Automated Generation of Decision Table and Boundary Values from VDM++ Specification

Hiroki Tachiyama[1], Tetsuro Katayama[2], and Tomohiro Oda[3]

[1] University of Miyazaki, Japan tachiyama@earth.cs.miyazaki-u.ac.jp
[2] University of Miyazaki, Japan kat@cs.miyazaki-u.ac.jp
[3] Software Research Associates, Inc. Japan tomohiro@sra.co.jp

Abstract. The purpose of this research is to improve the efficiency of test process in software development using formal methods. This paper proposes both methods to generate decision tables and boundary values automatically from the specification written in the formal specification description language VDM++ to reduce the burden in designing test cases. Each method has been implemented as VDTable[1] and BWDM[2] in Java. They parse the VDM++ specification by using VDMJ[3]. VDTable treats with the parsed result as internal representation data, generates a truth table, and displays a decision table on the GUI. BWDM analyzes boundary values of the argument types in function definitions and the condition expressions in if-expressions, and outputs the result as a csv file. This paper describes the both tools, shows their results applied to simple VDM++ specifications, and discusses the improved efficiency of designing test cases.

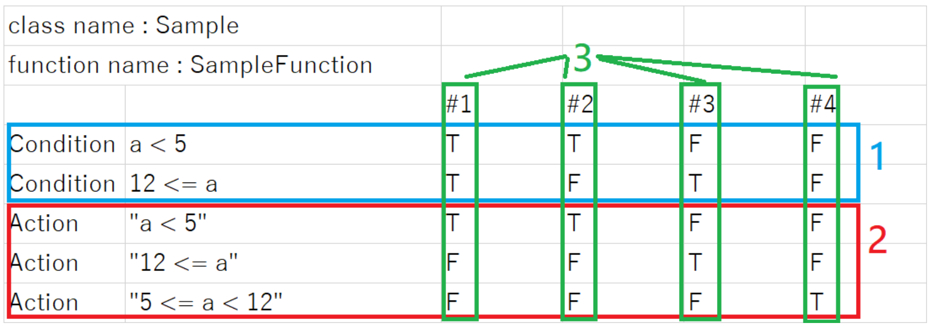
Keywords: VDM, Software Testing, Decision Table, Test Case, Automated Generation.

1 Introduction

In the design of software development, the Decision Table [1] is used to represent the accurate logic of software. The decision table expresses the logic of software by dividing it into three terms: conditions, actions, and combination rules of conditions. It's applied widely not only at the design, but also the testing. However, it is sometimes troublesome and time consuming to manually prepare the decision table, because it is necessary to understand the target system or specification and to extract conditions and actions from specifications.

Also, the software testing plays an important role in software development. The testing involves design of test cases and design documents based on the specification, which costs time and efforts. Techniques to find bugs with fewer test cases and methods to design such test cases will reduce the cost of the testing. Boundary Value Analysis [2, 3] is generally known as a promising testing technique for this purpose.

[1] https://earth.cs.miyazaki-u.ac.jp/research/2016/vdtable.html
[2] https://earth.cs.miyazaki-u.ac.jp/research/2016/bwdm.html
[3] https://github.com/nickbattle/vdmj



class name : Sample					
function name : SampleFunction					
		#1	#2	#3	#4
Condition	a < 5	T	T	F	F
Condition	12 <= a	T	F	T	F
Action	"a < 5"	T	T	F	F
Action	"12 <= a"	F	F	T	F
Action	"5 <= a < 12"	F	F	F	T

Fig. 2.1. A example of Decision Table

We propose a pair of methods to generate (1) a decision table (2) and boundary values from a formal specification to improve the efficiency of the testing in the software development with VDM++. VDM (Vienna Development Method) [4, 5, 6] is one of the Formal Methods, and VDM++ [7] is the object-oriented dialect of VDM-SL that is the formal specification description language of VDM. Both methods first parse the VDM++ specification. Thereafter, the former extracts conditions and actions from the parsing data, creates truth values and then generates a decision table. The latter extracts the if-expressions and argument types of function definition, performs boundary value analysis, and finally generates test cases. Each of the proposed methods is implemented as tools in Java, namely VDTable [8, 9] and BWDM [10]. VDTable generates a decision table from VDM++ specification. BWDM generates test cases based on the boundary value analysis of the VDM++ specification. The contents of the following chapters are shown below.

Existing techniques that this work is based upon and related work is briefly introduced in Section 2.

In Section 3, two tools implemented based on the proposed methods are explained.

In Section 4, the tools are evaluated.

In Section 5, we summarize this research and show future issues.

2 Background

In this section, we describe the decision table and boundary value analysis. We also explain related research what is improve the efficiency of software testing.

2.1 Decision Table

The decision table [1] summarizes actions on inputs in a table form. It can be used at the software design and the test design. Fig. 2.1 shows a sample of decision table. It consists of three parts.

1. Condition part (1 in Fig. 2.1) describes the conditional statements in software and the possible combinations of these truth values.
2. Action part (2 in Fig. 2.1) describes the operation of the software for the combination that the truth value of the condition part can have.
3. Rule part (3 in Fig. 2.1) crosses the condition part and action part vertically. This part shows the operation of action part for the combination of each truth value of the condition part. For example, in the case of #2, it shows when an input variable a that $a < 5$ is True and $12 <= a$ is False, the “a < 5” (String value) is returned. The other actions are not executed.

2.2 Boundary Value Analysis

It is known from empirical studies that many software bugs exist on “boundaries” within software. Therefore, Equivalence Partitioning [11] and Boundary Value Analysis [2, 3], which are software testing methods, are used.

Equivalence partitioning is one of the test design technique, focusing on inputs and behaviors to test objects and summarizing the range of inputs in which the test objects behave in the same way as equivalence classes. The input values in the equivalence class are regarded as the same behavior, and by narrowing down the input values, the total number of test cases can be reduced. Boundary value analysis is also a test design technique focusing on the fact that program bugs are likely to exist near the boundary of equivalence classes. Boundary value analysis is also a test design technique focusing on the fact that software bugs are likely to exist near the boundary of equivalence classes. It is based on an empirical rule that an error tends to occur at the time of judging the end condition of the loop and the branching condition.

2.3 Related Work

In software development using formal methods, research to improve the efficiency of testing process has been actively conducted [12]. In this section, we briefly describe two researches on software testing using VDM specifications.

CEGTest[4] is a tool for automatically generating a decision table. The cause result graph as the input of CEGTest has to be created manually by the user directly from the VDM++ specification. In other words, in order to obtain the decision table, the user needs to understand the contents of the VDM++ specification and to extract conditions and actions, which takes time and labor. On the other hand, VDTable, the decision table generation tool we developed in this research, does not need to manually create new data for generating the decision table by using the VDM++ specification as the tool input. Therefore, compared to CEGTest, the VDTable has the advantage that the time and labor required for generating the decision table is small.

Also, Overture[5] is the IDE for supporting the development using VDM, and it has editor, syntax checker, and many other functions. The support for Combinatorial

[4] http://softest.jp/tools/CEGTest/
[5] http://overturetool.org/

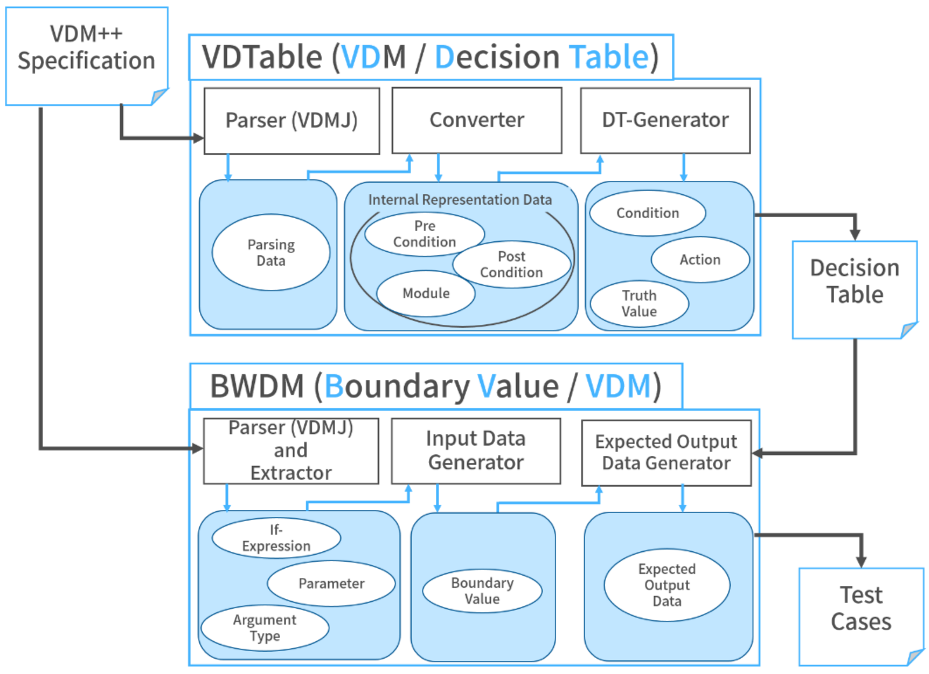


Fig. 3.1. The flow of two tools

Testing [13] is one of the function of Overture. This function enables to automatically generate test cases that is all combinations of input data specified by user using regular expression. Also, test is automatically conducted. Since, input data needs to be prepared by the user, by using BWDM to generate input data, it is possible to perform more effective test against bug finding. In other words, by specifying boundary values generated by BWDM as input data, boundary value analysis and combinatorial testing can be combined and conducted.

The two researches, CEGTest and combinatorial testing, are above highly related to this research, but neither of them has been conducted to reduce the labor and time for creating test cases and decision tables by automatically generating from the VDM++ specification to improve the work efficiency at the time of the test design.

3 Proposed Methods and Implemented Tools

We describe two tools, namely VDTable and BWDM, implemented based on the proposed methods in this research. Fig. 3.1 shows the ovreview of the tools. VDTable generates a decision table from the VDM++ specification automatically, and also BWDM generates, from the decision table and the VDM++ specification, test cases with the boundary values and expected outputs. These test cases can be used for testing of the software implemented from VDM++ specifications.

3.1 VDTable

VDTable [8, 9] is a tool to automatically generate a decision table from the VDM++ specification. VDTable has a GUI to select a VDM++ specification file for input, and display a decision table, and it consists of the following three modules.

- Parser
- Converter
- DT-Generator

The processing procedure of VDTable is shown below.

1. The user runs with a VDM++ specification file, which should pass the syntax and type checking.
2. VDTable parses the given VDM++ specification into an abstract syntax tree.
3. VDTable extracts if-expressions, cases expressions, pre- conditions and post conditions from the syntax tree and generates an internal representation data of those components.
4. VDTable extracts conditions and actions from the internal representation data, and creates truth values from them, and finally composes a decision table.

The current implementation of VDTable has two limitations. Although the parser accepts valid VDM++ specifications, the current implementation of the Converter and the DT-Generator takes only account of if expressions, cases expressions, preconditions and postconditions to generate a decision table. Another limitation is that truth values of conditions and actions are created only based on the control flow of each syntax. VDTable does not always generate all the combinations of truth values.
Below we explain each module.

Parser
VDTable uses VDMJ to obtain an abstract syntax tree of the given VDM++ specification, and outputs only syntax trees of function definitions as parsing data.

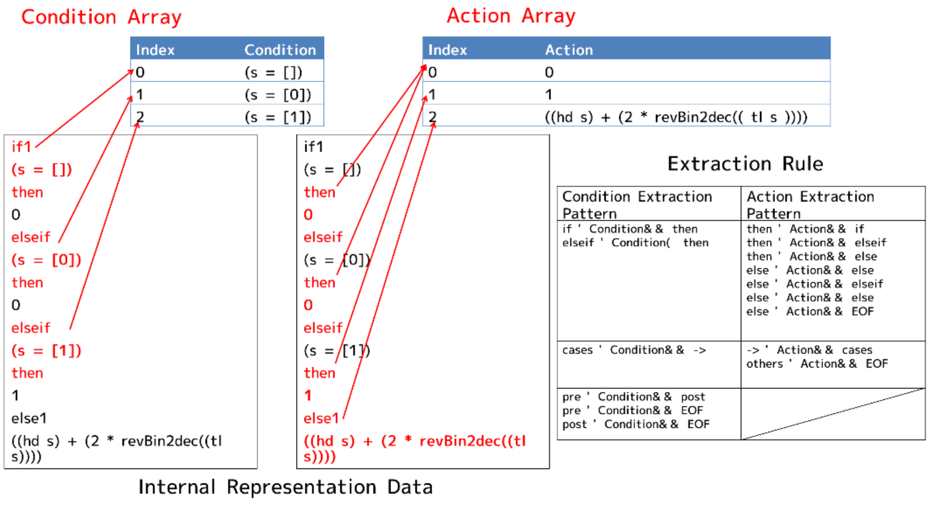
Converter
This unit converts the parsing data output by Parser into internal representation data in order to facilitate extraction of conditions and actions and creation of truth values which are necessary in generating a decision table.
Fig. 3.2 shows an example of VDM++ specification, and Fig. 3.3 shows an example of internal representation data generated from the specification of Fig. 3.2. The internal representation data consists of expressions whose keyword tokens, conditions and alternatives are divided into lines, and after each occurrence of “if” and “else” tokens, a number to indicate a corresponding pair is appended.

```
class Sample
functions

  revBin2dec : seq of nat -> nat
  revBin2dec( s ) ==
      if s=[] then 0
      elseif s=[0] then 0
      elseif s=[1] then 1
      else hd s + 2*revBin2dec( tl s );

end Sample
```

Fig. 3.2. An Example of VDM++ Specification

```
if1
(s = [])
then
0
elseif
(s = [0])
then
0
elseif
(s = [1])
then
1
else1
((hd s) + (2 * revBin2dec((tl s)))
```

Fig. 3.3. Internal Representation Data



Index	Condition
0	(s = [])
1	(s = [0])
2	(s = [1])

Index	Action
0	0
1	1
2	((hd s) + (2 * revBin2dec((tl s))))

Extraction Rule

Condition Extraction Pattern	Action Extraction Pattern

if ' Condition& & then
elseif ' Condition(then
	then ' Action& & if
	then ' Action& & elseif
	then ' Action& & else
	else ' Action& & else
	else ' Action& & elseif
	else ' Action& & else
	else ' Action& & EOF

cases ' Condition& & ->	-> ' Action& & cases others ' Action& & EOF
pre ' Condition& & post pre ' Condition& & EOF post ' Condition& & EOF	

Fig. 3.4. Making of the Condition Array and the Action Array from Internal Representation Data and Extraction Rule

DT-Generator

DT-Generator is an abbreviation of Decision Table Generator. It extracts conditions that is matched with the extraction pattern of the condition from internal representation data, and stores them in a String type array (Condition Array). It also extracts actions that is matched with the extraction pattern of the actions from internal representation data, and stores them in a String type array (Action Array). Each extraction pattern is defined by Extraction Rule. Fig. 3.4 shows examples of Condition Array and Action Array generated by DT-Generator from the VDM++ specification of Fig. 3.2. Also, Extraction Rule is shown in the Fig. 3.4 too. When DT-Generator extracts conditions and actions, CA-Table is generated. Fig. 3.5 shows the CA-Table generated from

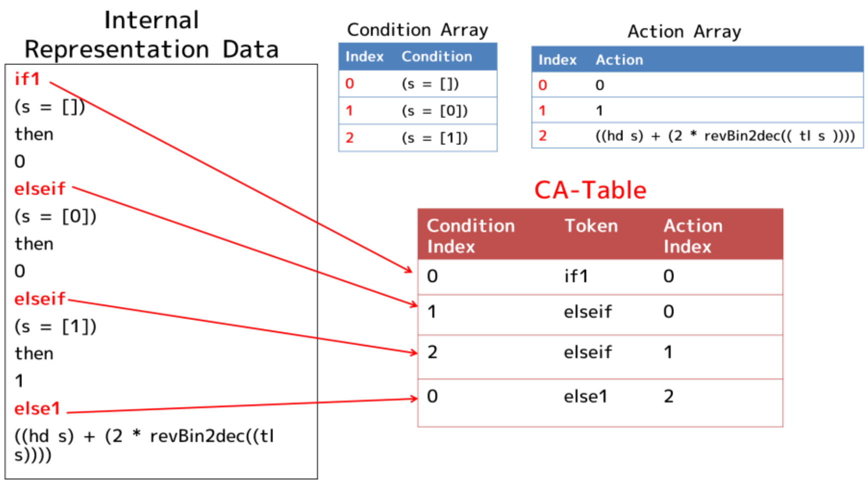


Fig. 3.5. Making of Condition Action Table

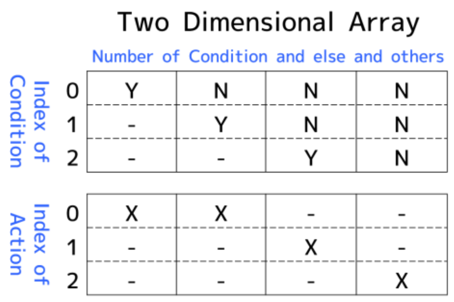
Fig. 3.4. CA-Table is an abbreviation of Condition Action Table, and it is a table that corresponds of conditions and actions. CA-Table consists of three columns: condition index, token, and action index.

This unit generates the truth value of the conditions and actions based on this CA-Table. The truth value generation procedure is shown below.

1. Create an array that is two dimensional and stores truth value.
2. Select the first column of created array.
3. Select a row of CA-Table from the first row.
4. Compare tokens and store truth value in the array,

   a. When “if”, “elseif”, “cases” are matched,
      (1) Store "Y" to the column selected for the conditional index row and "N" from the next column to the end of the column.
      (2) Store “X” in action index of selected column.
   b. When “else”, “others” are matched,
      (1) Store "N" to the column selected for the conditional index row and "-" from the next column to the end of the column.
      (2) Store “X” in action index of selected column.

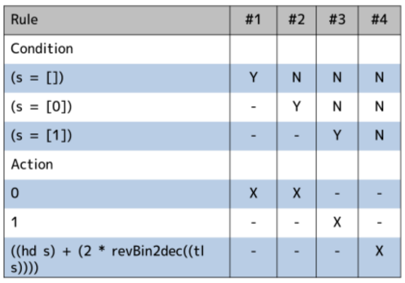
5. If there are unselected rows, select the next column of the array and return to 3. When there is no, truth value is completed.


Two Dimensional Array
Number of Condition and else and others
Index of Condition
Index of Action

0	Y	N	N	N
1	-	Y	N	N
2	-	-	Y	N

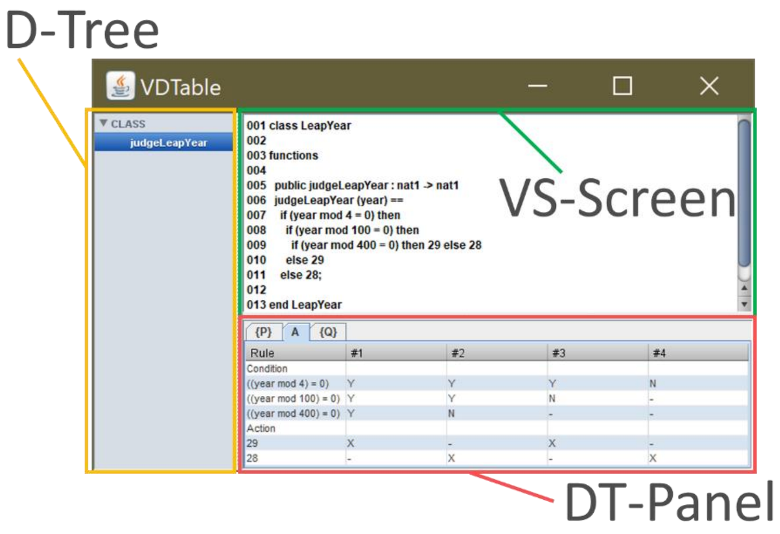
0	X	X	-	-
1	-	-	X	-
2	-	-	-	X

Fig. 3.6. The Truth Value



Rule	#1	#2	#3	#4
Condition				
(s = [])	Y	N	N	N
(s = [0])	-	Y	N	N
(s = [1])	-	-	Y	N
Action				
0	X	X	-	-
1	-	-	X	-
((hd s) + (2 * revBin2dec((tl s))))	-	-	-	X

Fig. 3.7. The Decision Table



```
001 class LeapYear
002
003 functions
004
005   public judgeLeapYear : nat1 -> nat1
006   judgeLeapYear (year) ==
007     if (year mod 4 = 0) then
008       if (year mod 100 = 0) then
009         if (year mod 400 = 0) then 29 else 28
010       else 29
011     else 28;
012
013 end LeapYear
```

Rule	#1	#2	#3	#4
Condition				
((year mod 4) = 0)	Y	Y	Y	N
((year mod 100) = 0)	Y	Y	N	-
((year mod 400) = 0)	Y	N	-	-
Action				
29	X	-	X	-
28	-	X	-	X

Fig. 3.8. Leap year specification

Fig. 3.6 shows a completed truth table. The finally generated decision table is shown in Fig. 3.7.

GUI parts and display

In this section, we show GUI parts of VDTable and the result of application to leap year specification in VDM++. Fig. 3.8 shows application to leap year specification. VDTable consists of the following three parts.

- VS-Screen (VDM++ Specification Screen)
  Display the VDM++ specification specified by the user along with the line number.

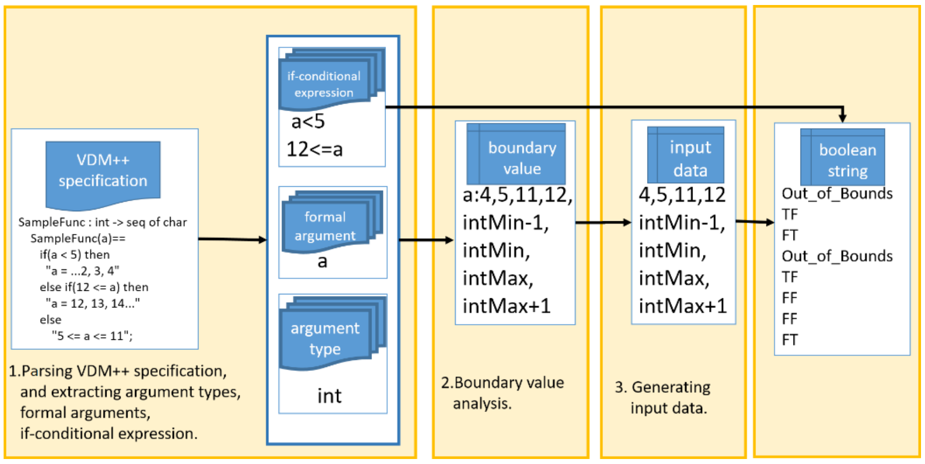

VDM++ specification
SampleFunc : int -> seq of char
SampleFunc(a)==
if(a < 5) then
"a = ...2, 3, 4"
else if(12 <= a) then
"a = 12, 13, 14..."
else
"5 <= a <= 11";
1.Parsing VDM++ specification, and extracting argument types, formal arguments, if-conditional expression.
if-conditional expression
a<5
12<=a
formal argument
a
argument type
int
boundary value
a:4,5,11,12, intMin-1, intMin, intMax, intMax+1
2.Boundary value analysis.
input data
4,5,11,12 intMin-1, intMin, intMax, intMax+1
3. Generating input data.
boolean string
Out_of_Bounds
TF
FT
Out_of_Bounds
TF
FF
FF
FT

Fig. 3.9. The process of Input Data Generator

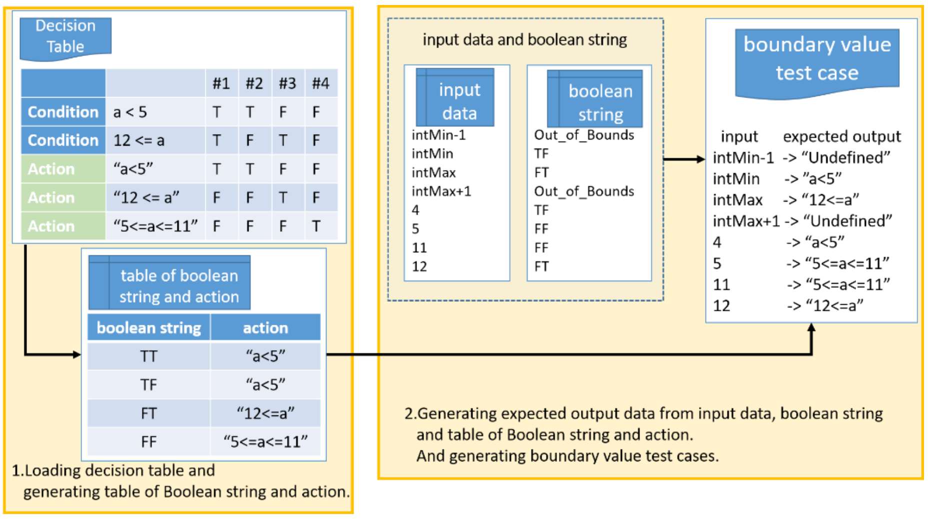


Fig. 3.10. The process of Expected Output Data Generator

- DT-Panel
  Display the decision table of the function definition or operation definition defined in the VDM++ specification. Also, by switching tabs, a decision table of modules, pre- conditions and post conditions is displayed individually.

- D-Tree
  Display the list of definition names in the VDM++ specification.

3.2 BWDM

BWDM [10] analyzes boundary values of function definitions in VDM++ specification, and automatically generates test cases that consists of boundary values and expected output. BWDM is a command line based tool, and it consists of following four processing modules.

- Parser
- Extractor
- Input Data Generator
- Expected Output Data Generator

BWDM takes two inputs: a VDM++ specification and a decision table generated by VDTable. The user previously describes the VDM++ specification and the generates of the decision table by VDTable. The Parser generates the abstract syntax tree from VDM++ specification by using VDMJ. The Extractor extracts if-expressions, parameter, and argument type.

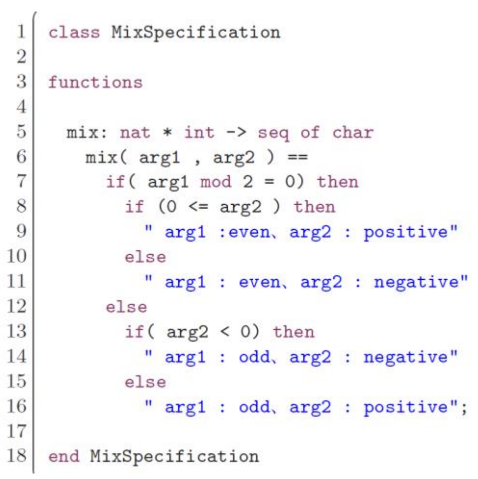
A decision table generated by VDTable is used to generate expected output data. BWDM analyzes the boundary value of the function definition described in the input of VDM++ specification, and finally outputs the generated boundary values and expected output data as test cases. The generated test cases are stored in a CSV (Comma Separated Values) file. Following sections, we describe about Input Data Generator and Expected Output Data Generator.

Input Data Generator
Fig. 3.9 shows the flow from the Parser to the Input Data Generator. The Input Data Generator analyzes of boundary values of if-expressions and type argument types that is extracted from the VDM++ specification.

The target if-expression is limited to inequalities and remainder expressions in which one side is a variable on both sides, and is also not supported for complex conditional expressions. Argument types are limited to integer types (int, nat, nat1).

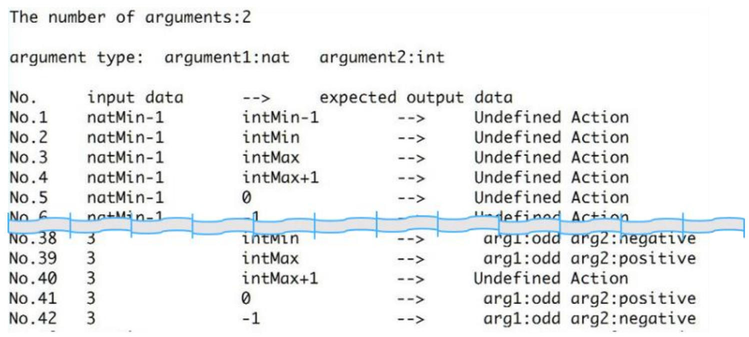
In the case of if-expressions, BWDM presents the value of the boundary at which the truth value changes for result of boundary value analysis. In the case of argument types, BWDM presents maximum value, maximum value + 1, minimum value, and minimum value - 1 of its type. The generated boundary value is used for deriving the Expected Output Generator. and it finally becomes the Input data of test cases.

Expected Output Data Generator
Fig. 3.10 shows the flow of the Expected Output Data Generator. The Expected Output Data Generator generates expected output data from boundary values generated by the



```
class MixSpecification

functions

  mix: nat * int -> seq of char
    mix( arg1 , arg2 ) ==
      if( arg1 mod 2 = 0) then
        if (0 <= arg2 ) then
          " arg1 :even、arg2 : positive"
        else
          " arg1 : even、arg2 : negative"
      else
        if( arg2 < 0) then
          " arg1 : odd、arg2 : negative"
        else
          " arg1 : odd、arg2 : positive";

end MixSpecification
```

Fig. 3.11. The VDM++ specification that mixed inequality and surplus expression



```
The number of arguments:2

argument type:  argument1:nat   argument2:int

No.     input data      -->     expected output data
No.1    natMin-1        intMin-1        -->     Undefined Action
No.2    natMin-1        intMin          -->     Undefined Action
No.3    natMin-1        intMax          -->     Undefined Action
No.4    natMin-1        intMax+1        -->     Undefined Action
No.5    natMin-1        0               -->     Undefined Action
No.6    natMin-1        -1              -->     Undefined Action
No.38   3               intMin          -->      arg1:odd arg2:negative
No.39   3               intMax          -->      arg1:odd arg2:positive
No.40   3               intMax+1        -->     Undefined Action
No.41   3               0               -->      arg1:odd arg2:positive
No.42   3               -1              -->      arg1:odd arg2:negative
```

Fig. 3.12. The generated test cases by applying the Fig. 3.11 to BWDM

Input Data Generator and the decision table. The pairs of input data and expected output data are presented as test cases.

Application Example

In this section, we show the result of application of BWDM. Fig. 3.11 shows the VDM++ specification that mixed inequality and surplus expression. And Fig. 3.12 shows test cases generated by BWDM from Fig. 3.11. These test cases are possible to use directly, such as combinatorial testing of overture.

4 Evaluation

In this section, we evaluate both tools. For both tools, we measure the time to finish their operation and evaluate what how much they can improve the efficiency of the work.

Table 5.1 Time taken for creation of decision tables for testers and VDTable (sec)

Tester	VDM++ Specification (the number of combinations of the truth values of the conditions)		
	Specification A(4)	Specification B (16)	Specification C (256)
A	252	405	1131
B	213	559	1292
C	169	499	1194
D	240	435	1859
E	134	557	1397
Mean Time	216	498	1629
VDTable	0.012	0.016	0.02

Table 5.2 Execution time of BWDM in each specification (ms)

Times	Specification 1	Specification 2	Specification 3
1^{st} time	325	316	6277
2^{nd} time	283	389	5321
3^{rd} time	500	371	6236
4^{th} time	291	334	5070
5^{th} time	269	371	5700
Average	334	356	5720

4.1 Evaluation of VDTable

We evaluated VDTable by measuring and comparing the both of the time to create the decision table by the test engineer manually and by VDTable automatically.

In the specification used in the experiment, the number of combinations of the truth values of the conditions is different. Each number of it are A: 4, specification B: 16, specification C: 256. The testers are five students, A, B, C, D, and E, belonging to the Department of Information Sciences, the University of Miyazaki.

Table 5.1 shows the average of the measured time and the comparison result of VDTable. Testers took about three minutes to create a decision table for the specification A with the smallest combination of truth values of the conditions. On the other hand, in the case of using the VDTable developed in this research, generation of the decision table was completed in about 20 milliseconds for 256 combinations of the truth values of the conditions, even for the specification C. From this, it can be said that VDTable significantly reduced the labor and time of manual create work when creating a decision table using the VDM++ specification. That is, the VDTable is useful when designing a decision table using the VDM++ specification.

4.2 Evaluation of BWDM

To consider BWDM usefulness, we used 3 specifications. We measured the time to generate test cases. Table 5.2 shows conclusion of measuring. Within the three specification, the if-conditional expressions of inequalities are described. The specification 1 has 5 if-conditional expressions, specification 2 has 10 expressions and specification 3 has 15 expressions. To measure execution time, we used System.nanoTime [14] method of Java. We measured the time that is to give the VDM++ specification and the decision table as an input to the BWDM and to finish outputting the test cases. The unit of time was millisecond, and measurements were made five times for each specification, and we calculated the average of 5 measuring as well. Here, we don't consider the time to prepare the VDM++ specification and the decision table.

For the average time in each specification, specification 1 and specification 2 were less than 0.5 seconds, and specification 3 was less than seven seconds. The number of rules on the decision table for specification 1 (number of columns in the decision table) is 32, specification 2 is 1024 and specification 3 is 1048576. These represent the number of states that the function can take, depending on the Boolean value in the if-conditional expression. When constraint conditions such as preconditions are described in the specification, the number of states actually taken by the function is less than this number. However, it is troublesome and time consuming to manually design test cases for functions that can have as many states as possible. On the other hand, BWDM can automatically generate boundary value test cases in several seconds if the specification is up to condition number 15.

Hence, we think that BWDM is usefulness when testing software that implemented from VDM++ specification.

5 Conclusion

In this research, in order to improve the efficiency of the test process in software development using formal methods, we have proposed the method to generate a decision tables and boundary values from VDM++ specifications and implemented two tools: VDTable and BWDM. First, the both methods parsed the VDM++ specification. Thereafter, VDTable extracted conditions and actions from the parsing data, created truth values and then generated a decision table. And BWDM extracted the if-expressions and argument types of function definition, performed boundary value analysis, and finally generated test cases.

As a result of applying the VDM++ specification to both tools, it was confirmed that decision table generation and boundary value analysis can be executed without problems. In comparison with testers, we confirmed that both tools can reduce time and labor than manual work. From this, it can be said that the work efficiency of the test process, which is the object of this research, was achieved. The future works are shown below.

Common task of both tools

- Correspondence to exponential explosion
  - As conditions increase, the number of the test cases and the state of decision table increases by exponentially. As a correspondence, we are thinking about improving the VDTable GUI and adding options to specify test cases to be generated by BWDM.
- Expanding coverage of the tool
  - The definition of VDM++ what supported by our tools is only function defintion. There are many unsupported syntaxes in the two tools. (operation definition, let be, for, while and others) For Resolving this issue, VDTable needs new extraction rule for unsupported syntaxes. In case of BWDM, it needs that new method to generate boundary values from unsupported syntaxes.
- Evaluation by using huge specification that used in real development
  - In this paper, we used specifications whose scale was small, and prepared by us. For future development, application data according to the specification used in actual development will be useful. The opinion on tools by actual SE will be very helpful too.
- Correspondence to other test design techniques
  - In addition to the decision table and boundary value analysis, many test design techniques are known; for example, pairwise testing. Automatic execution of them based on VDM++ specification also has room for further work.

The task of VDTable

- Correspondence to compound conditional expression
  - Compound conditional expression is a combination of two or more conditions with and, or. Although the current VDTable converts the compound conditional expression as it is to the decision table, decomposition into a simple conditional expression makes it easier to understand the specification. We think that it's possible to resolve this issue by splitting into individual simple conditions by logical operators in compound conditional expressions when VDTable extracts conditional expressions.

The task of BWDM

- Implementation of Symbolic Execution
  - At present, processing base on symbolic execution is being implemented in BWDM in order to make C1 coverage 100 % for structures such as if-then-else expression in function definition.

References

1. “How to use Decision Tables”, http://reqtest.com/requirements-blog/a-guide-to-using-decision-tables/, last accessed 2017/7/31
2. S. C. Reid: An empirical analysis of equivalence partitioning, boundary value analysis and random testing. In: Proceedings Fourth International Software Metrics Symposium, pp. 64-73 (1997)
3. Blake Neate: Boundary Value Analysis, University of Wales Swansea (2006)
4. Dines Bjørner and Cliff B. Jones.: The Vienna Development Method: The Meta-Language. In: Computer Science, Lecture Notes, Vol. 61, http://www2.imm.dtu.dk/pubdb/p.php?2256 (1978)
5. Cliff B. Jones.: Systematic Software Development using VDM Second Edition. Prentice Hall International, United States (1995)
6. ISO: Information technology – Programming languages, their environments and system software interfaces – Vienna Development Method – Specification Language – Part 1: Base language (1996)
7. John Fitzgerald, Peter Gorm Larsen, Paul Mukherjee, Nico Plat, and Marcel Verhoef.: Validated Designs for Object-oriented Systems. Springer, New York (2005), ISBN: 1-85233-881-4
8. Tetsuro Katayama, Kenta Nishikawa, Yoshihiro Kita, Hisaaki Yamaba, and Naonobu Okazaki.: Proposal of a Supporting Method to Generate a Decision Table from the Formal Specification. In: Journal of Robotics, Networking and Artificial Life (JRNAL 2014), Vol. 1, No. 3, pp. 174-178 (2014)
9. Kenta Nishikawa, Tetsuro Katayama, Yoshihiro Kita, Hisaaki Yamaba, Kentaro Aburada, and Naonobu Okazaki.: Prototype of a Decision Table Generation Tool from the Formal Specification. In: International Conference on Artificial Life and Robotics 2015 (ICAROB 2015), pp. 388-391 (2015)
10. Hiroki Tachiyama, Tetsuro Katayama, Yoshihiro Kita, Hisaaki Yamaba, and Naonobu Okazaki.: Prototype of Test Cases Automatic Generation tool BWDM Based on Boundary Value Analysis with VDM++. In: International Conference on Artificial Life and Robotics 2017 (ICAROB 2017), pp. 275-278 (2017)
11. “What is Equivalence partitioning in Software testing?”, http://istqbexamcertification.com/what-is-equivalence-partitioning-in-software-testing/, last accessed 2017/7/31
12. Peter W. V. Tran-Jørgensen, Peter Gorm Larsen, and Nick Battle.: Using JML-based Code Generation to Enhance Test Automation for VDM Models. In: Proceedings of the 14th Overture Workshop, pp. 79-93 (2016)
13. Peter Gorm Larsen, Kenneth Lausdahl, Nick Battle.: Combinatorial Testing for VDM. In: 8th IEEE International Conference on Software Engineering and Formal Methods (SEFM 2010)
14. “System (Java Platform SE 7)”, https://docs.oracle.com/javase/7/docs/api/java/lang/System.html, last accessed 2017/7/31