer Not Null
, Var_Id Number(3) Not Null
, Var_Sequence Integer Not Null
, Input_Output Varchar2(1)
, Source_Destination_Var_Id Integer
, Type Varchar2(240)
, Report_Name Varchar2(240)
, Destination_Device Varchar2(240)
, File_Name Varchar2(240)
, File_Type Varchar2(240)
Prompt Creating Table 'Triggers'
Create Table Triggers
(Trigger_Name Varchar2(30) Not Null
,Dataflow_Pos_Num Integer
,On_Gui_Object Varchar2(1) Default 'T'
,Gui_Object_Var_Id Integer
,Gui_Trigger_Event Varchar2(240)
,On_Table Varchar2(1) Default 'T'
,Before_After Varchar2(1)
,Table_Insert Varchar2(1) Default 'T'
,Table_Delete Varchar2(1) Default 'T'
,Table_Update Varchar2(1) Default 'T'
,Field_Name Varchar2(20)
,Table_Name Varchar2(30)
,For_Each_Row Varchar2(1) Default 'T'
,When_Condition_Num Integer )/

Prompt Creating Table 'Db_Operation_Details'
Create Table Db_Operation_Details
(Db_Operation_Num Integer Not Null
,Order_Num Integer Not Null
,Field_Type Varchar2(10)
,Field_Name Varchar2(30)
,Into_Var_Id Number
,From_Var_Id Integer
,Char_Value Varchar2(1)
,Text_Value Varchar2(100)
,Int_Value Integer
,Real_Value Real
,Bool_Value Varchar2(1) Default 'T'
,Date_Value Date
,Time_Value Date
,Whole_Table Varchar2(1) Default 'T'
,Order_By_Requested Varchar2(1) Default 'T'
,Order_Desc Varchar2(1) Default 'T'
,Order_By_First Varchar2(15)
,Order_By_Second Varchar2(15)
,Order_By_Third Varchar2(15)
,Order_By_Fourth Varchar2(15)
,Insert_Select Varchar2(1) Default 'T'
Appendix A: (Continued)

```
,Insert_Select_Db_Oper_Num Number
,Delete_Where_Pos_Num Number )/

Prompt Creating Table 'Types'
Create Table Types
  (Type_Name Varchar2(20) Not Null
  ,Type Varchar2(10)
  ,Type_Size Integer
  ,Has_Lov Varchar2(1) Default 'T'
  ,Has_Range Varchar2(1) Default 'T'
  ,Even_Odd Varchar2(1)
  ,Range_From_Int Integer
  ,Range_To_Int Integer
  ,Range_From_Real Real
  ,Range_To_Real Real
  ,Range_From_Char Char
  ,Range_To_Char Char
  ,Range_From_Date Date
  ,Range_To_Date Date )/

Prompt Creating Table 'Expected_Output'
Create Table Expected_Output
  (Eo_Op# Integer Not Null
  ,Eo_Test_Case# Integer Not Null
  ,Eo_Category Varchar2(240)
  ,Eo_Db_Table Varchar2(240)
  ,Eo_Change Varchar2(240)
  ,Eo_Field Varchar2(240)
  ,Eo_Row# Varchar2(240)
  ,Eo_Value Varchar2(240)
  ,Eo_From_Var Varchar2(240)
  ,Eo_Window Varchar2(240)
  ,Eo_Component Varchar2(240)
  ,Eo_Type Varchar2(240)
  ,Eo_Name Varchar2(240)
  ,Eo_Text Varchar2(240)
  ,Eo_Destination Varchar2(240)
  ,Eo_User Varchar2(240)
  ,Eo_Role_Id Varchar2(240)
  ,Eo_Privilege_Id Varchar2(240)
  ,Eo_Database Varchar2(240)
  ,Eo_Savepoint Varchar2(240) )/
```
Appendix A: (Continued)

Prompt Creating Table 'Db_Operations'
Create Table Db_Operations
(Db_Operation_Num Integer Not Null
,Operation_Type Varchar2(6)
,Db_Name Varchar2(30)
,Db_Condition_Num Integer
,Sequence_Name Varchar2(240)
,Sequence_Incremented Varchar2(1) Default 'T'
,Sequence_Start_With Number
,Savepoint_Rollback_Name Varchar2(30)
,Role_Priv_Name Varchar2(6)
,Role_Priv_User Varchar2(20)
,Priv_Object_Name Varchar2(240)
,Grant_Admin_Option Varchar2(1) Default 'T'
,View_Name Varchar2(240)
,Table_Name Varchar2(240)
,As_Table_Name Varchar2(240)
,As_Select Varchar2(1) Default 'T'
,Cursor_Name Varchar2(240) )/

Prompt Creating Table 'Db_Test_Operation'
Create Table Db_Test_Operation
(Operation_Id Number Not Null )/

Prompt Creating Table 'Test_Templates'
Create Table Test_Templates
(Db_Operation_Id Number Not Null
,Test_Template_Name Varchar2(30) Not Null
)/

Prompt Creating Table 'Type_Restrictions'
Create Table Type_Restrictions
(Type_Name Varchar2(20) Not Null
,Restriction# Integer Not Null
,Include_Exclude Varchar(1)
,Has_Range Char
,Even_Or_Odd Varchar2(1)
,Range_From_Int Integer
,Range_To_Int Integer
,Int_Value Integer
,Range_From_Char Char
,Range_To_Char Char
,Char_Value Char
Appendix A: (Continued)

,Range_From_Real Real
,Range_To_Real Real
,Real_Value Real
,Range_From_Date Date
,Range_To_Date Date
,Date_Value Date
,Range_From_Time Date
,Range_To_Time Varchar2(240)
,Time_Value Date
,Text_Value Varchar2(240)
)/

Prompt Creating Table 'Attributes_Changed'
Create Table Attributes_Changed
(DB_Operation_Id Number Not Null
,Attribute Varchar2(30)
,Of_Table Varchar2(40)
)/

Prompt Creating Table 'Type_Excluded_Ranges'
Create Table Type_Excluded_Ranges
(Type_Name Varchar2(20) Not Null
,Restriction_Num Integer Not Null
,Has_Range Char
,Range_From_Int Integer
,Range_To_Int Integer
,Int_Value Integer
,Range_From_Char Char
,Range_To_Char Char
,Char_Value Char
,Range_From_Real Real
,Range_To_Real Real
,Real_Value Real
,Range_From_Date Date
,Range_To_Date Date
,Date_Value Date
,Range_From_Time Date
,Range_To_Time Varchar2(240)
,Time_Value Date
,Text_Value Varchar2(240)
)/
Appendix A: (Continued)

Prompt Creating Table 'Sub_Routine_Ip_Instance'
Create Table Sub_Routine_Ip_Instance
(Instance_Num Integer Not Null
,Ip_Order Integer Not Null
,Subroutine_Name Varchar2(30) Not Null
,Source_Type Varchar2(5)
,From_Var_Id Integer
,From_Gui_Object_Name Varchar2(50)
,From_Gui_Window Varchar2(50)
,From_Seq_Name Varchar2(30)
,From_Char_Const_Value Varchar2(1)
,From_Text_Const_Value Varchar2(100)
,From_Int_Const_Value Integer
,From_Real_Const_Value Real
,From_Bool_Const_Value Varchar2(1) Default 'T'
,From_Date_Const_Value Date
,From_Time_Const_Value Date
,Object_Name Varchar2(240) )/

Prompt Creating Table 'Function_Declaration'
Create Table Function_Declaration
(Func_Name Varchar2(240) Not Null
,Num_Of_Inputs Integer
,Op_Type Varchar2(240) Not Null
,Op_Var_Id Integer
,Op_Var_Name Varchar2(30)
,Op_Has_Lov_Restrictions Varchar2(1) Default 'T'
,Op_Has_Table_Restictions Varchar2(1) Default 'T'
,Op_Has_Other_Restrictions Varchar2(1) Default 'T'
,Changes_System_State Varchar2(1) Default 'T'
,Class_Name Varchar2(240)
,Encapsulation Varchar2(10) )/

Prompt Creating Table 'Var_Restrictions'
Create Table Var_Restrictions
(Var_Id Integer
,Restriction_Num Integer Not Null
,Sub_Routine_Name Varchar2(50)
,Var_Name Varchar2(30)
,Input_Or_Output Char
,Include_Exclude Varchar(1)
,Has_Range Char
,Even_Or_Odd Varchar2(1)
Appendix A: (Continued)

,Equivalence_Class Varchar2(240)
,Range_From_Int Integer
,Range_To_Int Integer
,Int_Value Integer
,Range_From_Char Char
,Range_To_Char Char
,Char_Value Char
,Range_From_Real Real
,Range_To_Real Real
,Real_Value Real
,Range_From_Date Date
,Range_To_Date Date
,Date_Value Date
,Range_From_Time Date
,Range_To_Time Varchar2(240)
,Time_Value Date
,Text_Value Varchar2(240)
)/

Prompt Creating Table 'Host_Var'
Create Table Host_Var
(Db_Operation_Id Number Not Null
,Host_Var_Name Varchar2(30) Not Null
,Input_Output Varchar2(1)
,Direct_Indirect Varchar2(1)
,On_Attribute Varchar2(30)
,Of_Table Varchar2(40)
)/

Prompt Creating Table 'Objects'
Create Table Objects
(Object_Name Varchar2(240) Not Null
,Class_Name Varchar2(240) Not Null
)/

Prompt Creating Table 'Data_Group'
Create Table Data_Group
(Db_Operation_Id Number Not Null
,Data_Group_Name Varchar2(240) Not Null
,Attribute Varchar2(30) Not Null
,Of_Table Varchar2(40)
,Probability Number
)
Appendix A: (Continued)

Prompt Creating Table 'Var_Table_Restrictions'
Create Table Var_Table_Restrictions
(Var_Tab_Rst_Id Integer Not Null
 ,Var_Id Integer Not Null
 ,Sub_Routine_Name Varchar2(240)
 ,Var_Name Varchar2(30)
 ,Input_Or_Output Varchar2(1)
 ,Operator Varchar2(240) Not Null
 ,Which_Table Varchar2(240) Not Null
 ,Which_Field Varchar2(240)
 ,Equivalence_Class Varchar2(240) )/

Prompt Creating Table 'Operator'
Create Table Operator
(Operator_Id Number(2) Not Null
 ,Symbol_Syntax Varchar2(5) Not Null
 ,Over_Loaded Varchar2(1) Default 'F' )/

Prompt Creating Table 'Constants'
Create Table Constants
(Constant_Name Varchar2(240)
 ,Type Varchar2(240)
 ,Int_Value Integer
 ,Real_Value Real
 ,Char_Value Char
 ,String_Value Varchar2(240)
 ,Date_Value Date
 ,Time_Value Date )/

Prompt Creating Table 'Expected_Input'
Create Table Expected_Input
(Ei_Ip# Number(7) Not Null
 ,Ei_Test_Case# Number(7) Not Null
 ,Ei_Category Varchar2(4)
 ,Ei_Db_Table Varchar2(20)
 ,Ei_Db_Field Varchar2(25)
 ,Ei_Db_Row# Integer
 ,Ei_Value Varchar2(240)
 ,Ei_Into_Var Varchar2(240)
 ,Ei_Gui_Window Varchar2(240)
 ,Ei_Gui_Component Varchar2(240)
 ,Ei_Type Varchar2(240)
 ,Ei_File_Name Varchar2(240) )/
Appendix A: (Continued)

Prompt Creating Table 'Char_Equiv_Class'
Create Table Char_Equiv_Class
   (Var_Name Varchar2(15) Not Null
    ,Range_From Varchar2(1) Not Null
    ,Range_To Varchar2(1) Not Null
    ,Equivalence_Class Integer Not Null
   )/

Prompt Creating Table 'Runtime_Sr_Post_Conditions'
Create Table Runtime_Sr_Post_Conditions
   (Sub_Routine_Name Varchar2(50) Not Null
    ,Post_Cond_Num Integer Not Null
    ,Instance_Num Integer Not Null
    ,Source Varchar2(5)
    ,Post_Cond_Type Varchar2(10)
    ,From_Var_Id Integer
    ,From_Gui_Obj_Name Varchar2(30)
    ,From_Gui_Window Varchar2(30)
    ,From_Seq_Name Varchar2(20)
    ,From_Char_Const_Value Varchar2(1)
    ,From_Text_Const_Value Varchar2(100)
    ,From_Int_Const_Value Integer
    ,From_Real_Const_Value Real
    ,From_Bool_Const_Value Varchar2(1) Default 'T'
    ,From_Date_Const_Value Date
    ,From_Time_Const_Value Date
    ,To_Gui_Obj_Name Varchar2(30)
    ,To_Gui_Window Varchar2(30)
    ,To_Var_Id Integer
    ,To_File_Name Varchar2(100)
    ,To_File_Type Varchar2(10)
    ,To_File_Change Varchar2(10)
    ,To_File_New_Location Varchar2(100)
    ,To_Message_Text Varchar2(240)
    ,To_Report_Name Varchar2(100)
    ,To_Report_Destination Varchar2(10)
    ,To_Report_File_Name Varchar2(100)
    ,To_Inc_Seq_Name Varchar2(30)
    ,Run_Exe_Name Varchar2(100)
    ,Savepoint_Rollback_Name Varchar2(20)
    ,Open_Window_Name Varchar2(40)
    ,Db_Operation_Num Integer )/
Appendix A: (Continued)

Prompt Creating Table 'Equivalence_Class'
Create Table Equivalence_Class
( Db_Operation_Id Number Not Null
 , Data_Group_Name Varchar2(240) Not Null
 , Attribute_Name Varchar2(30)
 , Int_Value Integer
 , Real_Value Real
 , Char_Value Varchar2(1)
 , Text_Value Varchar2(100)
 , Date_Value Date
 )/

Prompt Creating Table 'Dataflow'
Create Table Dataflow
( Pos_Num Integer Not Null
 , Suc_Num Integer Not Null
 , Pos_Type Varchar2(9) Default 'Process'
 , Result Varchar2(1) Default 'T'
 , Procedure_Name Varchar2(240)
 , Instance_Num Integer
 , Open_Window_Name Varchar2(240)
 , Run_Exe_Name Varchar2(240)
 , Object_Name Varchar2(240)
 , Class_Name Varchar2(240)
 )/

Prompt Creating Table 'Real_Equiv_Class'
Create Table Real_Equiv_Class
( Var_Name Varchar2(15) Not Null
 , Range_From Real Not Null
 , Range_To Real Not Null
 , Equivalence_Class Integer Not Null
 )/

Prompt Creating Table 'Procedure'
Create Table Procedure
( Proc_Name Varchar2(30) Not Null
 , Num_Of_Inputs Integer
 , Changes_System_State Varchar2(1) Default 'T'
 , Class_Name Varchar2(240)
 , Encapsulation Varchar2(10)
 )/
Appendix A: (Continued)

Prompt Creating Table 'Db_Operation_Tables'
Create Table Db_Operation_Tables
   (Id Integer Not Null
   ,Db_Operation_Num Integer
   ,Db_Constraint_Num Number
   ,Table_Name Varchar2(30) Not Null
   ,Table_Alias Varchar2(240)
   )/

Prompt Creating Table 'Data_Group_Values'
Create Table Data_Group_Values
   (Db_Operation_Id Number Not Null
   ,Data_Group_Name Varchar2(240) Not Null
   ,Int_Value Integer
   ,Real_Value Real
   ,Char_Value Varchar2(1)
   ,Text_Value Varchar2(100)
   ,Date_Value Date
   )/

Prompt Creating Table 'Int_Equiv_Class'
Create Table Int_Equiv_Class
   (Var_Name Varchar2(15) Not Null
   ,Range_From Integer Not Null
   ,Range_To Integer Not Null
   ,Equivalence_Class Integer Not Null
   )/

Prompt Creating Table 'Var_Lov'
Create Table Var_Lov
   (Id Number Not Null
   ,Var_Id Integer
   ,Var_Name Varchar2(30)
   ,Sub_Routine_Name Varchar2(240)
   ,Input_Or_Output Char
   ,Equivalence_Class Varchar2(240)
   ,Char_Value Varchar2(240)
   ,Text_Value Varchar2(100)
   ,Int_Value Integer
   ,Real_Value Real
   ,Date_Value Date
   ,Time_Value Date
   )/
Appendix A: (Continued)

Prompt Creating Table 'Constraints'
Create Table Constraints
(Constraint_Name Number Not Null
,Constraint_Num Integer Not Null
,Constraint_Or_Assertion Varchar2(1)
,On_Field Varchar2(1) Default 'T'
,Field_Name Varchar2(240)
,Table_Name Varchar2(240)
,Db_Name Varchar2(240)
,Condition_Predicate_Pos_Num Integer
,Constraint_Clause_Pred_Num Number
,Db_Op_Tables_Id Integer
,Db_Operation_Num Integer
,Db_Operator_Id Number(2)
)/

Prompt Creating Table 'Calculation'
Create Table Calculation
(Cal_Num Integer Not Null
,Component_Num Integer Not Null
,Left_Operand_Type Varchar2(5) Default 'Comp' Not Null
,Left_Component_Num Integer
,Left_Var_Id Integer
,Left_Text_Const Varchar2(240)
,Left_Num_Const Float(12)
,Left_Func_Name Varchar2(50)
,Left_Func_Instance_Num Integer
,Right_Operand_Type Varchar2(5) Default 'Comp' Not Null
,Right_Component_Num Integer
,Right_Var_Id Integer
,Right_Text_Const Varchar2(240)
,Right_Num_Const Float(12)
,Right_Func_Name Varchar2(50)
,Right_Func_Instance_Num Integer
,Operator_Id Integer Not Null
)/

Prompt Creating Table 'Classes'
Create Table Classes
(Class_Name Varchar2(240) Not Null
)/
Prompt Creating Table 'Table_Field_Description'
Create Table Table_Field_Description
 (Has_Constraint_Restriction Varchar2(1) Default 'T'
 ,Table_Name Varchar2(240) Not Null
 ,Field_Name Varchar2(240) Not Null
 ,Field_Id Varchar2(240)
 ,Field_Type Varchar2(240)
 ,Is_Primary_Key Varchar2(1) Default 'T'
 ,Is_Unique Varchar2(1) Default 'T'
 ,Mandatory Varchar2(1) Default 'T'
 ,Length Integer
 ,Has_Lov_Restriction Varchar2(1) Default 'T'
 ,Has_Table_Restriction Varchar2(1) Default 'T'
 ,Has_Range_Restriction Varchar2(1) Default 'T'
 ,Num_Of_Constraint_Restriction Integer
 ,Which_Table Varchar2(240)
 ,Which_Field Varchar2(240)
 )/
Appendix B: SQL Scripts for SpecDB Constraints

Prompt Creating Primary Key On 'Designed_Sr_Post_Conditions'
Alter Table Designed_Sr_Post_Conditions
Add (Constraint Post_Cond_Pk Primary Key
(Post_Cond_Num))
/

Prompt Creating Primary Key On 'Actual_Input'
Alter Table Actual_Input
Add (Constraint Ip_Pk Primary Key
(Ai_Ip#,Ai_Test_Case#))
/

Prompt Creating Primary Key On 'Var'
Alter Table Var
Add (Constraint Var_Pk Primary Key
(Var_Id))
/

Prompt Creating Primary Key On 'Types_Lov'
Alter Table Types_Lov
Add (Constraint Tlov_Pk Primary Key
(Lov_Id))
/

Prompt Creating Primary Key On 'Assignment'
Alter Table Assignment
Add (Constraint Fun_Pk Primary Key
(Pos_Num))
/

Prompt Creating Primary Key On 'Sub_Routine_Ip_Description'
Alter Table Sub_Routine_Ip_Description
Add (Constraint Sr_Id_Pk Primary Key
(Subroutine_Name,Ip_Order))
/

Prompt Creating Primary Key On 'Predicate'
Alter Table Predicate
Add (Constraint Pre_Pk Primary Key
(Pos_Num,Component_Num))
/
Appendix B: (Continued)

Prompt Creating Primary Key On 'Actual_Output'
Alter Table Actual_Output
Add (Constraint Actual_Op_Pk Primary Key
(Ao_Op#, Ao_Test_Case#))
/

Prompt Creating Primary Key On 'Class_Relationships'
Alter Table Class_Relationships
Add (Constraint Cls_Rel_Pk Primary Key
(Relationship_Id))
/

Prompt Creating Primary Key On 'Input_Output_Definition'
Alter Table Input_Output_Definition
Add (Constraint Odn_Pk Primary Key
(Var_Id, Pos_Num, Var_Sequence))
/

Prompt Creating Primary Key On 'Triggers'
Alter Table Triggers
Add (Constraint Trg_Pk Primary Key
(Trigger_Name))
/

Prompt Creating Primary Key On 'Db_Operation_Details'
Alter Table Db_Operation_Details
Add (Constraint Dit_Pk Primary Key
(Db_Operation_Num, Order_Num))
/

Prompt Creating Primary Key On 'Types'
Alter Table Types
Add (Constraint Type_Pk Primary Key
(Type_Name))
/

Prompt Creating Primary Key On 'Expected_Output'
Alter Table Expected_Output
Add (Constraint A_Op_Pk Primary Key
(Eo_Op#, Eo_Test_Case#))
/
Appendix B: (Continued)

Prompt Creating Primary Key On 'Db_Operations'
Alter Table Db_Operations
Add (Constraint Db_Oper_Pk Primary Key
(Db_Operation_Num))
/

Prompt Creating Primary Key On 'Db_Test_Operation'
Alter Table Db_Test_Operation
Add (Constraint Dbto_Pk Primary Key
(Operation_Id))
/

Prompt Creating Primary Key On 'Test_Templates'
Alter Table Test_Templates
Add (Constraint Tte_Pk Primary Key
(Db_Operation_Id
,Test_Template_Name))
/

Prompt Creating Primary Key On 'Type_Restrictions'
Alter Table Type_Restrictions
Add (Constraint Type_Res_Pk Primary Key
(Restriction#))
/

Prompt Creating Primary Key On 'Type_Excluded_Ranges'
Alter Table Type_Excluded_Ranges
Add (Constraint Type_Exran_Pk Primary Key
(Restriction_Num))
/

Prompt Creating Primary Key On 'Sub_Routine_Ip_Instance'
Alter Table Sub_Routine_Ip_Instance
Add (Constraint Sr_Ipin_Pk Primary Key
(Instance_Num
,Ip_Order))
/

Prompt Creating Primary Key On 'Function_Declaration'
Alter Table Function_Declaration
Add (Constraint Func_Dcl_Pk Primary Key
(Func_Name))
/
Appendix B: (Continued)

Prompt Creating Primary Key On 'Var_Restrictions'
Alter Table Var_Restrictions
Add (Constraint Tres_Pk Primary Key
(Restriction_Num))
/

Prompt Creating Primary Key On 'Host_Var'
Alter Table Host_Var
Add (Constraint Hvr_Pk Primary Key
(Db_Operation_Id
,Host_Var_Name))
/

Prompt Creating Primary Key On 'Objects'
Alter Table Objects
Add (Constraint Obj_Pk Primary Key
(Object_Name))
/

Prompt Creating Primary Key On 'Data_Group'
Alter Table Data_Group
Add (Constraint Dgp_Pk Primary Key
(Data_Group_Name
,Db_Operation_Id))
/

Prompt Creating Primary Key On 'Var_Table_Restrictions'
Alter Table Var_Table_Restrictions
Add (Constraint Fvtr_Pk Primary Key
(Var_Tab_Rst_Id))
/

Prompt Creating Primary Key On 'Operator'
Alter Table Operator
Add (Constraint Opr_Pk Primary Key
(Operator_Id))
/

Prompt Creating Primary Key On 'Expected_Input'
Alter Table Expected_Input
Add (Constraint Exop_Pk Primary Key
(Ei_Ip#, Ei_Test_Case#))
/
Appendix B: (Continued)

Prompt Creating Primary Key On 'Char_Equiv_Class'
Alter Table Char_Equiv_Class
Add (Constraint Ces_Pk Primary Key
(Var_Name
,Range_From
,Range_To))
/

Prompt Creating Primary Key On 'Runtime_Sr_Post_Conditions'
Alter Table Runtime_Sr_Post_Conditions
Add (Constraint Rt_Post_Cond_Pk Primary Key
(Post_Cond_Num
,Instance_Num))
/

Prompt Creating Primary Key On 'Dataflow'
Alter Table Dataflow
Add (Constraint Daw_Pk Primary Key
(Pos_Num ,Suc_Num))
/

Prompt Creating Primary Key On 'Real_Equiv_Class'
Alter Table Real_Equiv_Class
Add (Constraint Rec_Pk Primary Key
(Var_Name ,Range_From ,Range_To))
/

Prompt Creating Primary Key On 'Procedure'
Alter Table Procedure
Add (Constraint Proced_Pk Primary Key
(Proc_Name))
/

Prompt Creating Primary Key On 'Db_Operation_Tables'
Alter Table Db_Operation_Tables
Add (Constraint Otab_Pk Primary Key (Id))
/

Prompt Creating Primary Key On 'Int_Equiv_Class'
Alter Table Int_Equiv_Class
Add (Constraint Iec_Pk Primary Key
(Var_Name ,Range_From ,Range_To))
/
Prompt Creating Primary Key On 'Var_Lov'
Alter Table Var_Lov
  Add (Constraint Fvlov_Pk Primary Key
       (Id))
/

Prompt Creating Primary Key On 'Constraints'
Alter Table Constraints
  Add (Constraint Con_Pk Primary Key
       (Constraint_Num))
/

Prompt Creating Primary Key On 'Calculation'
Alter Table Calculation
  Add (Constraint Calc_Pk Primary Key
       (Cal_Num,
        Component_Num))
/

Prompt Creating Primary Key On 'Classes'
Alter Table Classes
  Add (Constraint Cls_Pk Primary Key
       (Class_Name))
/

Prompt Creating Primary Key On 'Table_Field_Description'
Alter Table Table_Field_Description
  Add (Constraint Tdesc_Pk Primary Key
       (Table_Name ,Field_Name))
/

Prompt Creating Unique Key On 'Sub_Routine_Ip_Description'
Alter Table Sub_Routine_Ip_Description
  Add (Constraint Sr_Ipdesc_Uk Unique
       (Subroutine_Name ,Sr_Var_Name))
/

Prompt Creating Unique Key On 'Sub_Routine_Ip_Instance'
Alter Table Sub_Routine_Ip_Instance
  Add (Constraint Sr_Ipinst_Uk Unique
       (Subroutine_Name))
/
Appendix B: (Continued)

Prompt Creating Unique Key On 'Table_Field_Description'
Alter Table Table_Field_Description
  Add (Constraint Field_Id_Uk Unique
       (Field_Id))
/

Prompt Creating Check Constraint On 'Valid_Operands'
Alter Table Valid_Operands
  Add (Constraint Avcon_1144240102_Oper_000 Check (Operand1_Type
       In ('Char', 'Text', 'Int', 'Real', 'Bool', 'Date', 'Time', 'Db_Table')))  
/

Prompt Creating Check Constraint On 'Valid_Operands'
Alter Table Valid_Operands
  Add (Constraint Avcon_1144240102_Oper_001 Check (Operand2_Type
       In ('Char', 'Text', 'Int', 'Real', 'Bool', 'Date', 'Time', 'Db_Table')))  
/

Prompt Creating Check Constraint On 'Valid_Operands'
Alter Table Valid_Operands
  Add (Constraint Avcon_1144240102_Oper_002 Check (Operand3_Type
       In ('Char', 'Text', 'Int', 'Real', 'Bool', 'Date', 'Time', 'Db_Table')))  
/

Prompt Creating Check Constraint On 'Valid_Operands'
Alter Table Valid_Operands
  Add (Constraint Avcon_1144240102_Over__000 Check (Over_Loaded In ('T', 'F')))  
/

Prompt Creating Check Constraint On 'Designed_Sr_Post_Conditions'
Alter Table Designed_Sr_Post_Conditions
  Add (Constraint Avcon_1144240102_Sourc_000 Check (Source In ('Var', 'Const', 'Gui', 'Seq')))  
/

Prompt Creating Check Constraint On 'Designed_Sr_Post_Conditions'
Alter Table Designed_Sr_Post_Conditions
  Add (Constraint Avcon_1144240102_Post__000 Check (Post_Cond_Type
       In ('Gui', 'Db', 'File', 'Report', 'Seq', 'Message', 'Priv_Role', 'Global_Var', 'Commit', 'Savepoint', 'Rollback', 'Run_Exe', 'Open_Win', 'Terminate')))  
/
Appendix B: (Continued)

Prompt Creating Check Constraint On 'Designed Sr Post Conditions'
Alter Table Designed Sr Post Conditions
  Add (Constraint Avcon_1144240102_From_000 Check (From_Bool_Const_Value In ('T', 'F')))
 /

Prompt Creating Check Constraint On 'Designed Sr Post Conditions'
Alter Table Designed Sr Post Conditions
  Add (Constraint Avcon_1144240102_To_Fi_000 Check (To_File_Change In ('Create', 'Delete', 'Moidify', 'Relocate')))
 /

Prompt Creating Check Constraint On 'Designed Sr Post Conditions'
Alter Table Designed Sr Post Conditions
  Add (Constraint Avcon_1144240102_To_Re_000 Check (To_Report_Destination In ('Printer', 'Screen', 'File')))
 /

Prompt Creating Check Constraint On 'Actual_Input'
Alter Table Actual_Input
  Add (Constraint Avcon_1144240102_Ai_Ca_000 Check (Ai_Category In ('Db', 'Gui', 'File')))
 /

Prompt Creating Check Constraint On 'Var'
Alter Table Var
  Add (Constraint Avcon_1144240102_Is_In_000 Check (Is_Input In ('T', 'F')))
 /

Prompt Creating Check Constraint On 'Var'
Alter Table Var
  Add (Constraint Avcon_1144240102_Is_Ou_000 Check (Is_Output In ('T', 'F')))
 /

Prompt Creating Check Constraint On 'Var'
Alter Table Var
  Add (Constraint Avcon_1144240102_Is_Co_000 Check (Is_Constant In ('T', 'F')))
 /

Prompt Creating Check Constraint On 'Var'
Alter Table Var
  Add (Constraint Avcon_1144240102_Is_Gu_000 Check (Is_Gui_Object In ('T', 'F')))
 /
Appendix B: (Continued)

Prompt Creating Check Constraint On 'Var'
Alter Table Var
  Add (Constraint Avcon_1144240102_Encap_000 Check (Encapsulation In ('Private', 'Protected', 'Public')))
/

Prompt Creating Check Constraint On 'Var'
Alter Table Var
  Add (Constraint Avcon_1144240102_Has_L_000 Check (Has_Lov_Restrictions In ('T', 'F')))
/

Prompt Creating Check Constraint On 'Var'
Alter Table Var
  Add (Constraint Avcon_1144240102_Has_T_000 Check (Has_Table_Restrictions In ('T', 'F')))
/

Prompt Creating Check Constraint On 'Var'
Alter Table Var
  Add (Constraint Avcon_1144240102_Has_O_000 Check (Has_Other_Restrictions In ('T', 'F')))
/

Prompt Creating Check Constraint On 'Var'
Alter Table Var
  Add (Constraint Avcon_1144240102_Manda_000 Check (Mandatory In ('T', 'F')))
/

Prompt Creating Check Constraint On 'Var'
Alter Table Var
  Add (Constraint Avcon_1144240102_Defau_000 Check (Default_Bool_Value In ('T', 'F')))
/

Prompt Creating Check Constraint On 'Var'
Alter Table Var
  Add (Constraint Avcon_1144240102_Curre_000 Check (Current_Bool_Value In ('T', 'F')))
/

Prompt Creating Check Constraint On 'List_Of_Tables'
Alter Table List_Of_Tables
Appendix B: (Continued)

Add (Constraint Avcon_1144240102_Compo_000 Check (Composite_Primary_Key In ('T', 'F'))) /
Prompt Creating Check Constraint On 'Types_Lov'
Alter Table Types_Lov
  Add (Constraint Avcon_1144240102_Bool__000 Check (Bool_Value In (0, 1))) /

Prompt Creating Check Constraint On 'Assignment'
Alter Table Assignment
  Add (Constraint Avcon_1144240102_Rhs_T_000 Check (Rhs_Type In ('Calc', 'Func', 'Var', 'Const', 'Seq', 'Gui', 'Db'))) /

Prompt Creating Check Constraint On 'Assignment'
Alter Table Assignment
  Add (Constraint Avcon_1144240102_Const_000 Check (Const_Type In ('Char', 'Text', 'Int', 'Real', 'Bool', 'Date', 'Time', 'Db_Table'))) /

Prompt Creating Check Constraint On 'Assignment'
Alter Table Assignment
  Add (Constraint Avcon_1144240102_Const_001 Check (Const_Boolean_Value In ('T', 'F'))) /

Prompt Creating Check Constraint On 'Sub_Routine_Ip_Description'
Alter Table Sub_Routine_Ip_Description
  Add (Constraint Avcon_1144240102_Has_L_001 Check (Has_Lov_Restrictions In ('T', 'F'))) /

Prompt Creating Check Constraint On 'Sub_Routine_Ip_Description'
Alter Table Sub_Routine_Ip_Description
  Add (Constraint Avcon_1144240102_Has_T_001 Check (Has_Table_Restrictions In ('T', 'F'))) /

Prompt Creating Check Constraint On 'Sub_Routine_Ip_Description'
Alter Table Sub_Routine_Ip_Description
  Add (Constraint Avcon_1144240102_Has_O_001 Check (Has_Other_Restrictions In ('T', 'F'))) /
Appendix B: (Continued)

Prompt Creating Check Constraint On 'Predicate'
Alter Table Predicate
    Add (Constraint Avcon_1144240102_Left__000 Check (Left_Operand_Type
    In ('Comp', 'Var', 'Const', 'Func', 'Db_Table', 'Db_Field')))/

Prompt Creating Check Constraint On 'Predicate'
Alter Table Predicate
    Add (Constraint Avcon_1144240102_Right_000 Check (Right_Operand_Type
    In ('Comp', 'Var', 'Const', 'Func', 'Db_Table', 'Db_Field')))
/

Prompt Creating Check Constraint On 'Pre_Post_Conditions'
Alter Table Pre_Post_Conditions
    Add (Constraint Avcon_1144240102_Pre_P_000 Check (Pre_Post In ('Pre', 'Post')))
/

Prompt Creating Check Constraint On 'Actual_Output'
Alter Table Actual_Output
    Add (Constraint Avcon_1144240102_Ao_Ca_000 Check (Ao_Category
    In ('Db', 'Gui', 'File', 'Sequence', 'Message', 'Report', 'Variable'
    , 'Role', 'Privilege', 'Commit', 'Savepoint', 'Rollback')))
/

Prompt Creating Check Constraint On 'Class_Relationships'
Alter Table Class_Relationships
    Add (Constraint Avcon_1144240102_Relat_000 Check (Relationship
    In ('Inherits_From', 'Aggregate_Of', 'Associated_With', 'Uses',
    'Instantiates', 'Metaclass_Of')))
/

Prompt Creating Check Constraint On 'Triggers'
Alter Table Triggers
    Add (Constraint Avcon_1144240102_On_Gu_000 Check (On_Gui_Object In ('T', 'F')))
/

Prompt Creating Check Constraint On 'Triggers'
Alter Table Triggers
    Add (Constraint Avcon_1144240102_On_Ta_000 Check (On_Table In ('T', 'F')))
/

Prompt Creating Check Constraint On 'Triggers'
Alter Table Triggers

Appendix B: (Continued)

Add (Constraint Avcon_1144240102_Before_000 Check (Before_After In ('B', 'A')))
/

Prompt Creating Check Constraint On 'Triggers'
Alter Table Triggers
Add (Constraint Avcon_1144240102_Table_000 Check (Table_Insert In ('T', 'F')))
/

Prompt Creating Check Constraint On 'Triggers'
Alter Table Triggers
Add (Constraint Avcon_1144240102_Table_001 Check (Table_Delete In ('T', 'F')))
/

Prompt Creating Check Constraint On 'Triggers'
Alter Table Triggers
Add (Constraint Avcon_1144240102_Table_002 Check (Table_Update In ('T', 'F')))
/

Prompt Creating Check Constraint On 'Triggers'
Alter Table Triggers
Add (Constraint Avcon_1144240102_For_E_000 Check (For_Each_Row In ('T', 'F')))
/

Prompt Creating Check Constraint On 'Db_Operation_Details'
Alter Table Db_Operation_Details
Add (Constraint Avcon_1144240102_Field_000 Check (Field_Type In ('Char', 'Text', 'Int', 'Real', 'Bool', 'Date', 'Time', 'Db_Table')))
/

Prompt Creating Check Constraint On 'Db_Operation_Details'
Alter Table Db_Operation_Details
Add (Constraint Avcon_1144240102_Bool_001 Check (Bool_Value In ('T', 'F')))
/

Prompt Creating Check Constraint On 'Db_Operation_Details'
Alter Table Db_Operation_Details
Add (Constraint Avcon_1144240102_Whole_000 Check (Whole_Table In ('T', 'F')))
/

Prompt Creating Check Constraint On 'Db_Operation_Details'
Alter Table Db_Operation_Details
Add (Constraint Avcon_1144240102_Order_000 Check (Order_By_Requested In ('T', 'F')))/
Appendix B: (Continued)

Prompt Creating Check Constraint On 'Db_Operation_Details'
 Alter Table Db_Operation_Details
     Add (Constraint Avcon_1144240102_Order_001 Check (Order_Desc In ('T', 'F')))

Prompt Creating Check Constraint On 'Db_Operation_Details'
 Alter Table Db_Operation_Details
     Add (Constraint Avcon_1144240102_Inser_000 Check (Insert_Select In ('T', 'F')))

Prompt Creating Check Constraint On 'Types'
 Alter Table Types
     Add (Constraint Avcon_1144240102_Has_L_002 Check (Has_Lov In ('T', 'F')))

Prompt Creating Check Constraint On 'Types'
 Alter Table Types
     Add (Constraint Avcon_1144240102_Has_R_000 Check (Has_Range In ('T', 'F')))

Prompt Creating Check Constraint On 'Types'
 Alter Table Types
     Add (Constraint Avcon_1144240102_Even__000 Check (Even_Odd In ('E', 'O')))

Prompt Creating Check Constraint On 'Expected_Output'
 Alter Table Expected_Output
     Add (Constraint Avcon_1144240102_Eo_Ca_000 Check (Eo_Category
 In ('Db', 'Gui', 'File', 'Sequence', 'Message', 'Report', 'Variable'
 , 'Role', 'Privilege', 'Commit', 'Savepoint', 'Rollback')))

Prompt Creating Check Constraint On 'Db_Operations'
 Alter Table Db_Operations
     Add (Constraint Avcon_1144240102_Opera_003 Check (Operation_Type
 In ('Select *', 'Select', 'Insert', 'Update', 'Delete', 'Create View'
 , 'Create_Table', 'Drop_Table', 'Create_Seq', 'Access_Seq',
 'Create Role', 'Select_Count', 'Commit', 'Rollback', 'Savepoint',
 'Grant', 'Revoke', 'Cursor', 'Fetch')))

Appendix B: (Continued)

Prompt Creating Check Constraint On 'Db_Operations'
Alter Table Db_Operations
  Add (Constraint Avcon_1144240102_Sequ_000 Check (Sequence_Incremented In ('T', 'F')))/

Prompt Creating Check Constraint On 'Db_Operations'
Alter Table Db_Operations
  Add (Constraint Avcon_1144240102_Grante_000 Check (Grant_Admin_Option In ('Grant', 'Admin')))/

Prompt Creating Check Constraint On 'Db_Operations'
Alter Table Db_Operations
  Add (Constraint Avcon_1144240102_Asse_000 Check (As_Select In ('T', 'F')))/

Prompt Creating Check Constraint On 'Type_Restrictions'
Alter Table Type_Restrictions
  Add (Constraint Avcon_1144240102_Inclus_000 Check (Include_Exclude In ('I', 'E')))/

Prompt Creating Check Constraint On 'Type_Restrictions'
Alter Table Type_Restrictions
  Add (Constraint Avcon_1144240102_HasR_000 Check (Has_Range In ('T', 'F')))/

Prompt Creating Check Constraint On 'Type_Restrictions'
Alter Table Type_Restrictions
  Add (Constraint Avcon_1144240102_EvenO_001 Check (Even_Or_Odd In ('E', 'O')))/

Prompt Creating Check Constraint On 'Type_Excluded_Ranges'
Alter Table Type_Excluded_Ranges
  Add (Constraint Avcon_1144240102_HasR_002 Check (Has_Range In ('T', 'F')))/

Prompt Creating Check Constraint On 'Sub_Routine_Ip_Instance'
Alter Table Sub_Routine_Ip_Instance
  Add (Constraint Avcon_1144240102_Sourc_001 Check (Source_Type In ('Var', 'Const', 'Gui', 'Seq')))/

Prompt Creating Check Constraint On 'Sub_Routine_Ip_Instance'
Alter Table Sub_Routine_Ip_Instance
Appendix B: (Continued)

Add (Constraint Avcon_1144240102_From__001 Check (From_Bool_Const_Value In ('T', 'F')))  
/

Prompt Creating Check Constraint On 'Function_Declaration'  
Alter Table Function_Declaration  
Add (Constraint Avcon_1144240102_Op_Ha_000 Check (Op_Has_Lov_Restrictions In ('T', 'F')))  
/

Prompt Creating Check Constraint On 'Function_Declaration'  
Alter Table Function_Declaration  
Add (Constraint Avcon_1144240102_Op_Ha_001 Check (Op_Has_Table_Restrictions In ('T', 'F')))  
/

Prompt Creating Check Constraint On 'Function_Declaration'  
Alter Table Function_Declaration  
Add (Constraint Avcon_1144240102_Op_Ha_002 Check (Op_Has_Other_Restrictions In ('T', 'F')))  
/

Prompt Creating Check Constraint On 'Function_Declaration'  
Alter Table Function_Declaration  
Add (Constraint Avcon_1144240102_Chang_000 Check (Changes_System_State In ('T', 'F')))  
/

Prompt Creating Check Constraint On 'Var_Restrictions'  
Alter Table Var_Restrictions  
Add (Constraint Avcon_1144240102_Input_000 Check (Input_or_Output In ('I', 'O')))  
/

Prompt Creating Check Constraint On 'Var_Restrictions'  
Alter Table Var_Restrictions  
Add (Constraint Avcon_1144240102_Inclu_001 Check (Include_Exclude In ('I', 'E')))  
/

Prompt Creating Check Constraint On 'Var_Restrictions'
Appendix B: (Continued)

Appendix B: (Continued)

Alter Table Var_Restrictions
   Add (Constraint Avcon_1144240102_Has_R_003 Check (Has_Range In ('T', 'F')))
   /

Prompt Creating Check Constraint On 'Var_Restrictions'
Alter Table Var_Restrictions
   Add (Constraint Avcon_1144240102_Even__002 Check (Even_Or_Odd In ('E', 'O')))
   /

Prompt Creating Check Constraint On 'Host_Var'
Alter Table Host_Var
   Add (Constraint Avcon_1144240102_Input_001 Check (Input_Output In ('I', 'O')))
   /

Prompt Creating Check Constraint On 'Host_Var'
Alter Table Host_Var
   Add (Constraint Avcon_1144240102_Direc_000 Check (Direct_Indirect In ('D', 'I')))
   /

Prompt Creating Check Constraint On 'Var_Table_Restrictions'
Alter Table Var_Table_Restrictions
   Add (Constraint Avcon_1144240102_Input_002 Check (Input_Or_Output In ('I', 'O')))
   /

Prompt Creating Check Constraint On 'Operator'
Alter Table Operator
   Add (Constraint Avcon_1144240102_Over__001 Check (Over_Loaded In ('T', 'F')))
   /

Prompt Creating Check Constraint On 'Expected_Input'
Alter Table Expected_Input
   Add (Constraint Avcon_1144240102_Ei_Ca_000 Check (Ei_Category In ('Db', 'Gui', 'File')))
   /

Prompt Creating Check Constraint On 'Runtime_Sr_Post_Conditions'
Alter Table Runtime_Sr_Post_Conditions
   Add (Constraint Avcon_1144240102_Sourc_002 Check (Source In ('Var', 'Const', 'Gui', 'Seq')))
   /

Prompt Creating Check Constraint On 'Runtime_Sr_Post_Conditions'
Alter Table Runtime_Sr_Post_Conditions
   Add (Constraint Avcon_1144240102_Post__001 Check (Post_Cond_Type

206
Appendix B: (Continued)

In ('Gui', 'Db', 'File', 'Report', 'Seq', 'Message', 'Priv_Role',
'Global_Var', 'Commit', 'Savepoint', 'Rollback', 'Run_Exe', 'Open_Win',
'Terminate')))/

Prompt Creating Check Constraint On 'Runtime_Sr_Post_Conditions'
Alter Table Runtime_Sr_Post_Conditions
Add (Constraint Avcon_1144240102_From__002 Check (From_Bool(Const_Value In ('T', 'F')))
/

Prompt Creating Check Constraint On 'Runtime_Sr_Post_Conditions'
Alter Table Runtime_Sr_Post_Conditions
Add (Constraint Avcon_1144240102_To_Fi_001 Check (To_File_Change In ('Create',
'Delete', 'Moidify', 'Relocate')))
/

Prompt Creating Check Constraint On 'Runtime_Sr_Post_Conditions'
Alter Table Runtime_Sr_Post_Conditions
Add (Constraint Avcon_1144240102_To_Re_001 Check (To_Report_Destination In
('Printer', 'Screen', 'File')))
/

Prompt Creating Check Constraint On 'Dataflow'
Alter Table Dataflow
Add (Constraint Avcon_1144240102_Pos_T_000 Check (Pos_Type
In ('Initial', 'Begin', 'Input', 'Assign', 'Decision', 'Whileloop',
'Forloop', 'Run_Exe', 'Output', 'Call', 'Db', 'Open_Window',
'Construct_Object', 'Destruct_Object', 'Call_Object_Method', 'End',
'Endfor', 'End Loop', 'Terminal')))
/

Prompt Creating Check Constraint On 'Dataflow'
Alter Table Dataflow
Add (Constraint Avcon_1144240102_Resul_000 Check (Result In ('T', 'F')))
/

Prompt Creating Check Constraint On 'Procedure'
Alter Table Procedure
Add (Constraint Avcon_1144240102_Chang_001 Check (Changes_System_State In ('T',
'F')))/

Prompt Creating Check Constraint On 'Procedure'
Alter Table Procedure

207
Appendix B: (Continued)

Add (Constraint Avcon_1144240102_Encap_002 Check (Encapsulation In ('Private', 'Protected', 'Public')))
/

Prompt Creating Check Constraint On 'Var_Lov'
Alter Table Var_Lov
  Add (Constraint Avcon_1144240102_Input_003 Check (Input_Or_Output In ('I', 'O'))) /

Prompt Creating Check Constraint On 'Constraints'
Alter Table Constraints
  Add (Constraint Avcon_1144240102_Const_002 Check (Constraint_Or_Assertion In ('C', 'A'))) /

Prompt Creating Check Constraint On 'Constraints'
Alter Table Constraints
  Add (Constraint Avcon_1144240102_On_Fi_000 Check (On_Field In ('T', 'F'))) /

Prompt Creating Check Constraint On 'Calculation'
Alter Table Calculation
  Add (Constraint Avcon_1144240102_Left__001 Check (Left_Operand_Type In ('Comp', 'Var', 'Const', 'Func', 'Db_Table', 'Db_Field'))) /

Prompt Creating Check Constraint On 'Calculation'
Alter Table Calculation
  Add (Constraint Avcon_1144240102_Right_001 Check (Right_Operand_Type In ('Comp', 'Var', 'Const', 'Func', 'Db_Table', 'Db_Field'))) /

Prompt Creating Check Constraint On 'Table_Field_Description'
Alter Table Table_Field_Description
  Add (Constraint Avcon_1144240102_Has_C_000 Check (Has_Constraint_Restriction In ('T', 'F'))) /

Prompt Creating Check Constraint On 'Table_Field_Description'
Alter Table Table_Field_Description
  Add (Constraint Avcon_1144240102_Is_Pr_000 Check (Is_Primary_Key In ('T', 'F'))) /

Prompt Creating Check Constraint On 'Table_Field_Description'
Appendix B: (Continued)

Alter Table Table_Field_Description
  Add (Constraint Avcon_1144240102_Is_Un_000 Check (Is.Unique In ('T', 'F'))) /

Prompt Creating Check Constraint On 'Table_Field_Description'
Alter Table Table_Field_Description
  Add (Constraint Avcon_1144240102_Manda_001 Check (Mandatory In ('T', 'F'))) /

Prompt Creating Check Constraint On 'Table_Field_Description'
Alter Table Table_Field_Description
  Add (Constraint Avcon_1144240102_Has_L_003 Check (Has.Lov.Restriction In ('T', 'F'))) /

Prompt Creating Check Constraint On 'Table_Field_Description'
Alter Table Table_Field_Description
  Add (Constraint Avcon_1144240102_Has_T_002 Check (Has.Table.Restriction In ('T', 'F'))) /

Prompt Creating Check Constraint On 'Table_Field_Description'
Alter Table Table_Field_Description
  Add (Constraint Avcon_1144240102_Has_R_004 Check (Has.Range.Restriction In ('T', 'F'))) /

Prompt Creating Foreign Key On 'Valid_Operands'
Alter Table Valid_Operands Add (Constraint
  Oprnd_Opr_Fk Foreign Key
  (Operator_Id) References Operator
  (Operator_Id)) /

Prompt Creating Foreign Key On 'Designed_Sr_Post_Conditions'
Alter Table Designed_Sr_Post_Conditions Add (Constraint
  Post_Cond_Proc_Fk Foreign Key
  (Sub_Routine_Name) References Procedure
  (Proc_Name)) /

Prompt Creating Foreign Key On 'Designed_Sr_Post_Conditions'
Alter Table Designed_Sr_Post_Conditions Add (Constraint
Appendix B: (Continued)

Post_Cond_Fdel_Fk Foreign Key
(Sub_Routine_Name) References Function_Declaration
(Func_Name))
/

Prompt Creating Foreign Key On 'Var'
Alter Table Var Add (Constraint
Var_Cls_Fk Foreign Key
(Class_Name) References Classes
(Class_Name))
/

Prompt Creating Foreign Key On 'Var'
Alter Table Var Add (Constraint
Var_Odn_Fk Foreign Key
(Var_Id) References Input_Output_Definition
(Var_Id))
/

Prompt Creating Foreign Key On 'Var'
Alter Table Var Add (Constraint
Var_Obj_Fk Foreign Key
(Object_Name) References Objects
(Object_Name))
/

Prompt Creating Foreign Key On 'Types_Lov'
Alter Table Types_Lov Add (Constraint
Tlov_Tdesc_Fk Foreign Key
(Type_Name) References Table_Field_Description
(Field_Id))
/

Prompt Creating Foreign Key On 'Types_Lov'
Alter Table Types_Lov Add (Constraint
Tlov_Tps_3_Fk Foreign Key
(Type_Name) References Types
(Type_Name))
/

Prompt Creating Foreign Key On 'Assignment'
Alter Table Assignment Add (Constraint
Appendix B: (Continued)

Fun_Calc_Fk Foreign Key  
(Cal_Num) References Calculation  
(Cal_Num))
/

Prompt Creating Foreign Key On 'Assignment'
Alter Table Assignment Add (Constraint
Fun_Daw_Fk Foreign Key  
(Pos_Num) References Dataflow  
(Pos_Num))
/

Prompt Creating Foreign Key On 'Assignment'
Alter Table Assignment Add (Constraint  
Fun_Fdcl Foreign Key  
(Func_Name) References Function_Declaration  
(Func_Name))
/

Prompt Creating Foreign Key On 'Sub_Routine_Ip_Description'
Alter Table Sub_Routine_Ip_Description Add (Constraint
Srp_Proc_Fk Foreign Key  
(Subroutine_Name) References Procedure  
(Proc_Name))
/

Prompt Creating Foreign Key On 'Sub_Routine_Ip_Description'
Alter Table Sub_Routine_Ip_Description Add (Constraint
Finrest1_Fdcl_Fk Foreign Key  
(Subroutine_Name) References Function_Declaration  
(Func_Name))
/

Prompt Creating Foreign Key On 'Sub_Routine_Ip_Description'
Alter Table Sub_Routine_Ip_Description Add (Constraint
Srp_Tps_3_Fk Foreign Key  
(Type) References Types  
(Type_Name))
/

Prompt Creating Foreign Key On 'Predicate'
Alter Table Predicate Add (Constraint
Pre_Opr Foreign Key
Appendix B: (Continued)

(Operator_Id) References Operator
(Operator_Id))
/

Prompt Creating Foreign Key On 'Predicate'
Alter Table Predicate Add (Constraint
Right_Var_Fk Foreign Key
(Right_Var_Id) References Var
(Var_Id))
/

Prompt Creating Foreign Key On 'Predicate'
Alter Table Predicate Add (Constraint
Pre_Trg_Fk2 Foreign Key
(Pos_Num) References Triggers
(Trigger_Name))
/

Prompt Creating Foreign Key On 'Predicate'
Alter Table Predicate Add (Constraint
Pre_Daw_Fk Foreign Key
(Pos_Num) References Dataflow
(Pos_Num))
/

Prompt Creating Foreign Key On 'Predicate'
Alter Table Predicate Add (Constraint
Left_Comp_Fk Foreign Key
(Left_Component_Num ,Pos_Num) References Predicate
(Component_Num ,Pos_Num))
/

Prompt Creating Foreign Key On 'Predicate'
Alter Table Predicate Add (Constraint
Left_Var_Fk Foreign Key
(Left_Var_Id) References Var
(Var_Id))
/

Prompt Creating Foreign Key On 'Predicate'
Alter Table Predicate Add (Constraint
Appendix B: (Continued)

Right_Comp_Fk Foreign Key
(Right_Component_Num, Pos_Num) References Predicate
(Component_Num, Pos_Num)

Prompt Creating Foreign Key On 'Pre_Post_Conditions'
Alter Table Pre_Post_Conditions Add (Constraint
Ppn_Dbto_Fk Foreign Key
(Db_Operation_Id) References Db_Test_Operation
(Operation_Id))

Prompt Creating Foreign Key On 'Class_Relationships'
Alter Table Class_Relationships Add (Constraint
Crfk2 Foreign Key
(Class_Name2) References Classes
(Class_Name))

Prompt Creating Foreign Key On 'Class_Relationships'
Alter Table Class_Relationships Add (Constraint
Crfk1 Foreign Key
(Class_Name1) References Classes
(Class_Name))

Prompt Creating Foreign Key On 'Input_Output_Definition'
Alter Table Input_Output_Definition Add (Constraint
Odn_Daw_Fk Foreign Key
(Pos_Num) References Dataflow
(Pos_Num))

Prompt Creating Foreign Key On 'Db_Operation_Details'
Alter Table Db_Operation_Details Add (Constraint
Op_Detail_Fk Foreign Key
(Db_Operation_Num) References Db_Operations
(Db_Operation_Num))
Appendix B: (Continued)

Prompt Creating Foreign Key On 'Test_Templates'
Alter Table Test_Templates Add (Constraint
Tte_Dbto_Fk Foreign Key
(Db_Operation_Id) References Db_Test_Operation
(Operation_Id))
/

Prompt Creating Foreign Key On 'Type_Restrictions'
Alter Table Type_Restrictions Add (Constraint
Type_Res_Tdesc_Fk Foreign Key
(Type_Name) References Table_Field_Description
(Field_Id))
/

Prompt Creating Foreign Key On 'Type_Restrictions'
Alter Table Type_Restrictions Add (Constraint
Type_Res_Tps_Fk Foreign Key
(Type_Name) References Types-Old
(Type_Name))
/

Prompt Creating Foreign Key On 'Attributes_Changed'
Alter Table Attributes_Changed Add (Constraint
Ace_Dbto_Fk Foreign Key
(Db_Operation_Id) References Db_Test_Operation
(Operation_Id))
/

Prompt Creating Foreign Key On 'Type_Excluded_Ranges'
Alter Table Type_Excluded_Ranges Add (Constraint
Tres_1_Tps_3_Fk Foreign Key
(Type_Name) References Types
(Type_Name))
/

Prompt Creating Foreign Key On 'Type_Excluded_Ranges'
Alter Table Type_Excluded_Ranges Add (Constraint
Type_Exran_Tdesc_Fk Foreign Key
Appendix B: (Continued)

(Type_Name) References Table_Field_Description
(Field_Id))
/

Prompt Creating Foreign Key On 'Sub_Routine_Ip_Instance'
Alter Table Sub_Routine_Ip_Instance Add (Constraint
Srp_1_Obj_Fk Foreign Key
(Object_Name) References Objects (Object_Name))
/

Prompt Creating Foreign Key On 'Sub_Routine_Ip_Instance'
Alter Table Sub_Routine_Ip_Instance Add (Constraint
Sp_Fk Foreign Key
(Subroutine_Name
,Ip_Order) References Sub_Routine_Ip_Description
(Subroutine_Name ,Ip_Order))
/

Prompt Creating Foreign Key On 'Sub_Routine_Ip_Instance'
Alter Table Sub_Routine_Ip_Instance Add (Constraint
Srp_Proc1_Fk Foreign Key
(Subroutine_Name) References Procedure_1
(Proc_Name))
/

Prompt Creating Foreign Key On 'Sub_Routine_Ip_Instance'
Alter Table Sub_Routine_Ip_Instance Add (Constraint
Finrest_Fdcl_Fk Foreign Key
(Subroutine_Name) References Function_Declaration_1
(Func_Name))
/

Prompt Creating Foreign Key On 'Function_Declaration'
Alter Table Function_Declaration Add (Constraint
Fdcl_Cls_Fk Foreign Key
(Class_Name) References Classes
(Class_Name))
/

Prompt Creating Foreign Key On 'Function_Declaration'
Alter Table Function_Declaration Add (Constraint
Fdcl_Tps_3_Fk Foreign Key
(Op_Type) References Types (Type_Name))
Appendix B: (Continued)

Prompt Creating Foreign Key On 'Var_Restrictions'
Alter Table Var_Restrictions Add (Constraint
  Varres_Fdcl_Fk Foreign Key
  (Sub_Routine_Name) References Function_Declaration
  (Func_Name))
/

Prompt Creating Foreign Key On 'Var_Restrictions'
Alter Table Var_Restrictions Add (Constraint
  Varres_Var_Fk Foreign Key
  (Var_Id) References Var
  (Var_Id))
/

Prompt Creating Foreign Key On 'Var_Restrictions'
Alter Table Var_Restrictions Add (Constraint
  Var_Fip_Fk Foreign Key
  (Sub_Routine_Name
  ,Var_Name) References Sub_Routine_Ip_Description
  (Subroutine_Name
  ,Sr_Var_Name))
/

Prompt Creating Foreign Key On 'Host_Var'
Alter Table Host_Var Add (Constraint
  Hvr_Dbto_Fk Foreign Key
  (Db_Operation_Id) References Db_Test_Operation
  (Operation_Id))
/

Prompt Creating Foreign Key On 'Objects'
Alter Table Objects Add (Constraint
  Obj_Cls_Fk Foreign Key
  (Class_Name) References Classes
  (Class_Name))
/

Prompt Creating Foreign Key On 'Data_Group'
Alter Table Data_Group Add (Constraint
  Dgp_Dbto_Fk Foreign Key
  (Db_Operation_Id) References Db_Test_Operation
  (Operation_Id))
/
Appendix B: (Continued)

Prompt Creating Foreign Key On 'Var_Table_Restrictions'
Alter Table Var_Table_Restrictions Add (Constraint
  Vtr_Fip_Fk Foreign Key
  (Sub_Routine_Name
   ,Var_Name) References Sub_Routine_Ip_Description
  (Subroutine_Name
   ,Sr_Var_Name))
 /

Prompt Creating Foreign Key On 'Var_Table_Restrictions'
Alter Table Var_Table_Restrictions Add (Constraint
  Vtr_Fdcl Foreign Key
  (Sub_Routine_Name) References Function_Declaration
  (Func_Name))
 /

Prompt Creating Foreign Key On 'Var_Table_Restrictions'
Alter Table Var_Table_Restrictions Add (Constraint
  Vtr_Var_Fk Foreign Key
  (Var_Id) References Var
  (Var_Id))
 /

Prompt Creating Foreign Key On 'Runtime_Sr_Post_Conditions'
Alter Table Runtime_Sr_Post_Conditions Add (Constraint
  Post_Con_1_Fdcl_Fk Foreign Key
  (Sub_Routine_Name) References Function_Declaration
  (Func_Name))
 /

Prompt Creating Foreign Key On 'Runtime_Sr_Post_Conditions'
Alter Table Runtime_Sr_Post_Conditions Add (Constraint
  Post_Con_1_Proc_Fk Foreign Key
  (Sub_Routine_Name) References Procedure
  (Proc_Name))
 /

Prompt Creating Foreign Key On 'Equivalence_Class'
Alter Table Equivalence_Class Add (Constraint
  Ecs_Dgp_Fk Foreign Key
  (Db_Operation_Id
   ,Data_Group_Name) References Data_Group
  (Db(Operation_Id
   ,Data_Group_Name)))
 /
Appendix B: (Continued)

Prompt Creating Foreign Key On 'Dataflow'
Alter Table Dataflow Add (Constraint
Daw_Obj_Fk Foreign Key
(Object_Name) References Objects
(Object_Name))
/

Prompt Creating Foreign Key On 'Dataflow'
Alter Table Dataflow Add (Constraint
Daw_Cls_Fk Foreign Key
(Class_Name) References Classes
(Class_Name))
/

Prompt Creating Foreign Key On 'Procedure'
Alter Table Procedure Add (Constraint
Proc_Cls_Fk Foreign Key
(Class_Name) References Classes
(Class_Name))
/

Prompt Creating Foreign Key On 'Db_Operation_Tables'
Alter Table Db_Operation_Tables Add (Constraint
Otab_Con_Fk Foreign Key
(Db_Constraint_Num) References Constraints
(Constraint_Num))
/

Prompt Creating Foreign Key On 'Db_Operation_Tables'
Alter Table Db_Operation_Tables Add (Constraint
Otab_Db_Oper_Fk Foreign Key
(Db_Operation_Num) References Db_Operations
(Db_Operation_Num))
/

Prompt Creating Foreign Key On 'Data_Group_Values'
Alter Table Data_Group_Values Add (Constraint
Dgv_Dgp_Fk Foreign Key
(Db_Operation_Id
,Data_Group_Name) References Data_Group
(Db_Operation_Id
,Data_Group_Name))
/
Appendix B: (Continued)

Prompt Creating Foreign Key On 'Var_Lov'
Alter Table Var_Lov Add (Constraint
  Vlov_Fdcl Foreign Key
  (Var_Id ,Sub_Routine_Name) References Function_Declaration
  (Op_Var_Id ,Func_Name))
/

Prompt Creating Foreign Key On 'Var_Lov'
Alter Table Var_Lov Add (Constraint
  Vlov_Var_Fk Foreign Key   (Var_Id) References Var  (Var_Id))
/

Prompt Creating Foreign Key On 'Var_Lov'
Alter Table Var_Lov Add (Constraint
  Vlov_Srp_Fk Foreign Key
  (Sub_Routine_Name ,Var_Name) References Sub_Routine_Ip_Description
  (Subroutine_Name ,Sr_Var_Name))
/

Prompt Creating Foreign Key On 'Constraints'
Alter Table Constraints Add (Constraint
  Con_Db_Oper_Fk Foreign Key
  (Db_Operation_Num) References Db_Operations (Db_Operation_Num))
/

Prompt Creating Foreign Key On 'Constraints'
Alter Table Constraints Add (Constraint
  Con_Tdesc_Fk Foreign Key
  (Field_Name ,Table_Name) References Table_Field_Description (Field_Name
  ,Table_Name))
/

Prompt Creating Foreign Key On 'Constraints'
Alter Table Constraints Add (Constraint
  Con_Opr_Fk Foreign Key
  (Db_Operator_Id) References Operator (Operator_Id))
/

Prompt Creating Foreign Key On 'Calculation'
Alter Table Calculation Add (Constraint
  Right_Comp_Var_Fk2 Foreign Key
  (Right_Var_Id) References Var
  (Var_Id))
Appendix B: (Continued)

Prompt Creating Foreign Key On 'Calculation'
Alter Table Calculation Add (Constraint
Left_Comp_Var_Fk2 Foreign Key
(Left_Var_Id) References Var
(Var_Id))
/

Prompt Creating Foreign Key On 'Calculation'
Alter Table Calculation Add (Constraint
Left_Comp_Calc_Fk Foreign Key
(Left_Component_Num
,Cal_Num) References Calculation
(Component_Num ,Cal_Num))
/

Prompt Creating Foreign Key On 'Calculation'
Alter Table Calculation Add (Constraint
Right_Comp_Calc_Fk Foreign Key
(Right_Component_Num
,Cal_Num) References Calculation
(Component_Num ,Cal_Num))
/

Prompt Creating Foreign Key On 'Calculation'
Alter Table Calculation Add (Constraint
Calc_Opr_Fk Foreign Key
(Operator_Id) References Operator
(Operator_Id))
/

Prompt Creating Foreign Key On 'Calculation'
Alter Table Calculation Add (Constraint
Calc_Fdcl_Fk2 Foreign Key
(Right_Func_Name) References Function_Declaration
(Func_Name))
/

Prompt Creating Foreign Key On 'Calculation'
Alter Table Calculation Add (Constraint
Calc_Fdcl_Fk Foreign Key
(Left_Func_Name) References Function_Declaration
(Func_Name))
/
PROCEDURE "CONSTRAINT_LOADER" --is

(T_Table_Name    varchar2(240);
T_Field_ID    varchar2(240);
T_Field_Name   varchar2(240);
T_Field_Type    varchar2(240);
T_Primary_Key   varchar2(1);
T_Is_Unique    varchar2(1);
T_Mandatory    varchar2(1);
T_LENGTH     INT;
T_Has_LOV_Restrictions    varchar2(1);
T_Has_Table_Restrictions    varchar2(1);
T_Has_Range_Restrictions    varchar2(1);
T_Has_Constraint_Restrictions    varchar2(1);
T_Num_of_Constraint_Restrictions    INT;
T_Which_Table    varchar2(240);
T_Which_Field    varchar2(240);

F_Type_Name VARCHAR2(20);
F_Restriction_Num int;
F_Include_Exclude VARCHAR2(1); F_Has_Range char(1);
F_Even_Or_Odd VARCHAR2(1);
F_Range_From_Int int; F_Range_To_Int int; F_Int_Value int;
F_Range_From_Char char; F_Range_To_Char char; F_Char_Value char;
F_Range_From_Real real; F_Range_To_Real real; F_Real_Value real;
F_Range_From_Date date; F_Range_To_Date date; F_Date_Value date;
F_Range_From_Time date; F_Range_To_Time date; F_Time_Value date;
F_Text_Value    VARCHAR2(240);

ScriptFile UTL_FILE.FILE_TYPE;

LOV_Tab_Name varchar2(20);

included_ranges int; excluded int;
int1 int;
int2 int;
char1 char;
char2 char;
real1 real;
real2 real;
date1 date;
date2 date;
CURSOR  Field_Desc IS
Select *
From Table_Field_Description;

CURSOR  field_restrictions IS
Select *
From TYPE_EXCLUDED_RANGES
Where Type_Name = T_Field_ID;

BEGIN
-- A file should already be created, or else, an exception handler should be created
ScriptFile := UTL_FILE.FOPEN (Directory, File_Name, 'w');

OPEN Field_Desc;
LOOP
  Fetch Field_Desc Into T_Table_Name, T_Field_ID, T_Field_Name,
  T_Field_Type, T_Primary_Key, T_Is_Unique, T_Mandatory, T_LENGTH,
  T_Has_LOV_Restrictions, T_Has_Table_Restrictions, T_Has_Range_Restrictions,
  T_Has_Constraint_Restrictions, T_Num_of_Constraint_Restrictions, T_Which_Table,
  T_Which_Field;

  EXIT WHEN Field_Desc%NOTFOUND;
  IF (T_Primary_Key = 'T') OR (T_Primary_Key = 't')-- Primary Key
    THEN  UTL_FILE.PUTF(ScriptFile,'ALTER TABLE %s ADD CONSTRAINT %s_PK PRIMARY KEY (%s);
\n',T_Table_Name,T_Field_Name,T_Field_Name);
  END IF;
  IF (T_Is_Unique = 'T') OR (T_Is_Unique = 't')-- Unique Field Constraint
    THEN  UTL_FILE.PUTF(ScriptFile,'ALTER TABLE %s ADD CONSTRAINT %s_UK UNIQUE (%s);
\n',T_Table_Name,T_Field_Name,T_Field_Name);
  END IF;
  IF (T_Mandatory = 'M') OR (T_Mandatory = 'm')-- Not NULL
    THEN UTL_FILE.PUTF(ScriptFile,'ALTER TABLE %s MODIFY %s NOT NULL;\n\n',T_Table_Name, T_Field_Name);
  END IF;

END IF;
Appendix C: (Continued)

IF (T_Has_LOV_Restrictions = 'T') OR (T_Has_LOV_Restrictions = 't')-- Has a List of Valid Values
    THEN
        LOV_Tab_Name := T_Table_Name ||'_' || T_Field_Name || '_LOV';
        UTL_FILE.PUTFS(ScriptFile,'CREATE TABLE %s (%s VARCHAR2, valid_value %s);
', LOV_Tab_Name, T_Field_Name, T_Field_Type);
        IF T_Field_Type IN ('INTEGER','SMALLINT')
            THEN UTL_FILE.PUTFS(ScriptFile,'INSERT INTO %s
SELECT INT_Value FROM Types_LOV WHERE Type_Name = %s;
', LOV_Tab_Name, T_Field_ID);
        ELSIF T_Field_Type IN ('CHAR','VARCHAR2','LONG')
            THEN UTL_FILE.PUTFS(ScriptFile,'INSERT INTO %s
SELECT CHAR_Value FROM Types_LOV WHERE Type_Name = %s;
', LOV_Tab_Name, T_Field_ID);
        ELSIF T_Field_Type IN ('NUMBER','REAL')
            THEN UTL_FILE.PUTFS(ScriptFile,'INSERT INTO %s
SELECT REAL_Value FROM Types_LOV WHERE Type_Name = %s;
', LOV_Tab_Name, T_Field_ID);
        ELSIF T_Field_Type IN ('BOOLEAN')
            THEN UTL_FILE.PUTFS(ScriptFile,'INSERT INTO %s
SELECT BOOL_Value FROM Types_LOV WHERE Type_Name = %s;
', LOV_Tab_Name, T_Field_ID);
        END IF;
        UTL_FILE.PUTFS(ScriptFile,'ALTER TABLE %s ADD
CONSTRAINT %s_FK FOREIGN KEY(%s) REFERENCES %s(valid_value);
', T_Table_Name, T_Field_Name, T_Field_Name, LOV_Tab_Name);
    END IF;

    IF (T_Has_Table_Restrictions = 'T') OR (T_Has_Table_Restrictions = 't')-- References another field in the same or a different table
        THEN
            UTL_FILE.PUTFS(ScriptFile,'ALTER TABLE %s ADD
CONSTRAINT %s_FK FOREIGN KEY(%s) REFERENCES %s(%s);
', T_Table_Name, T_Field_Name, T_Field_Name, T_Which_Table, T_Which_Field);
        END IF;
    END IF;
Appendix C: (Continued)

IF (T_Has_Range_Restrictions = 'T') OR (T_Has_Range_Restrictions = 't')-- if there are other restrictions in the range, including certain numbers, or excluding others, etc.
    THEN
        --1. Look in TYPE, to define the range of valid values.

        IF T_Field_Type IN ('INTEGER', 'SMALLINT')
            THEN
                select min(range_from_int), max(range_to_int) into int1, int2
                from TYPES where type_name = T_Field_ID and upper (has_range) = 'T';
                UTL_FILE.PUTF(ScriptFile,'ALTER TABLE %s ADD CONSTRAINT %s_CK CHECK(%s BETWEEN %s AND %s); 
                T_Table_Name, T_Field_Name, T_Field_Name, int1, int2);
            ELSIF T_Field_Type = 'CHAR'
                THEN
                select min(range_from_char), max(range_to_char) into char1, char2
                from TypeS where type_name = T_Field_ID and upper (has_range) = 'T';
                UTL_FILE.PUTF(ScriptFile,'ALTER TABLE %s ADD CONSTRAINT %s_CK CHECK(%s BETWEEN %s AND %s); 
                T_Table_Name, T_Field_Name, T_Field_Name, char1, char2);
            ELSIF T_Field_Type IN ('NUMBER', 'REAL')
                THEN
                select min(range_from_REAL), max(range_to_REAL) into REAL1, REAL2
                from TypeS where type_name = T_Field_ID and upper (has_range) = 'T';
                UTL_FILE.PUTF(ScriptFile,'ALTER TABLE %s ADD CONSTRAINT %s_CK CHECK(%s BETWEEN %s AND %s); 
                T_Table_Name, T_Field_Name, T_Field_Name, REAL1, REAL2);
            ELSIF T_Field_Type = 'DATE'
                THEN
                select min(range_from_DATE), max(range_to_DATE) into DATE1, DATE2
                from TypeS where type_name = T_Field_ID and upper (has_range) = 'T';

                224
Appendix C: (Continued)

UTL_FILE.PUTF(ScriptFile,'ALTER TABLE %s ADD CONSTRAINT
%s_CK CHECK(%s BETWEEN %s AND %s); 
',
T_Table_Name, T_Field_Name,
T_Field_Name,DATE1, DATE2);
--               ELSIF T_Field_Type = 'TIME'
--                THEN
--                    select range_from_TIME, range_to_TIME into TIME1, TIME2
from Type_Restrictions where type_name = T_Field_ID;
--                    UTL_FILE.PUTF(ScriptFile,'ALTER TABLE %s ADD
CONSTRAINT %s_CK CHECK(%s BETWEEN %s AND %s); 
',
--                    T_Table_Name, T_Field_Name,
T_Field_Name,TIME1, TIME2);
END IF;

Select count(*) into excluded
From TYPE_EXCLUDED_RANGES
Where Type_Name = T_Field_ID ;

if excluded > 0 --there is at least one excluded range or value
then
   OPEN field_restrictions;
   LOOP
      Fetch field_restrictions Into F_Type_Name, F_Restriction_Num,
F_Has_Range, F_Range_From_Int,F_Range_To_Int,F_Int_Value,
F_Range_From_Char,F_Range_To_Char,F_Char_Value,F_Range_From_Real,F_Range_To_Real,
F_Real_Value,F_Range_From_Date,F_Range_To_Date,F_Date_Value,
F_Range_From_Time,
F_Range_To_Time,F_Time_Value,F_Text_Value;
      EXIT WHEN field_restrictions%NOTFOUND;

      IF T_Field_Type IN ('INTEGER','SMALLINT')
      THEN
         IF upper(F_Has_Range) = 'T'
            then UTL_FILE.PUTF(ScriptFile,'ALTER TABLE %s
ADD CONSTRAINT %s_CK CHECK(%s NOT BETWEEN %s AND %s); 
',
T_Table_Name, T_Field_Name,
Appendix C: (Continued)

T_Field_Name,F_Range_From_Int, F_Range_To_Int);
else UTL_FILE.PUTF(ScriptFile,'ALTER TABLE %s
ADD CONSTRAINT %s_CK CHECK(%s <> %s); 'n',
T_Table_Name, T_Field_Name,
T_Field_Name,F_Int_Value);
end if; --has range or single value

ELSIF T_Field_Type = 'CHAR'
THEN
  IF upper(F_Has_Range) = 'T'
    then UTL_FILE.PUTF(ScriptFile,'ALTER TABLE %s
ADD CONSTRAINT %s_CK CHECK(%s NOT BETWEEN %s AND %s); 'n',
T_Table_Name, T_Field_Name,
T_Field_Name,F_Range_From_Char, F_Range_To_Char);
  else UTL_FILE.PUTF(ScriptFile,'ALTER TABLE %s
ADD CONSTRAINT %s_CK CHECK(%s <> %s); 'n',
T_Table_Name, T_Field_Name,
T_Field_Name,F_Char_Value);
end if; --has range or single value

ELSIF T_Field_Type IN ('NUMBER','REAL')
THEN
  IF upper(F_Has_Range) = 'T'
    then UTL_FILE.PUTF(ScriptFile,'ALTER TABLE %s
ADD CONSTRAINT %s_CK CHECK(%s NOT BETWEEN %s AND %s); 'n',
T_Table_Name, T_Field_Name,
T_Field_Name,F_Range_From_Real, F_Range_To_Real);
  else UTL_FILE.PUTF(ScriptFile,'ALTER TABLE %s
ADD CONSTRAINT %s_CK CHECK(%s <> %s); 'n',
T_Table_Name, T_Field_Name,
T_Field_Name,F_Real_Value);
end if; --has range or single value

ELSIF T_Field_Type = 'DATE'
THEN
  IF upper(F_Has_Range) = 'T'
    then UTL_FILE.PUTF(ScriptFile,'ALTER TABLE %s
ADD CONSTRAINT %s_CK CHECK(%s NOT BETWEEN %s AND %s); 'n',
T_Table_Name, T_Field_Name,
T_Field_Name,F_Range_From_Date, F_Range_To_Date);
  else UTL_FILE.PUTF(ScriptFile,'ALTER TABLE %s
ADD CONSTRAINT %s_CK CHECK(%s <> %s); 'n',
T_Table_Name, T_Field_Name,
Appendix C: (Continued)

T_Field_Name,F_Date_Value);
end if; --has range or single value

END LOOP;
END IF; --there is at least excluded range or value

END IF;
END loop ;

UTL_FILE.FFLUSH(ScriptFile);
CLOSE Field_Desc;
COMMIT;
UTL_FILE.FCLOSE (ScriptFile);
END;
ABOUT THE AUTHOR

Rana Mikhail finished her Bachelor's Degree in Computer Science from the American University in Cairo (AUC) in 1998. She received her M.Sc. degree in Computer Science from AUC in 1999. She worked as a decision support systems consultant for two years and then started her Ph.D. in Computer Science in 2000. Her research interests include database systems, software testing, data mining, data warehousing and software engineering including system analysis and design.