A Project of System Level Requirement Specification Test Cases Generation

Submitted to
Dr. Jane Pavelich
Dr. Jeffrey Joyce

by

Cai, Kelvin (47663000)

for the course of

EECE 496

on

August 8, 2003

Abstract

A system of automating the process of system-level specification-base testing is implemented. There are five phases of the system: preprocess, parse tree construction, logical manipulation, symbol table construction, and output. In the preprocess phase, the system removes extra white spaces in the original document and ensure the uniqueness of ROID numbers. In the parse tree construction phase, lexical analysis and syntax analysis are performed to parse the document into an internal data structure called Parse Tree. In the logical manipulation phase, logical expressions are converted to DNF form according a set of logical equivalences laws. In symbol table construction phase, data of test cases is transformed and optimized from Parse Tree to another internal data structure named Symbol Table. In the output phase, the generated test cases are formatted and stored in HTML files. In addition to the main phases, the system also implemented a graphic user interface front-end for user interaction. Finally, the system is a success for generating test cases from system level requirement specification document.

Table of Contents

List of Figures

List of Tables

1. INTRODUCTION

1.1. Objective

This report describes a system of generating test cases from system level requirement specifications. The analysis of test derivation is currently done manually because it is lack of automated analysis tools for test case generation. Applying various techniques and algorithms, this system researches possible approaches and methodologies of automation for this task. As a result, quality of test cases is improved, time and effort for derivation and review is reduced, thereby reducing the overall cost of system-level specification-based testing.

1.2. Architecture

The system consists of five phases: preprocess, parse tree construction, logical manipulation, symbol table construction, and output. Two internal data structures, Parse Tree and Symbol Table, are constructed during the process. These data structures are used to pass information from phase to phase for test cases generation. Figure 1.1 illustrates the entire workflow of the system.

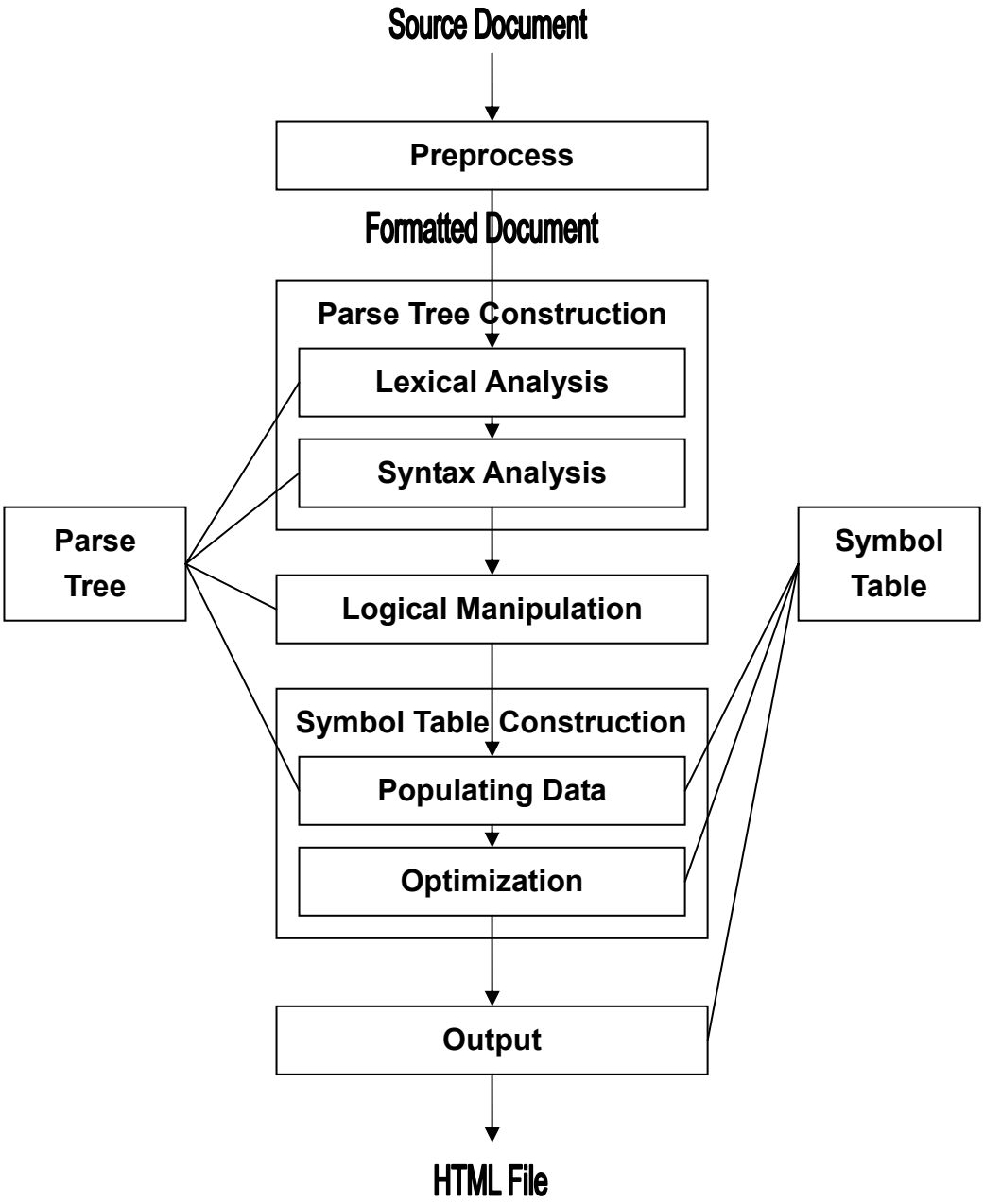


Figure 1.1: Workflow

2. PREPROCESS

2.1. Document Formatting

Document formatting, a new phrase, is introduced before the actual lexical analysis and syntax analysis. Its main purpose is to remove extra white spaces and ensure the uniqueness of ROID numbers.

The Formatdoc class implements the function of document formatting. The Formatdoc class scans every character in the original document and performs the desire tasks during the process. When the end of file is reached, it produces a new document in the defined format from the original document. In addition, the number of statements in the original document is also counted during the document formatting phrase.

2.1.1. Removing White Spaces

In a requirement specification document, extra white spaces might be needed to increase the readability of the document. However, the extra white spaces might confuse the scanner in lexical analysis. Thus, the Formatdoc class checks whether the current character is a tab, newline or extra white space then replaces it with a single white space character.

2.1.2. Uniqueness of ROID

ROID numbers must be unique for a requirement specification document. Each ROID number is associated with a term in a requirement statement. For a test case, the ROID numbers might be used to refer to the original requirement statements that derive the test case. If duplicates of ROID numbers exist, reference of requirement statements is confused.

To ensure the uniqueness of ROID numbers, the Formatdoc class keeps a record that store ROID numbers detected when scanning the document. If the pattern of "[number]" is detected, the value of the number is added to the record. Whenever a ROID number is encountered, the Formatdoc class checks if the ROID number has been declared in the record. If so, an error returns. Otherwise, the Formatdoc class adds it to the record.

2.2. Strength

The introduction of preprocessing document ensures the document is free of formatting errors before it is actually feed to the system. By placing error analysis before scanner and parser, errors can be reported to user in the earliest stage. In addition to error analysis, error recovery can also be included in the preprocess phrase. For example, common syntax errors, such as missing "." period after a requirement statement, can be corrected in this stage.

Because the preprocessing phrase is independent of other phrase, it becomes very easy to maintain. Modification and revision can be done anytime without worrying the dependency of other phrases of the system.

Due to its extensibility and maintainability, the new phrase of preprocessing document is desired.

2.3. Weakness

A new phrase always means more processing time and extra resource used. Because the original document is formatted into a new document, it needs extra processing and takes up hardisk space.

Although errors can be reported in the earliest stage, the document needs an extra phrase to scan for errors. Thus, the entire system becomes less efficient. If too many functions are perform in this phrase, the overhead of preprocess is more significant.

A new document is created in the document formatting process before passing to the actual test cases generation system. Extra hardisk space is needed to store this temporary file. The overhead of writing and reading hardisk data is also carried in when new files needed to be created.

3. PARSE TREE CONSTRUCTION

3.1. Tools

Lexical analysis and syntax analysis are two complex and sophisticated processes of language compilation. Due to their complexity of implementation, there are numerous tools for automating the creation of scanner and parser. Among the various tools, JFlex and BYacc/J are chosen to generate scanner and parser, respectively, in JAVA, which is the primary programming language of the project.

3.1.1. JFlex

JFlex is the JAVA implementation of the well-known C/C++ scanner generator, Flex. JFlex inherits all the features of Flex: efficient generated scanner, fast scanner generation process, and simple and convenient specification syntax. More importantly, JFlex supports Unicode and is platform independent.

3.1.2. BYacc/J

BYacc/J is an extension of the Berkeley YACC-compatible parser generator. With a "-J" flag while executing, BYacc/J generates LALR parser written in JAVA. BYacc is coded in C so that the parser generation is extremely fast. Moreover, the generated parser is compact and efficient.

3.1.3.Interaction between Scanner and Parser

When JFlex and BYacc/J are working together, they can generate powerful and efficient scanner and parser. The interaction between the scanner and parser is illustrated in Figure 3.1

Figure 3.1: Interaction of Scanner and Parser

3.2. Scanner

Lexical analysis is the first phase of language compilation. Its main task is to read the input characters and produce output as a sequence of tokens that the parser uses for syntax analysis.

3.2.1. Why Use JFlex

A traditional method to build a scanner is to construct a diagram that illustrates the structure of the tokens of the language, and then hand-translates the diagram into a program for finding token. However, this method of building scanner is complicated and involves advanced pattern-matching techniques. Hand-translating the scanner is also time-consuming and error-prone. Thus, a scanner generator utilizing the best-known pattern-matching algorithms is needed to automate the construction of an efficient scanner. JFlex is chosen as the tool to combat the problems of scanner generation. With the help of JFlex, the task of building scanner becomes simple and straight-forward.

3.2.2. Tokens Definition

Instead of designing sophisticated algorithms for pattern-matching, defining the set of tokens and the associated actions is the only task needed to be done, with the aids of JFlex. Table 1 lists the set of tokens for system level requirement specification language. The first column contains the patterns of ASCII characters to be matched;

the second column contains the symbols of tokens passed to the parser; the third column contains the attributes of the tokens to be retrieved. In addition, NUM, ENCL_STRING_TEXT and NONINTERUPTED_TEXT are three regular definitions that define three sets of regular expressions which match the patterns of digital number, string enclosed by "< and ">, and string enclosed by "{" and "}", respectively.

Table 1: Tokens Definition

Patterns	Tokens	Attributes
if	IF	
the system shall	THE_SYSTEM_SHALL	
one or more	ONE_OR_MORE	
all	ALL	
of the following conditions are true	OF_FOLLOWING_CONDITIONS_ARE_TRUE	
is true if	IS_TRUE_IF	
Mutually exclusive conditions	MUTUALLY_EXCLUSIVE_CONDITIONS	
Upon receipt of a	UPON_RECEIPT_OF_A	
when	WHEN	
is produced	IS_PRODUCED	
upon internal event	UPON_INTERNAL_EVENT	
and	AND	

or	OR	
not	NOT	
,	,	
.	.	
;	;	
[	[	
]	]	
(	(	
)	)	
NUM	NUM	Double
ENCL_STRING_TEXT	TEXT	String
NONINTERUPTED_TEXT	NONTEXT	String

3.3. Parser

Syntax analysis is the next phase of language compilation following lexical analysis. The parser obtains a sequence of tokens from the scanner and verifies that the sequence can be generated by the grammar of the source language. Parser is expected to report syntax error in an intelligible fashion and recovers from common errors so that it can resume the process of syntax analysis.

3.3.1. Why BYacc/J

Unlike scanner construction, there are relatively simple algorithms and techniques that can be applied to implement an efficient parser directly by hands. For example, a predictive parser can be built upon LL grammars with top-down parsing algorithms and recursive-descent parsing functions. With well-defined unambiguous LL grammar rules, it is possible to construct an efficient top-down LL parser. However, LL grammars can merely parse a limited range of context-free languages, while LR parsing covers a larger set of context-free grammars. Top-down parsing techniques, known as shift-reduce parsing, is applied to construct an efficient LR parser. In common practice, parser generator is usually used to construct parser for LR grammars because of its complexity of implementing a LR parser and its utility of well-developed shift-reduce parsing algorithms. Thus, BYacc/J is used to generate a context-free LRLA parser.

3.3.2. Grammar Rules

To seize the advantages of parser generator, a set of unambiguous LR grammars is necessary to be defined in BNF forms. Figure 3.2 lists the collection of LR grammar rules for system level requirement specification language. Text in bold are names of terminals (tokens).

Specification ::= Requirements Requirement | <>

Requirement ::=

StimulusOrResponse **, if** Condition **, the system shall [** Roid **]** Response **.**

| StimulusOrResponse**, the system shall [** Roid **]** Response **.**

| StimulusOrResponse**, the system shall [** Roid **]** Response **if one or more of following conditions are true :** Conditions **.**

| StimulusOrResponse**, the system shall [** Roid **]** Response **if** { **one or more | all** } **of following conditions are true :** Conditions **.**

| DefinedCondition **is true if** Condition **[** Roid **].**

| DefinedCondition **is true if** { **one or more | all** } **of following conditions are true :** Conditions **.**

| **Mutually exclusive conditions :** Conditions **.**

| nontext

StimulusOrResponse ::=

Upon receipt of a Stimulus

| **When** Response **is produced**

| **Upon internal event** Event

Condition ::=

Condition **and** Condition

| Condition **or** Condition

| **(** Condition **)**

| **not** Condition

| **text**

Conditions ::= Condition **[** Roid **]** | Condition **[** Roid **]** ; Conditions

Figure 3.2: LR Grammar Rules in BNF Form

3.4. Parse Tree

A data structure called Parse Tree is designed to internally represent the source document, so that the data can be further manipulated for logical optimization as well as cases generation testing.

3.4.1. Data Structure

The Parse Tree is constructed with nodes. Each node is decorated by attributes of statements in the source document. The Parse Tree node includes the following attributes:

- kind – indicates which kind of statements the node is representing (i.e. REQUIREMENT, CONDITION, etc)
- type – indicates which variation of the specific statement (i.e. OR, AND of Condition)
- ROID – contains the value of ROID number; can be empty
- text – contains the string of text; can be empty
- child one – points to the next branch of the Parse Tree; can be null
- child two – points to the next branch of the Parse Tree; can be null

With the attributes, nodes can be distinguished from each other and the information is useful for further manipulation.

The ParseTree class is the main implementation of Parse Tree node in JAVA. The structure of ParseTree class is illustrated in Figure 3.3:

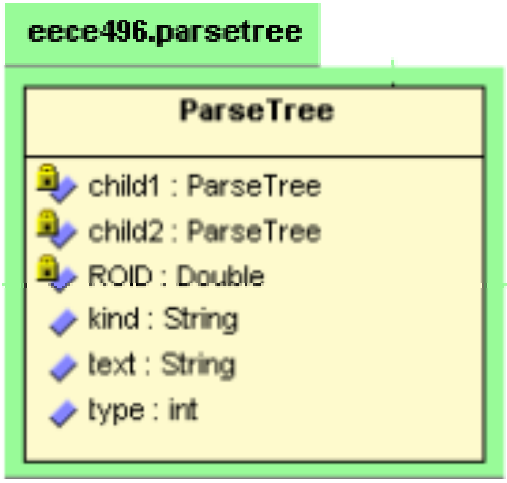


Figure 3.3: UML of ParseTree class

3.4.2. Semantic Actions

The Parse Tree is fully constructed after the lexical analysis and the syntax analysis. In the parser, each grammar rule is associated with a semantic action. The semantic actions implemented in the ParseTree class are being called upon by the Parser to construct the Parse Tree representation of the source document. In the parsing phrase, a new Parse Tree node is created with the specific semantic action to represent the current statement when a grammar rule is matched. For the system level requirement specification language, there are five semantic actions listed in Table 2:

Table 2: Semantic Actions

Non-Terminals	Semantic Actions	Parameters
Requirement	T_req	type, child one, child two, ROID, text
Requirements	T_reqs	child one, child two
StimulusOrResponse	T_stimOrResp	type, text
Condition	T_condition	type, child one, child two, text
Conditions	T_conditions	type, child one, ROID, text

4. LOGICAL MANIPULATION

4.1. Logical Manipulator

In requirement specification document, conditions are usually represented in logical expressions to derive the response, which may make the logical expressions of conditions too complex to be generated to test cases. Thus, these logical expressions must be transformed into some logical manipulator, which is able to implement the algorithm for converting logical expressions into a simpler form.

4.1.1. The DNF Conversion Algorithm

The basic approach of logical simplification is to transform the logical expression to disjunctive normal form that represents the fundamental conditions of the system. In DNF form, logical expressions of conditions can be divided into individual test cases of a particular response. As a result, a simpler set of test cases can be generated.

To derive the DNF form, logical equivalences laws are applied in the transformation process. Figure 4.1 lists the collection of logical equivalences laws used for DNF form derivation. Symbols *p*, *q*, and *r* are any logical statement variables; symbol *t*, a tautology, means always true; symbol *c*, a contradiction, means always false.

Identity laws:	$p \wedge t \equiv p$	$p \vee c \equiv p$
Negation laws:	$p \vee \sim p \equiv t$	$p \wedge \sim p \equiv c$
Double negation laws:	$\sim (\sim p) \equiv p$	
Idempotent laws:	$p \wedge p \equiv p$	$p \vee p \equiv p$
De Morgan's laws:	$\sim (p \wedge q) \equiv \sim p \vee \sim q$	$\sim (p \vee q) \equiv \sim p \wedge \sim q$
Universal bound laws:	$p \vee t \equiv t$	$p \wedge c \equiv c$
Absorption laws:	$p \vee (p \wedge q) \equiv p$	$p \wedge (p \vee q) \equiv p$
Negations of t and c:	$\sim t \equiv c$	$\sim c \equiv t$
Distributive law:	$p \wedge (q \wedge r) \equiv (p \wedge q) \vee (p \wedge r)$	

Figure 4.1: Logical Equivalences Laws

Besides the Distributive law, all other laws reduce the logical expression to a simpler form. Then the expression is rewrite to DNF form according to the Distributive law.

4.1.2. The LogicalManipulator Class

The algorithm of logical manipulation is implemented in the LogicalManipulator class to transform the structure of the Parse Tree. Since the logical expressions are denoted as Condition nodes, the logical manipulator retrieves the root of the sub-tree of Condition nodes. From bottom up, it recursively compares the structures of each branches to the left hand side of the logical equivalences laws. Once the structure is matched, the appropriate manipulation function which implements the specific logical equivalence law is called upon to transform the logical expression into the

right hand side of the law. For example, the deMorgan(ParseTree p) function is the implementation of De Morgan's law. Sub-trees of Condition nodes representing both sides of the De Morgan's law is illustrated in Figure 4.2:

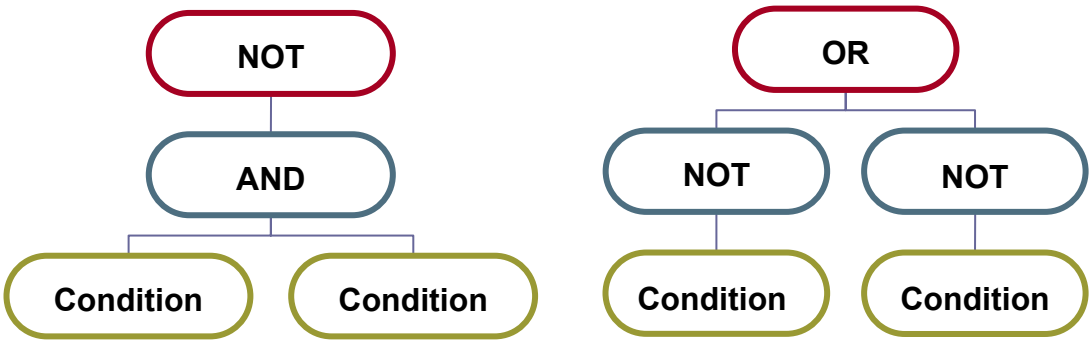


Figure 4.2: Tree Representation of De Morgan's Law

Then deMorgan() function transforms the left hand side to the right hand side of De Morgan's law in order to produce a simpler logical expression. The pseudo-code of De Morgan's law is listed in Figure 4.3:

```
ParseTree temp = p.child1;

if ( p.type == NOT && p.child1.type == AND)
   p.type = OR;
else if (p.type == NOT && p.child1.type == OR)
   p.type = AND;

p.child1.type = NOT;
p.child2.type = NOT;
p.child1.child1 = temp.child1;
p.child2.child1 = temp.child2;
```

Figure 4.3: Pseudo-Code of De Morgan's Law

First of all, the child node of the current node is stored in a temporary variable. If the structure of the sub-tree is matched to one of the De Morgan's law, transformation of the logical expression occurs. The current node is changed to type OR from AND or to type AND from OR corresponding to the type of De Morgan's law being applied. The child nodes are changed to type NOT with the values of child nodes of the temporary variable.

Other logical equivalences laws are implemented in similar manner. Each sub-tree of Conditions is recursively transformed until no laws are matched.

4.2. Future Improvements

There are several possible improvements in the current logical manipulator.
The current algorithm is able to convert any binary logical expressions to DNF forms. However, the DNF form converted might not be the simplest. In order to obtain the simplest DNF form, Commutative laws, Associative laws and the reverse version of Distributive law must be implemented. A more intelligent structure matching algorithm should be designed to determine every step of the transformation process with the conception of the whole sub-tree. Nevertheless, this algorithm has to be sophisticated enough to simulate the process of transforming logical expressions into simplest forms as human brain does.

In addition, the support of implication should be added in the revised version of the logical manipulator. New attribute values which indicates an implication Condition node must be added into the Parse Tree. The corresponding logical equivalences laws for implication should also be carried out in the logical manipulator. Thus, the logical manipulation process can transform a boarder range of logical expressions.

Eventually, quantifier in logical statements must be supported, which might require modification or re-design of the syntax and other phrases. As a result, the generator supports the entire range of logical expressions and gives the broadest freedom for the requirement specification document writers.

5. SYMBOL TABLE CONSTRUCTION

5.1. Symbol Table

Symbol Table is a data structure used to store the information from the attributes of Parse Tree for test cases generation. It is a collection of the relationship between a Response or Condition and a list of Stimulus and Conditions deriving to the particular Response or Condition. Each relationship forms the basic foundation of a test case.

5.1.1. Data Structure

Symbol Table has three columns. The first column, Response/Condition, contains the main Response or Condition implied from the test cases. The first column serves as the key of each row. The second column, Type, indicates whether the key is a Response or Condition. The third column, Mutual Exclusion, contains information derived from the Mutually Exclusive statements. Each row also points to a list of test cases. Each element of the list represents a test case which consists of a list of Stimulus and Conditions cause the Response or Condition, and a list of ROID numbers associated. Symbol Table relates the sets of Stimulus and Conditions to the Response or Condition. Figure 5.1 illustrates the structure of Symbol Table.

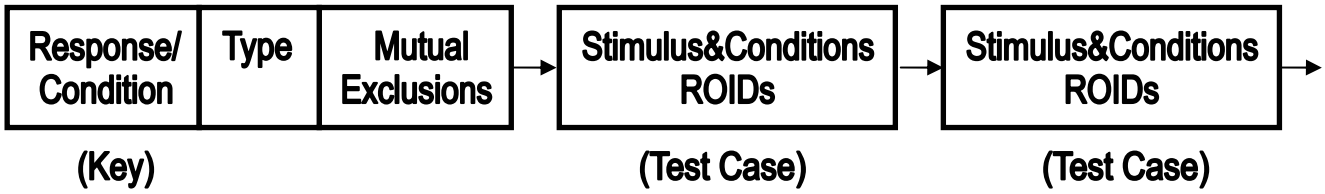


Figure 5.1: Structure of Symbol Table

5.1.2. The SymbolHashtable class

The SymbolHashtable class is the implementation of the Symbol Table. It contains two tables. One is used to store information of Type and Mutual Exclusions. The other functions as the storage for the list of test cases. Both tables share the same key, which is the Response or Condition.

The private member symbolInfoTable is the table that contains the three columns data with the first column as the key. It has a list of symbolInfoTableElement (SITE) which represents a row in the Symbol Table. Each SITE has private members to store the information of Type and Mutual Exclusions.

The other private member parentInfoTable contains the list of test cases for each Response or Condition that acted as the key of the table. The parentList class represents the list of test cases while each case is implemented in the parentListElement class. Each parentListElement object has two vectors for each test case: one for Stimulus and Conditions while the other for ROID numbers.

5.2. Populating Data

Initially, the Symbol Table is empty and contains no data for test cases generation. Then, followed by the phrase of logical manipulation, data is populated into the Symbol Table from the Parse Tree.

5.2.1. Converting DNF into Test Cases

After the process of logical manipulation, the logical expressions of Conditions in the Symbol Table are expected to be in DNF form. In DNF form, the expression is a disjunction of terms that are, indeed, conjunctions themselves. The expression will be evaluated as true if one disjunction is true, whether the other disjunctions are true or false. Therefore, each disjunction is a test case and the whole expression is a set of test cases. Since disjunction is logical OR operation and conjunction is logical AND operation, all terms in a AND operation are collected into a test case. The following pseudo-code in Figure 5.2 recursively checks the attributes of the Condition nodes in the Parse Tree. If the Condition node is in type AND, texts of the child nodes are collected in a vector.

```
dnf() {
    if OR type
        clear vector and add vector to Symbol Table as a new test case;
        child1.dnf();
        child2.dnf();
    else if AND type
        if child1.type is TEXT or NOT
                add text to vector;
        else
                child1.dnf();
        if child2.type is TEXT or NOT
                add text to vector;
        else
                child2.dnf();
    return;
}
```

Figure 5.2: Pseudo-Code of DNF to Test Cases

5.2.2. Populator

The Populating process is occurred in the Requirement level of Parse Tree. For the reason that each Requirement states a set of Stimulus and Conditions in order to derive the Response of the system, the test cases for the particular Requirement are determined in the Requirement statements.

The populator checks the type of the Requirement nodes recursively because the type attributes indicate the structure of the Requirement statement. As the sub-tree of Requirement nodes known, information of test cases becomes easy to be determined and inserted into the Symbol Table. For example, type 1 of Requirement represents the syntax of

StimulusOrResponse **, if** Condition **, the system shall [** Roid **]** Response **.**

and has the structure of Requirement sub-tree in Figure 5.3:

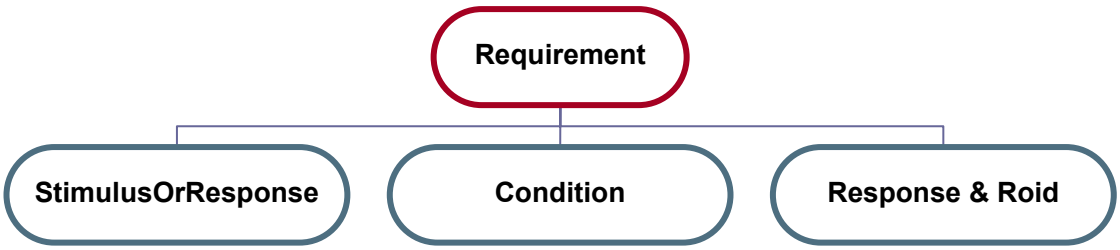


Figure 5.3: Sub-Tree of Requirement Node

The Response is the key of the new row in Symbol Table and the type is RESPONSE. The Condition node is needed to be converted from DNF form to a list of test cases with the algorithm discussed above. Then StimulusOrResponse node and Roid number are added to the test cases of the Response. As a result, a set of test cases for the particular Response in the Requirement statement is inserted into the Symbol Table. If there are more Requirement statements for this Response, the sets of test cases are extended to the list of test cases in the Symbol Table.

5.3. Optimization

In the phrase of test cases optimization, the Symbol Table is fully populated with information of all test cases. Optimization is done upon the Symbol Table to substitute Response with Stimulus and Conditions, and to remove duplicates of Conditions in test cases.

5.3.1. Optimizer

Since a Response can be derived from another Response in a test case, the causing Response should be substituted by the corresponding Stimulus and Conditions. Consider the following Requirement statements:

> **Upon receipt of a** <S1>**, if** <C1> **, the system shall [** 1 **]** <R1> **.**
>
> **Upon internal event** <E1>**, the system shall [** 2 **]** <R1> **.**
>
> **When** <R1> **is produced, the system shall [** 3 **]** <R2> **if one or more of following conditions are true :**
>
> <C1> **[** 4 **]**;
>
> <C2> **[** 5 **].**

This block of Specification produces the following test cases:

<S1>, <C1> => <R1> [1]

<E1> => <R1> [2]

<R1>, <C1> => <R2> [3, 4]

<R1>, <C2> => <R2> [3, 5]

Having observations from the above production, <R1> can be potentially substituted by its test cases in order to produce simpler test cases. The optimizer checks the values of test cases in each row in the Symbol Table and compares them to the keys. If a match is detected, optimizer substitutes the causing Response with its corresponding Stimulus and Conditions. In substitution, a logical manipulation of the two lists of test cases is performed on the causing Response and the resulting Response. In the list of test case of the resulting Response, the value of the causing Response is replaced by its Stimulus and Conditions. If there is more than one test case in the causing Response, new test cases for the resulting Response will be produced. Then, a new list of test cases is built and inserted into the existing list of test cases. ROID numbers of the causing Response are also added to the resulting Response. The substituted test cases are as the following:

<S1>, <C1> => <R1> [1]

<E1> => <R1> [2]

<S1>, <C1>, <C1> => <R2> [1, 3, 4]

<S1>, <C1>, <C2> => <R2> [1, 3, 5]

<E1>, <C1> => <R2> [2, 3, 4]

<E1>, <C1>, <C2> => <R2> [2, 3, 5]

5.3.2. Removing Duplication

After substitution, duplicates of Conditions are usually produced in a test case. The duplication increases the complexity of the test case and is redundant for test cases generation. Observing the previous test cases production, the Condition <C1> is duplicated in the test case:

<S1>, <C1>, <C1> => <R2> [1, 3, 4]

For each element in the list of test cases, the optimizer removes duplicates of each Stimuli and Condition in order to simplify the test case. After the removal of duplication, the test case would be as follows:

<S1>, <C1> => <R2> [1, 3, 4]

where ROID numbers are not removed.

6. OUTPUT

6.1. Test Cases Display

The final phrase of the test cases generation is displaying the test cases to users from the information in Symbol Table. The generated test cases are formatted as HTML files and presented to users in an intelligible fashion.

6.1.1. Why HTML

There are numerous file formats in which the test cases can be stored, such as MS Document, PDF, HTML, plain text and so on. Many of the file formats need extra overhead before storing the test cases into files. For example, both MS Document and PDF require additional API for utilizing the file formats. On the other hand, some file formats support few or none formatting features (i.e. plain text). For the ease of use and features supported, HTML format is the best choice as it supports a wide range of formatting features and needs no extra overhead for utility. In addition, HTML file is platform-independent and the most popular file format.

6.1.2. Format Template

Figure 6.1 shows the template of HTML file upon which test cases are formatted.

<Title>

Test Case ID	ROIDs	Test Case			Pass/Fail
		Mutually Exclusive	Stimuli and Conditions	Response/ Condition	
1	1,2		-<S#> -<C#>	-<R#>	☐ Pass / ◉ Fail

Figure 6.1: HTML File Template

In the HTML file, each row in the table represents a single test case. The first column is the Test Case ID with initial index of one. The second column contains the ROID numbers involved in each test case. The next three columns compose the main part of a test case. For Response-type test case, the Mutually Exclusive column is empty while there might be entries for Condition-type test case. The Stimuli and Conditions column and Response/Condition column together represent a test case. The last column asks for user input to indicate whether the test case is passed of failed.

6.1.3. The TestCaseGenerator Class

The TestCaseGenerator class facilitates the functions of retrieving test cases from Symbol Table and formatting them to HTML file.

Before the process of retrieving test cases, the TestCaseGenerator class creates an empty HTML file and writes a HTML header. The tags necessary for creating a HTML file and table are included in the header.

In Symbol Table, each element in the Stimulus and Conditions list of each row denotes a single test case. First of all, the TestCaseGenerator class retrieves the list of keys which is either Response or Condition. Then it retrieves the whole row for each key. It formats the data of Mutual Exclusions and Response into HTML file from the values of symbolInfoTable private member. Secondly, it retrieves the list of Stimuli and Conditions for each Response. Then, each element of the list is taken. Finally, it takes the vector of ROID numbers and the vector of Stimuli and Conditions for each element and formats them into the file. As a result, a single test case is formatted to the HTML file.

After all information in the Symbol Table is retrieved and formatted into the HTML file, footer is written to complete the table and close the HTML file. In addition, credits and copyright information is also included in the HTML file.

6.2. Graphical User Interface

Graphical User Interface (GUI) module is the front end of the system. It facilitates the interaction between the user and the system. It is responsible for obtaining necessary information from user for test cases generation and provides feedback about the generation process.

6.2.1. Model-View-Controller Pattern

The underlying implementation of GUI resembles a model-view-controller (MVC) design pattern. The purpose of employing the MVC pattern in the design is to decouple the front-end, the system and the interaction from each other. Figure 6.2 represents the MVC pattern.

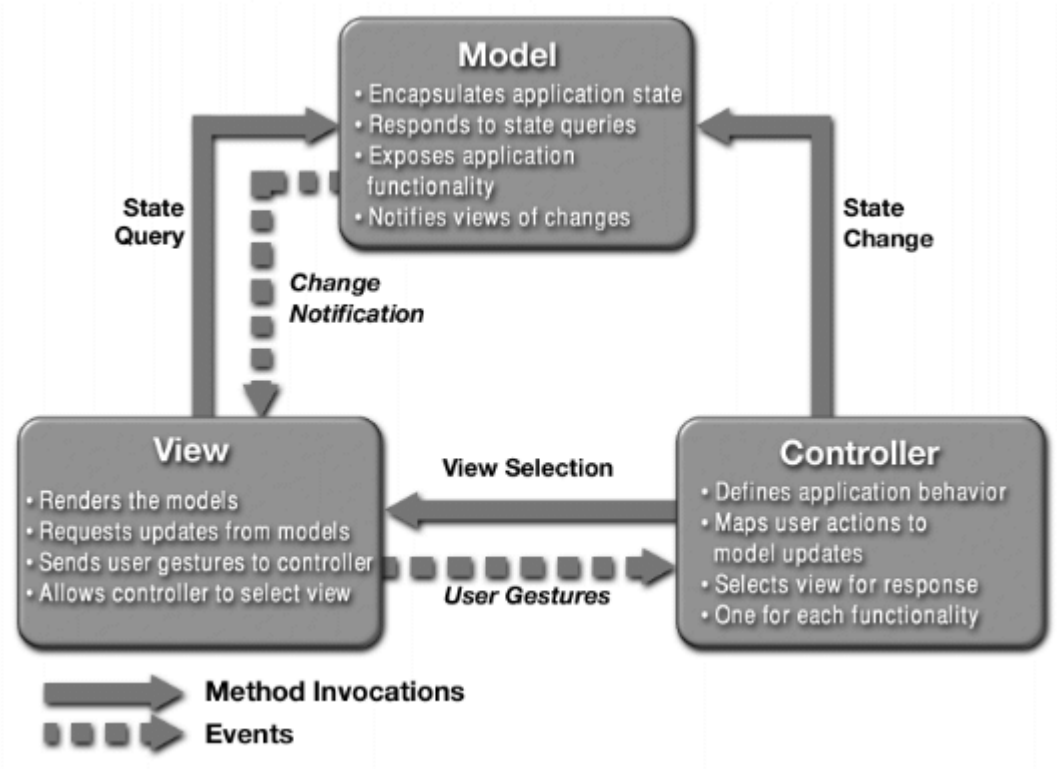


Figure 6.2: MVC Pattern

The system, as the model in the pattern, is relatively stable and would not experience significant change in the near future. The front-end, as the view and controller, is more prone to modification. Although the current system is deployed as a standalone application, other front-ends such as JSP pages, RMI and Applet might be done in subsequent release. The MVC pattern effectively decouples the three roles and allows them to vary independently. Modifiability is the emphasis of the GUI module.

6.2.2. View and Controller

The GuiDriver class implements the view in the MVC pattern. It is responsible to create the main application window and various dialogs. Figure 6.3 shows the main window of the GUI module.

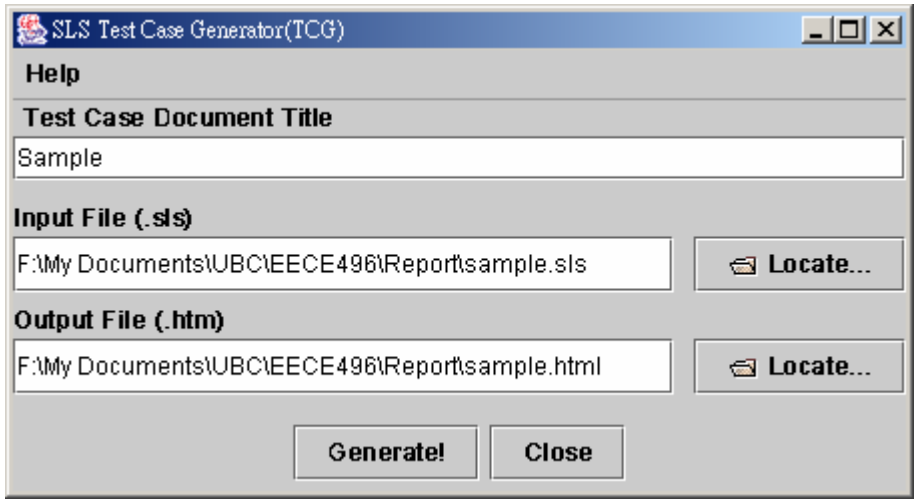
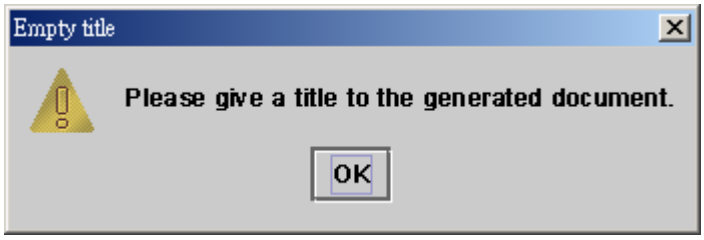


Figure 6.3: Main Window

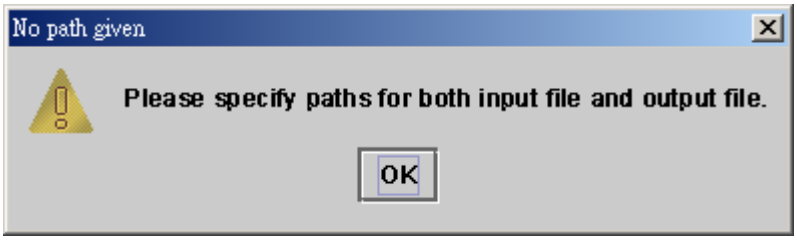
The GuiDriverControl class represents the controller in the MVC pattern. Its tasks include passing user input to the model, managing the phrases of test cases

generation, and providing feedback to user by calling the view to pop up dialogs.

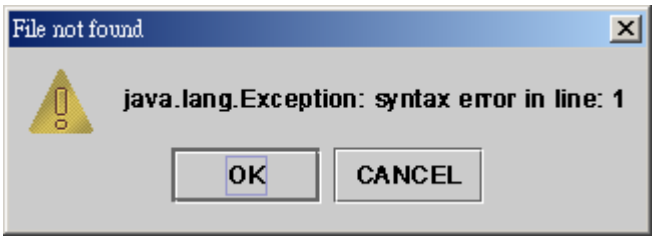
The GuiDriverControl class holds the global instances of Parse Tree and Symbol Table. It manages the process of test cases generation by scheduling the phrases of the system and manipulating the two data structures with the model. Then it provides visual response to user while it calls the view to pop up corresponding dialogs in each phrase. Figure 6.4 lists the three Warning Dialogs to alert user that errors occur.



(i)



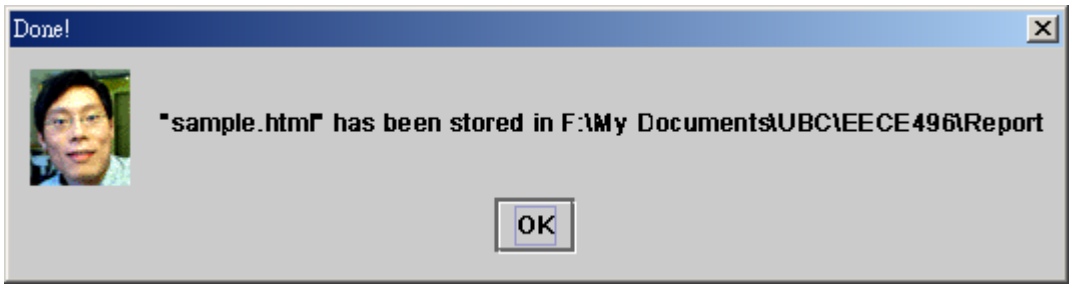
(ii)



(iii)

Figure 6.4: (i) Title field is empty; (ii) No file path is given; (iii) Syntax error found

Figure 6.5 shows the Done Dialog which pops up when the test cases are successfully generated, and the About Dialog which displays credits and copyright information.



(i)

(ii)

Figure 6.5: (i) Done Dialog; (ii) About Dialog

7. CONCLUSION

This project has identified two major methodologies for automation of system-level specification-base testing:

1. language compilation; and
2. logical manipulation.

In language compilation, lexical analysis and syntax analysis were applied to construct a tree structure representing the specification. These algorithms are similar to the methods of building a compiler. Upon the tree structure, logical manipulation was performed to simplify sets of test cases. The underlying algorithms of logical manipulation are based on the logical equivalence laws. With these algorithms and techniques, our system successfully generated a set of test cases from a system level requirement specification document. Therefore, the efficiency and quality of system-level specification-based testing have been improved and enhanced.

Appendix

Documentation

The documentation of source code is generated by Javadoc and maintained in the URL http://kelvincai.no-ip.com/eece496/javadoc/index.html

Executable

Executables of the system for various platforms are generated and can be downloaded from the URL http://kelvincai.no-ip.com/eece496/executable

Source Code

The source code of the entire project can be downloaded from the URL http://kelvincai.no-ip.com/eece496/src