2012 12th International Conference on Quality Software

Function Substitution: Towards Constraint Solving in Software Testing

Xin Feng
Division of Science and Technology
BNU-HKBU United International College
Zhuhai, P. R. China
xinfeng@uic.edu.hk

Simon Marr
Marine Robotics Research Centre
University of Limerick
Limerick, Ireland
simon.marr@ul.ie

Tony O'Callaghan
Interaction Design Centre
University of Limerick
Limerick, Ireland
tony.ocallaghan@ul.ie

Zhi Quan Zhou
School of Computer Science and Software Engineering
University of Wollongong
Wollongong, Australia
zhiquan@uow.edu.au

Jacques Carette
Department of Computing and Software
McMaster University
Hamilton, Canada
carette@mcmaster.ca

***Abstract*—Constraint solving is a fundamental tenet of most test case generation strategies. However, conventional constraint solving methods were not designed in consideration of that aspect of test case generation. It has led to a situation where test case generation techniques can miss test cases when constraints contain function invocations. This is due to the fact that, when solving constraints to generate test cases, function invocations are not effectively handled and, as a result, solutions to some constraints are not found even though they do exist. This problem occurs in both specification-based and code-based testing. To address the problem, this paper presents a function substitution method that transforms test case constraints that contain certain types of functions into equivalent constraints that do not contain those functions. The transformation preserves the solution space and avoids missed test cases. As a result, the completeness of test case generation can be improved.**

***Keywords*-software testing; test case generation; test case constraint; tabular expression; substitution completeness;**

I. Introduction

Testing strategies always give some criteria for selecting test cases. For example, the criterion for the basic branch testing strategy is to execute enough tests to ensure that every branch in a program has been executed at least once under some test [1]. To find the test cases that fulfill a criterion, the criterion must first be represented as a set of test case constraints with respect to a given program. Consider the following code segment: $\{if\ (x > 5)\ return\ 20;\ else\ return\ 15;\}$. For branch coverage, at least two test cases are needed to meet the criterion: one must satisfy $x > 5$ and the other must satisfy $x \leq 5$. We call $x > 5$ and $x \leq 5$ *test case constraints*, and assignments that satisfy the constraints *solutions* (i.e., test cases). Many constraint solvers have been developed to search for solutions automatically. They have been developed as either single-purpose tools or as functionality within a much larger commercial product such as *Maple* (*http://www.maplesoft.com*) and *Mathematica* (*http://www.wolfram.com*). Constraint solving algorithms usually search the domain of the variables in the constraint until a combination of value assignments is found to make the constraint hold *true*. Consider $x > 5$, for instance, a searching algorithm may yield $x = 6$ as a solution to this constraint.

It is common that a program calls other program functions to fulfill a task. Hence, it is typical that function invocations appear in the constraints. If $f(x)$ replaces x in the above example, the constraints then become $f(x) > 5$ and $f(x) \leq 5$. Consider the constraint $f(x) > 5$, for instance. Constraint solvers usually treat embedded functions in a constraint as variables. Tracing the function definition is out of their concerns. A constraint solver will yield, say, $f(x) = 6$ as a solution. To find a test case, we need to trace $f(x)$ to get an assignment of x for $f(x) = 6$. If no solution can be found for $f(x) = 6$ while there actually exists an assignment of x that satisfies $f(x) > 6$, test cases are missed.

A function invocation can sometimes be replaced according to the definition of the input/output relation of that function. If all the functions in a constraint could be replaced properly, the algorithm of finding test cases would be straightforward. The relations between the input and output of a function or a subprogram, however, can be far more complicated: firstly, such relations can be conditional; secondly, a subprogram can use other subprograms. Hence, in most situations, removing a function from a constraint is not as straightforward as replacing $f(x)$ with its function definition such as $x + 1$. The question of how to find test cases from the constraints that contain function invocations has often been ignored or sidestepped in existing test case generation techniques. This problem affects all kinds of code-based and specification-based testing techniques that need to solve constraints involving embedded functions.

1550-6002/12 $26.00 © 2012 IEEE
DOI 10.1109/QSIC.2012.32

The work presented herein addresses this kind of real complexity. To explain the approach, tabular-expression-based specifications [2]–[5] are adopted since tabular expressions are capable of describing complex I/O relations of subprograms, and the notation is both readable and precise. Furthermore, tabular expressions have a proven track record in mission-critical industrial applications, notably the US Navy's A-7 aircraft [6], the Service Evaluation System of Bell Laboratories [7], [8], and the Darlington Nuclear Power Station [9] and other real systems like a Dell keyboard testing program [10] and an Ericsson telecom software system [11], etc.

This paper is organized as follows: Section II reviews related work. Section III introduces tabular expressions and the corresponding testing strategies. Section IV analyzes the cases in which test cases are missed. Section V presents our approach. Section VI shows evaluation results of our approach. Section VII concludes the paper and discusses future research.

II. Related work

In program compilation, there is a technique known as *inline substitution* [12], [13], which replaces a function invocation in source code with a modified version of its function body. It is an optimization technique with the objective of minimizing execution time of the object code. It is not appropriate to apply this technique to unit testing because, firstly, the test obligations for an adequacy criterion before and after the inline substitution can be different; secondly, replacing all levels of function calls can make unit testing too complicated to be practical; and finally, replacing function calls can be difficult in some situations because they can occur not only as single statements but also at any place such as in assignment statements or in conditional expressions.

Another technique of substitution, known as *symbolic execution* [14], has been proposed for symbolic testing and debugging [14], [15], test case generation [16]–[18], verification of program (partial) correctness and program properties [15], [19], invariant generation [20], testing formal specifications [21], and program reduction [22]. It uses symbolic instead of concrete inputs to run a program. Research on symbolic execution and symbolic analysis has been conducted not only in the area of software engineering but also in the area of programming languages and compiler technologies in order to facilitate parallelism and optimization of program code as well as error detection [23], [24]. To analyze programs involving procedures/functions, both static techniques (such as inter-procedural symbolic analysis) and dynamic techniques (such as inter-procedural tracing of symbolic expressions) have been developed. While symbolic execution can be applied to generate white-box test cases, where the program under test involves embedded function invocations, it has similar limitations as those of in-line substitution techniques.

A related technique is testability transformation proposed in [25]. It is a source-code-based transformation that is aimed at improving the testability of a program without weakening the test adequacy criterion. It replaces the control flags in the program. It has been used in evolutionary test case generation. A further substitution technique used in evolutionary test case generation is the genetic algorithm that replaces or exchanges parts of tried test cases [26]. However, none of these substitution techniques take function invocation into consideration.

In this paper the substitution of function invocations for the purpose of test case generation is considered. The proposed algorithm does not intend to find the complete solution to the constraint solving problems in software testing; instead, it is a partial solution that should be seen as a first step in resolving this issue. The substitution is not performed in program source code but in the test case constraints that fulfill certain testing criteria. Finding solutions to the transformed constraints are left to constraint solvers, which is beyond the scope of this paper. This algorithm has been used in ESTP, a testing platform developed in our laboratory to support tabular-expression-based testing [27].

III. Introduction to tabular expressions

This section will describe the characteristics of tabular expressions. It will then examine two different strategies for generating test case constraints from such expressions.

A. Tabular expressions

A *tabular expression* is an indexed set [28] of *grids*, and a grid is an indexed set of *predicate expressions* or *evaluation expressions* [4]. An evaluation expression is evaluated if its corresponding predicate expressions are *true* with respect to an input assignment. Predicate expressions and evaluation expressions can be tabular themselves, i.e., defined in separate tables. Many table types have been defined and can be transformed into each other [4], [29], [30]. In this paper, two types of tables are used: normal and inverted. It is also required that all the programs specified by these tables be deterministic.

1) Normal table: The two-dimensional normal table in Fig. 1 shows the specification of a function that calculates the price of a product according to its type and quantity. An equivalent conventional mathematical expression for this function is also given.

- The program takes two input variables: q (quantity) and t (type), where q is an integer and the domain of t is $\{VA, VC, VE\}$ (denoted by $VTYPE$).
- The specification table contains three grids: $T[0]$, $T[1]$, and $T[2]$. $T[1]$ and $T[2]$ are *header grids* and contain only predicate expressions. $T[0]$ is the *main grid* and contains only evaluation expressions.

- The index sets for $T[1]$ and $T[2]$ are the same, namely $\{0, 1, 2\}$; the index set for $T[0]$ is $\{0, 1, 2\} \times \{0, 1, 2\}$.
- Both $T[1]$ and $T[2]$ must be *proper*, i.e., one and only one predicate expression in each of these grids is evaluated to *true* with respect to an assignment of q and t in their domains.

$T[0][i, j]$ for $0 \leq i \leq 2$ and $0 \leq j \leq 2$ is evaluated with respect to an assignment if and only if both $T[1][i]$ and $T[2][j]$ are evaluated to *true* with respect to that assignment.

2) Inverted table: The table in Fig. 2 is a two-dimensional inverted table showing the specification of a program that calculates the commission according to the sales by a salesperson.

The following features can be observed:

- The program takes four input variables: qa, qc, qe, and r, where qa, qc, and qe (quantity) are integers, and the domain of r is $\{NON\_EU, EU\}$ (denoted by $REGION$).
- The expressions in the grids $T[0]$ and $T[1]$ are predicate expressions. The expressions in the grid $T[2]$ are evaluation expressions.
- $T[1]$ and each row of $T[0]$ must be *proper*.

In $T[2]$, $T[2][j]$ for $0 \leq j \leq 2$ is evaluated only when $T[1][0] \wedge T[0][0, j]$ or $T[1][1] \wedge T[0][1, j]$ is evaluated to *true* with respect to an assignment. The function $Sales$ in Fig. 2 is defined in a conventional mathematical expression, which uses the function $Price$. It itself is used in the table for the function $Commission$.

These two tables together with the other tables in the appendix comprise the specification of program *Level*. This program grades the promotion levels for a salesperson based on sales amount.

B. Tabular-expression-based testing strategies

Two testing strategies have previously been proposed based on tabular expressions, namely the partition method [31] and the interesting point selection strategy [32]. Both generate test cases by exploiting the domain division in the tabular structure.

The partition method requires that each cell in the main grid be tested at least once. The total number of test case constraints for a two-dimensional normal or inverted table is given by the number of defined cells in the main grid. For example, to test the cell $T[0][0, 2]$ in the specification $Commission$, a test case that satisfies the constraint $(r \neq EU) \wedge (Sales(qa, qc, qe) > 4800)$ can be used.

The interesting point selection strategy uses boundary values given in the tables for stress testing. For example, for the specification table in Fig. 2, this strategy can create a test case constraint such as $(r \neq EU) \wedge (Sales(qa, qc, qe) = 4800)$.

In the above examples, the test case constraints created with these two strategies both contain functions. For consistency, we will use the partition method in all further examples in this paper. Table I shows the sets of test case constraints for $Price$ and $Commission$ with the partition method.

IV. Concealed test cases

When a test case constraint contains functions, the following algorithm is a straightforward solution that is used to search for test cases.

(1) Evaluate all the functions with constant parameters and replace them with the evaluation values.
(2) Treat the functions with variable parameters as variables and use a constraint solver to get an assignment to these functions.
(3) Trace the definitions of these functions and find an assignment to the input variables of these functions.
(4) If no assignments of the input variables are found to fulfill the assignment to these functions, go to (2) to try another assignment.

However, as stated below, the algorithm does not execute efficiently and can miss valid test cases. Take the following constraint from Table I as an example: $(r \neq EU) \wedge (Sales(qa, qc, qe) > 4800)$. Since constraint solvers treat the function $Sales(qa, qc, qe)$ as a variable, one solution a constraint solver may give for this constraint is $\langle Sales(qa, qc, qe) = 4801, r = NON\_EU \rangle$. This solution does not actually determine the values of qa, qc, and qe. Further processing is needed to find their values such that $Sales(qa, qc, qe) = 4801$. Since $Sales$ contains the function $Price$, which is defined in another table, we must search that table to find the values of qa, qc, and qe such that $Price(qa, VA) \times qa + Price(qc, VC) \times qc + Price(qe, VE) \times qe = 4801$. Unfortunately, according to the definition of $Price$, such values do not exist. As a result, no assignments to the input variables are found. In other words, test cases are missed. The only way to handle the situation is to assign the function $Sales$ another value by searching for the next solution to the constraint and try again. This process has to be repeated until an assignment is found or no further assignments are possible.

There is also another problem with this algorithm. Consider the following constraint: $(r \neq EU) \wedge (Sales(qa, qc, qe)/2 + Sales(qe+qa-qe, qc, qe)/2 > 4800)$. Since $Sales(qa, qc, qe)$ and $Sales(qe + qa - qe, qc, qe)$ are treated as separate variables, then probably assigned different values, such as $Sales(qa, qc, qe) = 4800$ and $Sales(qe + qa - qe, qc, qe) = 4802$, even though they are equivalent. When that happens, no solution can ever be found. However, test cases do exist for this constraint (e.g., $\langle qa = 30, qc = 60, qe = 99, r = NON\_EU \rangle$).

V. The function substitution method

This section introduces an approach to processing a constraint that contains functions.

$Price(int\ q,\ VTYPE\ t) \equiv$

$T[1]$ \ $T[2]$	$t = VA$	$t = VC$	$t = VE$
$q \leq 30$	20	26	32
$30 < q \leq 60$	18	24	30
$q > 60$	16	22	28

$T[0]$

$$Price(q,t) = \begin{cases} 20 & if\ \ q \leq 30 \wedge t = VA \\ 26 & if\ \ q \leq 30 \wedge t = VC \\ 32 & if\ \ q \leq 30 \wedge t = VE \\ 18 & if\ \ 30 < q \leq 60 \wedge t = VA \\ 24 & if\ \ 30 < q \leq 60 \wedge t = VC \\ 30 & if\ \ 30 < q \leq 60 \wedge t = VE \\ 16 & if\ \ q > 60 \wedge t = VA \\ 22 & if\ \ q > 60 \wedge t = VC \\ 28 & if\ \ q > 60 \wedge t = VE \end{cases}$$

Figure 1. A normal table (Price)

$Commission(int\ qa,\ int\ qc,\ int\ qe,\ REGION\ r) \equiv$

$T[1]$ \ $T[2]$	$Sales(qa,qc,qe) \times 0.1$	$Sales(qa,qc,qe) \times 0.15$	$Sales(qa,qc,qe) \times 0.2$
$r \neq EU$	$Sales(qa,qc,qe) \leq 3000$	$3000 < Sales(qa,qc,qe) \leq 4800$	$Sales(qa,qc,qe) > 4800$
$r = EU$	$Sales(qa,qc,qe) \leq 2800$	$2800 < Sales(qa,qc,qe) \leq 4500$	$Sales(qa,qc,qe) > 4500$

$T[0]$

$$Commission(qa,qc,qe,r) = \begin{cases} Sales(qa,qc,qe) \times 0.1 & if\ r \neq EU \wedge Sales(qa,qc,qe) \leq 3000 \\ & \vee\ r = EU \wedge Sales(qa,qc,qe) \leq 2800 \\ Sales(qa,qc,qe) \times 0.15 & if\ r \neq EU \wedge 3000 < Sales(qa,qc,qe) \leq 4800 \\ & \vee\ r = EU \wedge 2800 < Sales(qa,qc,qe) \leq 4500 \\ Sales(qa,qc,qe) \times 0.2 & if\ r \neq EU \wedge Sales(qa,qc,qe) > 4800 \\ & \vee\ r = EU \wedge Sales(qa,qc,qe) > 4500 \end{cases}$$

$$Sales(int\ qa, int\ qc, int\ qe) = Price(qa, VA) \times qa + Price(qc, VC) \times qc + Price(qe, VE) \times qe$$

Figure 2. An inverted table (Commission)

Table I
THE SETS OF TEST CASE CONSTRAINTS WITH THE PARTITION METHOD

Table	Set of test case constraints with the partition method
$Price(q,t)$	$\{(q \leq 30) \wedge (t = VA), (q \leq 30) \wedge (t = VC), (q \leq 30) \wedge (t = VE), (30 < q \leq 60) \wedge (t = VA), (30 < q \leq 60) \wedge (t = VC), (30 < q \leq 60) \wedge (t = VE), (q > 60) \wedge (t = VA), (q > 60) \wedge (t = VC), (q > 60) \wedge (t = VE)\}$
$Commision(qa,qc,qe,r)$	$\{(r \neq EU) \wedge (Sales(qa,qc,qe) \leq 3000), (r \neq EU) \wedge (3000 < Sales(qa,qc,qe) \leq 4800), (r \neq EU) \wedge (Sales(qa,qc,qe) > 4800), (r = EU) \wedge (Sales(qa,qc,qe) \leq 2800), (r = EU) \wedge (2800 < Sales(qa,qc,qe) \leq 4500), (r = EU) \wedge (Sales(qa,qc,qe) > 4500)\}$

A. Functions in two categories

Let us consider the following conditions of a function in a constraint:

(1) the function is defined either in conventional mathematical expressions or in tabular expressions,
(2) the function is not defined recursively, either directly or indirectly, and
(3) the variables of the function are simple variables, i.e., they do not refer to other expressions.

If a function meets the above conditions, it is a category A function; otherwise it is a category B function. Our approach removes category A functions in constraints by transforming those constraints into equivalent ones that do not contain category A functions. If a constraint contains only category B functions, the approach will not be applied. If a constraint contains both category A and category B functions, then all the category A functions can be removed, and the category B functions will remain.

This approach makes it easier to generate test cases from constraints given that category A functions account for a very large portion of functions in the real world. After applying this approach, the number of functions in constraints will be reduced (in this case the only remaining functions belong to category B) or become zero (in this case no category B functions are involved). In either case, finding solutions to the resultant constraints is simplified. Further discussions on constraint solving, however, are beyond the scope of this paper.

B. Substitution completeness

Let us consider a constraint $f(x) > 5$, where $f(x) = 1000x+5$ and x is an integer. If we do not replace $f(x)$ with its definition $1000x + 5$, the solution to $f(x) > 5$ is usually $f(x) = 6$. However, we cannot find a value of x such that $1000x + 5 = 6$. Hence, we have to try successive solutions, for example, $f(x) = 7,\ 8,\ \ldots$, until we reach $f(x) = 1005$.

However, if we replace $f(x)$ with its definition $1000x + 5$ first, the constraint becomes $1000x + 5 > 5$. Then a solution $x = 1$ is obtained immediately.

In the above example, $f(x)$ is a simple linear function but in practice, function definitions are often more complicated. Let us consider, for instance, the constraint $(r \neq EU) \wedge (Price(qa, VA) \times qa + Price(qc, VC) \times qc + Price(qe, VE) \times qe > 4800)$, where the specification of $Price$ is given in Fig. 1. If we randomly select a combination of three prices for the products VA, VC, and VE from the table shown in Fig. 1 (e.g., prices in the first row of the main grid) to replace the three occurrences of the function $Price$ in the constraint, the constraint then becomes $(r \neq EU) \wedge (20 \times qa + 26 \times qc + 32 \times qe > 4800)$.

To satisfy this constraint, at least one variable qa, qc, or qe must be assigned a value greater than 61. However, the prices 20, 26, and 32 for VA, VC, and VE are applicable only when the quantity of each product is less than or equals 30.

The approach presented herein handles this problem in a different way through function substitution.

Definition 1 (Substitution completeness). *Let **fc** and **tc** be two constraints, where **fc** contains category **A** functions and **tc** is transformed from **fc** by following certain function substitution rules. The substitution of the functions is complete in transforming **fc** to **tc** if and only if:*

(1) ***tc** contains no category **A** functions, and*
(2) ***tc** and **fc** are equivalent, i.e., if **t** is a solution to **tc**, it must be a solution to **fc**, and vice versa.*

Two definitions are given below in the context of tabular-expression-based specifications.

(1) A *formal function* is a table or a conventional mathematical expression in a program specification, similar to a function definition in a programming language. Its syntax is: $f(type_1\ p_1,\ type_2\ p_2,\ \ldots,\ type_n\ p_n)$, where $p_1, p_2, \ldots, p_n$ are *formal parameters*, to which normal variable naming conventions apply.
(2) An *actual function* is one that appears in a test case constraint, similar to a function invocation in a programming language. Its syntax is: $f(a_1,\ a_2, \ldots, a_n)$, where a_1, a_2, …, a_n are *actual parameters*, a_i with $1 \leq i \leq n$ can be
    a) a constant,
    b) a variable,
    c) an actual function, or
    d) an expression that comprises constant(s), variable(s), and/or actual function(s).

C. Function substitution

In this paper, a *test case constraint* is defined as a logical expression that comprises *simple predicates*. The definition of simple predicate in this paper is an extension of that given by Tai [33]. A simple predicate is

1) a Boolean variable or its negation,
2) a Boolean valued function or its negation, or
3) a relational expression of the form $e_1\ \mathbf{op_1}\ e_2\ \mathbf{op_2}\ \ldots\ \mathbf{op_{n-1}}\ e_n$, where $n > 1$, op_i, $i = 1, 2, \ldots, n-1$, is a relational operator, and e_j, $j = 1, 2, \ldots, n$, is an arithmetic expression involving constants, variables, and/or actual functions.

We use the same mathematical representations as in tabular expressions so op_i can be '$<$', '$=$', '$>$', '$\leq$', '$\geq$', or '$\neq$'.

If all simple predicates are represented as Boolean variables, then a test case constraint is considered to be a Boolean expression. Usually, the solutions to a constraint are found in two steps: first, determine the truth values of these Boolean variables such that the Boolean expression is evaluated to *true*; then for a combination of truth values, find the assignments to those variables in the simple predicates. If the function substitution in each simple predicate is complete, the function substitution in the test case constraint is complete.

Let s and m be non-negative integers with $s \geq 0$ and $m \geq 1$. We use $R(v_1, v_2, \ldots, v_s, f_1, f_2, \ldots, f_m)$ to denote a simple predicate involving variables and category A functions, where

- v_1, v_2, …, v_s are variables excluding those appearing in the actual parameters of the functions,
- f_1, f_2, …, f_m are the occurrences of all the category A functions,
- f_k and $f_{k'}$ for $1 \leq k, k' \leq m$ and $k \neq k'$ can be duplicated, and f_k can also be a function with only constant actual parameters.

The predicate after the transformation is denoted by $R(v_1, v_2, \ldots, v_s, \ldots, v_{s'})$, where $s' \geq 1$, v_1, v_2, …, $v_{s'}$ are all the variables in the predicate.

To satisfy the first condition of substitution completeness, the substitutions starting from $R(v_1,\ v_2,\ \ldots,\ v_s,\ f_1,\ f_2,\ \ldots,\ f_m)$ must end at a predicate of the form $R(v_1, v_2, \ldots, v_s, \ldots, v_{s'})$. Under the assumption that no category A functions are defined recursively, function substitutions can be repeated until no category A functions remain. To satisfy the second condition, $R(v_1, v_2, \ldots, v_s, f_1, f_2, \ldots, f_m)$ and $R(v_1, v_2, \ldots, v_s, \ldots, v_{s'})$ must be equivalent.

Let $R(v_1, v_2, \ldots, v_s, o, f_2, \ldots, f_m)$ denote the simple predicate obtained from $R(v_1, v_2, \ldots, v_s, f_1, f_2, \ldots, f_m)$ with f_1 replaced with o. Then we have Theorem 1.

Theorem 1. *$R(v_1, v_2, \ldots, v_s, f_1, f_2, \ldots, f_m)$ is equivalent to $(R(v_1, v_2, \ldots, v_s, o_1^1, f_2, \ldots, f_m) \wedge c_1^1) \vee (R(v_1,\ v_2,\ \ldots,\ v_s,\ o_1^2,\ f_2, \ldots, f_m) \wedge c_1^2) \vee \cdots \vee (R(v_1, v_2, \ldots, v_s, o_1^n, f_2, \ldots, f_m) \wedge c_1^n)$ if the equivalent conventional mathematical expression of f_1 is*

$$f_1 = \begin{cases} o_1^1 & \textit{if condition } c_1^1 \textit{ holds true} \\ o_1^2 & \textit{if condition } c_1^2 \textit{ holds true} \\ \vdots & \vdots \\ o_1^n & \textit{if condition } c_1^n \textit{ holds true} \end{cases}$$

where o_1^k *and* c_1^k *for* $1 \leq k \leq n$ *in* $(R(v_1, v_2, \ldots, v_s, o_1^1, f_2, \ldots, f_m) \wedge c_1^1) \vee (R(v_1, v_2, \ldots, v_s, o_1^2, f_2, \ldots, f_m) \wedge c_1^2) \vee \cdots \vee (R(v_1, v_2, \ldots, v_s, o_1^n, f_2, \ldots, f_m) \wedge c_1^n)$ *are* o_1^k *and* c_1^k *in* f_1 *with its formal parameters replaced by actual parameters.*

Proof: To prove $R(v_1, v_2, \ldots, v_s, f_1, f_2, \ldots, f_m)$ is equivalent to $(R(v_1, v_2, \ldots, v_s, o_1^1, f_2, \ldots, f_m) \wedge c_1^1) \vee (R(v_1, v_2, \ldots, v_s, o_1^2, f_2, \ldots, f_m) \wedge c_1^2) \vee \cdots \vee (R(v_1, v_2, \ldots, v_s, o_1^n, f_2, \ldots, f_m) \wedge c_1^n)$, we need to show that a solution to either is a solution to both.

Assume t is a solution to $R(v_1, v_2, \ldots, v_s, f_1, f_2, \ldots, f_m)$, that is, R is *true* with respect to t. Since f_1 is a function in the constraint, f_1 must be evaluated to a value with respect to t. Therefore, t must evaluate one of $c_1^1, c_1^2, \ldots, c_1^n$ to *true* (otherwise f_1 is not defined). Let us assume c_1^k with $1 \leq k \leq n$ holds *true* with respect to t, then because the corresponding output of f_1 for c_1^k is o_1^k, $R(v_1, v_2, \ldots, v_s, o_1^k, f_2, \ldots, f_m)$ must hold *true*. Hence, $R(v_1, v_2, \ldots, v_s, o_1^k, f_2, \ldots, f_m) \wedge c_1^k$ also holds *true*. This means that $(R(v_1, v_2, \ldots, v_s, o_1^1, f_2, \ldots, f_m) \wedge c_1^1) \vee \cdots \vee (R(v_1, v_2, \ldots, v_s, o_1^k, f_2, \ldots, f_m) \wedge c_1^k) \vee \cdots \vee (R(v_1, v_2, \ldots, v_s, o_1^n, f_2, \ldots, f_m) \wedge c_1^n)$ must evaluate to *true*. In other words, t is a solution to $(R(v_1, v_2, \ldots, v_s, o_1^1, f_2, \ldots, f_m) \wedge c_1^1) \vee \cdots \vee (R(v_1, v_2, \ldots, v_s, o_1^k, f_2, \ldots, f_m) \wedge c_1^k) \vee \cdots \vee (R(v_1, v_2, \ldots, v_s, o_1^n, f_2, \ldots, f_m) \wedge c_1^n)$.

If $(R(v_1, v_2, \ldots, v_s, o_1^1, f_2, \ldots, f_m) \wedge c_1^1) \vee (R(v_1, v_2, \ldots, v_s, o_1^2, f_2, \ldots, f_m) \wedge c_1^2) \vee \cdots \vee (R(v_1, v_2, \ldots, v_s, o_1^n, f_2, \ldots, f_m) \wedge c_1^n)$ is *true* with respect to t, one or more disjunctive terms in this constraint must be *true* with respect to *t*. Since it is assumed that the program specified by the table is deterministic, only one of the conditions c_1^1, c_1^2, $\ldots$, and c_1^n can be evaluated to *true* at any one time. Hence, the disjunctive terms in the above constraint cannot be *true* simultaneously. Only one term can be *true*. If we let this term be $(R(v_1, v_2, \ldots, v_s, o_1^k, f_2, \ldots, f_m) \wedge c_1^k)$ with $1 \leq k \leq n$, then because o_1^k is the output of f_1 with respect to the assignment that satisfies c_1^k, t also satisfies $R(v_1, v_2, \ldots, v_s, f_1, f_2, \ldots, f_m)$. In other words, t is a solution to $R(v_1, v_2, \ldots, v_s, f_1, f_2, \ldots, f_m)$. ■

The following example illustrates the above theorem. Assume $R(v_1, v_2, \ldots, v_s, f_1, f_2, \ldots, f_m)$ is $20 < Price(q_1, t_1) < 30$. Hence, f_1 is $Price(q_1, t_1)$, $m = 1$, and $s = 0$. It is known from Fig. 1 that there are nine evaluation expressions (i.e., $n = 9$) and o_1^1, o_1^2, ..., o_1^9 are 20, 26, 32, 18, 24, 30, 16, 22, and 28, respectively. $(R(v_1, v_2, \ldots, o_1^1, f_2, \ldots, f_m) \wedge c_1^1) \vee (R(v_1, v_2, \ldots, o_1^2, f_2, \ldots, f_m) \wedge c_1^2) \vee \ldots \vee (R(v_1, v_2, \ldots, o_1^n, f_2, \ldots, f_m) \wedge c_1^n)$ is then $(R(o_1^1) \wedge c_1^1) \vee (R(o_1^2) \wedge c_1^2) \vee \cdots \vee (R(o_1^9) \wedge c_1^9)$, i.e., $((20 < 20 < 30) \wedge (q_1 \leq 30) \wedge (t_1 = VA)) \vee ((20 < 26 < 30) \wedge (q_1 \leq 30) \wedge (t_1 = VC)) \vee ((20 < 32 < 30) \wedge (q_1 \leq 30) \wedge (t_1 = VE)) \vee ((20 < 18 < 30) \wedge (30 < q_1 \leq 60) \wedge (t_1 = VA)) \vee ((20 < 24 < 30) \wedge (30 < q_1 \leq 60) \wedge (t_1 = VC)) \vee ((20 < 30 < 30) \wedge (30 < q_1 \leq 60) \wedge (t_1 = VE)) \vee ((20 < 16 < 30) \wedge (q_1 > 60) \wedge (t_1 = VA)) \vee ((20 < 22 < 30) \wedge (q_1 > 60) \wedge (t_1 = VC)) \vee ((20 < 28 < 30) \wedge (q_1 > 60) \wedge (t_1 = VE))$. This constraint can be simplified to $((q_1 \leq 30) \wedge (t_1 = VC)) \vee ((30 < q_1 \leq 60) \wedge (t_1 = VC)) \vee ((q_1 > 60) \wedge (t_1 = VC)) \vee ((q_1 > 60) \wedge (t_1 = VE))$. It is equivalent to $20 < Price(q_1, t_1) < 30$.

Theorem 1 specifies the substitution of a single function in a constraint. In practice, a constraint may contain more than one function, and a function definition may involve multiple levels of nested functions (see the function $Level$ in the appendix). However, by repeatedly applying Theorem 1 to every simple predicate in the constraint, the functions can be removed one by one until no more category A functions remain.

Corollary 1. *If a constraint contains two duplicated functions, the constraint obtained by replacing the duplicated functions separately is equivalent to that with the duplicated functions replaced simultaneously.*

Proof: Assume f_1 and f_w with $1 < w \leq m$ are duplicates in the constraint $R(v_1, v_2, \ldots, v_s, f_1, f_2, \ldots, f_w, \ldots, f_m)$ and they have the same definition as f_1 in Theorem 1.

First, we replace these two functions separately. Assume f_1 is replaced first. After f_1 is replaced, the constraint is transformed to $(R(v_1, v_2, \ldots, v_s, o_1^1, f_2, \ldots, f_w, \ldots, f_m) \wedge c_1^1) \vee (R(v_1, v_2, \ldots, v_s, o_1^2, f_2, \ldots, f_w, \ldots, f_m) \wedge c_1^2) \vee \cdots \vee (R(v_1, v_2, \ldots, v_s, o_1^n, f_2, \ldots, f_w, \ldots, f_m) \wedge c_1^n)$, where o_1^1, o_1^2, $\ldots$, o_1^n are the evaluation expressions and c_1^1, c_1^2, $\ldots$, c_1^n are the corresponding conditions for each evaluation expression. To then replace f_w in $(R(v_1, v_2, \ldots, v_s, o_1^1, f_2, \ldots, f_w, \ldots, f_m) \wedge c_1^1) \vee (R(v_1, v_2, \ldots, v_s, o_1^2, f_2, \ldots, f_w, \ldots, f_m) \wedge c_1^2) \vee \cdots \vee (R(v_1, v_2, \ldots, v_s, o_1^n, f_2, \ldots, f_w, \ldots, f_m) \wedge c_1^n)$, we need to replace f_w in each disjunctive term $R(v_1, v_2, \ldots, v_s, o_1^k, f_2, \ldots, f_w, \ldots, f_m) \wedge c_1^k$ for $1 \leq k \leq n$. Since f_1 and f_w are duplicates, we can use the same specification for f_1 to replace f_w. Therefore, after f_w is replaced, the constraint $R(v_1, v_2, \ldots, v_s, o_1^k, f_2, \ldots, f_w, \ldots, f_m) \wedge c_1^k$, $1 \leq k \leq n$, becomes $(R(v_1, v_2, \ldots, v_s, o_1^k, f_2, \ldots, o_1^1, \ldots, f_m) \wedge c_1^k \wedge c_1^1) \vee (R(v_1, v_2, \ldots, v_s, o_1^k, f_2, \ldots, o_1^2, \ldots, f_m) \wedge c_1^k \wedge c_1^2) \vee \cdots \vee (R(v_1, v_2, \ldots, v_s, o_1^k, f_2, \ldots, o_1^k, \ldots, f_m) \wedge c_1^k \wedge c_1^k) \vee \cdots \vee (R(v_1, v_2, \ldots, v_s, o_1^k, f_2, \ldots, o_1^n, \ldots, f_m) \wedge c_1^k \wedge c_1^n)$ for $1 \leq k \leq n$. Since $c_1^i \wedge c_1^j = false$ for $i \neq j$ and $c_1^i \wedge c_1^i = c_1^i$, the above constraint can be simplified as $R(v_1, v_2, \ldots, v_s, o_1^k, f_2, \ldots, o_1^k, \ldots, f_m) \wedge c_1^k$ for $1 \leq k \leq n$. Hence, the resultant constraint is $(R(v_1, v_2, \ldots, v_s, o_1^1, f_2, \ldots, o_1^1, \ldots, f_m) \wedge c_1^1) \vee (R(v_1, v_2, \ldots, v_s, o_1^2, f_2, \ldots, o_1^k, \ldots, f_m) \wedge c_1^2) \vee \cdots \vee (R(v_1, v_2, \ldots, v_s, o_1^n, f_2, \ldots, o_1^n, \ldots, f_m) \wedge c_1^n)$.

If we replace f_1 and f_w simultaneously in $R(v_1, v_2, \ldots, v_s, f_1, f_2, \ldots, f_m)$, by applying Theorem 1, we can directly get the resultant constraint $(R(v_1, v_2, \ldots, v_s, o_1^1, f_2, \ldots, o_1^1, \ldots, f_m) \wedge c_1^1) \vee \cdots \vee (R(v_1, v_2, \ldots, v_s, o_1^k, f_2, \ldots, o_1^k, \ldots, f_m) \wedge c_1^k) \vee \ldots \vee (R(v_1, v_2, \ldots, v_s, o_1^n, f_2, \ldots, o_1^n, \ldots, f_m) \wedge c_1^n)$. ■

The following example illustrates that the predicates obtained from simultaneous substitutions and from sequential substitutions are equivalent. Consider a constraint $2 \times f(x) < f(x) \times g(x)$. Here $f(x)$ occurs twice. Suppose the outputs of

$f(x)$ are o_1 and o_2 under conditions c_1 and c_2, respectively. If the two occurrences of $f(x)$ are replaced simultaneously, we obtain $((2 \times o_1 < o_1 \times g(x)) \wedge c_1) \vee ((2 \times o_2 < o_2 \times g(x)) \wedge c_2)$. If the two occurrences of $f(x)$ are replaced sequentially, after the first substitution we get $((2 \times o_1 < f(x) \times g(x)) \wedge c_1) \vee ((2 \times o_2 < f(x) \times g(x)) \wedge c_2)$; and after the second substitution we get $((2 \times o_1 < o_1 \times g(x)) \wedge c_1 \wedge c_1) \vee ((2 \times o_1 < o_2 \times g(x)) \wedge c_2 \wedge c_1) \vee ((2 \times o_2 < o_1 \times g(x)) \wedge c_1 \wedge c_2) \vee ((2 \times o_2 < o_2 \times g(x)) \wedge c_2 \wedge c_2)$, which is reduced to $((2 \times o_1 < o_1 \times g(x)) \wedge c_1) \vee ((2 \times o_2 < o_2 \times g(x)) \wedge c_2)$ (since $c_2 \wedge c_1$ is *false*).

Corollary 2. *If the actual parameters in a function are all constants, they can be taken as variables in the substitution without the loss of test cases.*

Proof: Assume f_1 has the same definition as that in Theorem 1 and this function has only constant parameters in the constraint $R(v_1, v_2, \ldots, v_s,\ f_1, f_2, \ldots, f_m)$. If the constant parameters evaluates c_1^k with $1 \leq k \leq n$ *true*, then the value of f_1 is o_1^k. If f_1 is replaced with o_1^k, the resultant constraint is $R(v_1, v_2, \ldots, v_s, o_1^k, f_2, \ldots, f_m)$. If we substitute the function according to Theorem 1, the resultant constraints is $(R(v_1, v_2, \ldots, v_s, o_1^1,\ f_2, \ldots, f_m) \wedge c_1^1) \vee\ (R(v_1, v_2, \ldots, v_s, o_1^2, f_2,\ \ldots, f_m) \wedge c_1^2) \vee \cdots \vee (R(v_1, v_2, \ldots, v_s, o_1^n, f_2, \ldots, f_m) \wedge c_1^n)$ where c_1^1, $\ldots$, c_1^k, $\ldots$, c_1^n can be evaluated with the constant parameters. Since the constant parameters evaluate c_1^k to *true* and all others to *false*, this constraint is simplified as $R(v_1, v_2, \ldots, v_s, o_1^k, f_2, \ldots, f_m)$, which is the same as the constraint where the function is directly evaluated. ■

If all the actual parameters of a function are either constants or constant expressions, the function itself can be evaluated and replaced with a constant. The result thus obtained is equivalent to that using Theorem 1, where constant parameters are handled in the same way as variable parameters. Consider, for instance, the example $Price(20, VA) < 19$. Let us directly evaluate this function first. With $\langle q = 20, t = VA \rangle$, $T[1][0]$ and $T[2][0]$ (see Fig. 1) will be satisfied, so we get $Price(20, VA) = 20$. Hence, $Price(20, VA) < 19$ is reduced to $20 < 19$. Using Theorem 1, the same result can be obtained as follows. The predicate after the substitution of $Price(20, VA)$ is $((20 < 19) \wedge (20 < 30) \wedge (VA = VA)) \vee ((18 < 19) \wedge (30 \leq 20 < 60) \wedge (VA = VA)) \vee ((16 < 19) \wedge (20 \geq 60) \wedge (VA = VA)) \vee ((26 < 19) \wedge (20 < 30) \wedge (VA = VC)) \vee ((24 < 19) \wedge (30 \leq 20 < 60) \wedge (VA = VC)) \vee ((22 < 19) \wedge (20 \geq 60) \wedge (VA = VC)) \vee ((32 < 19) \wedge (20 < 30) \wedge (VA = VE)) \vee ((30 < 19) \wedge (30 \leq 20 < 60) \wedge (VA = VE)) \vee ((28 < 19) \wedge (20 \geq 60) \wedge (VA = VE))$. Since "$VA = VC$", "$VA = VE$", "$30 \leq 20 < 60$", and "$20 \geq 60$" are all *false*, and "$20 < 30$" and "$VA = VA$" are both *true*, the above expression is reduced to $20 < 19$.

If the relationship of input and output is not conditional, the function definition can directly substitute for the actual function. Consider the example in Fig. 2. The function $Sales(qa,\ qc,\ qe)$ in the constraints can be directly replaced by $Price(qa, VA) \times qa + Price(qc, VC) \times qc + Price(qe, VE) \times qe$.

VI. Effectiveness and efficiency

The development team at the Software Quality Research Laboratory developed a testing platform (ESTP) supporting tabular-expression-based testing [27]. This platform consists of three tools: a test data generator, a mutant generator, and a test report analyzer. Two third-party software systems were used: Maple 11.0 from Maple Computer Algebra System Inc. and BoNus 2.4 from the Chinese Academy of Science. The function substitution method presented in this paper was used in this platform. Using the platform, we were able to perform some experiments to verify the effectiveness and efficiency of this method.

A. Effectiveness

In this section, the effectiveness of the algorithm, when applied to the examples in this paper, is discussed. Since the $Price$ function does not use other functions, it was ignored here. For the functions $Commission$, $Bonus$ and $Level$, their implementations were tested using three testing methods: the partition method [31], decision table-based testing [34], and the basic meaningful impact strategy [35]. These testing methods have been extended to generate test cases based on tabular expressions [36]. Details of these testing methods are omitted in this paper since the methods themselves have no impact on the effectiveness of the algorithm.

In this experiment, for every testing method, we created two sets of test cases for each function based on their tabular specifications. One set was created without function transformation; another was created after the constraints were transformed. Both sets were then used to test the implementations from the tabular specifications. *Mutation analysis* was used to compare the fault-detection effectiveness before and after the function transformations. A *mutation operator* describes a kind of syntactic change to a program. The mutant generator in the ESTP platform implemented 20 mutant operators (Table II). It generates mutants by applying these mutant operators. If a mutant produces a different result from the original program for a test case, this mutant is called a *killed mutant*. A *mutation score* is defined as the number of killed mutants divided by the *number of non-equivalent mutants*. This definition is used to verify the effectiveness of an individual testing method. However, the objective of this experiment was to compare the effectiveness of two test case sets. The denominators in the mutation scores are the same for both, regardless of whether the number of non-equivalent mutants or the total number of mutants was used. Therefore, in our

Table II
MUTATION OPERATORS

Name	Definition
OCOR	Cast operator replacement or type replacement
SMVB	Move a brace up or down
SSOM	Exchange the sequence of the statements in the same level
SSDL	Delete a simple statement
SCBR	Replace "break" by "continue" or replace "continue" by "break"
SCBM	Remove "continue" or "break" to the outer or inner level
SICC	Insert semicolon after "if", "while", or "for"
SSCB	Delete or add the "break" in "switch" statement
EARA	Replace an arithmetic assignment operator "$+=$", "$-=$", "$=$", "$\&=$", "$<<=$", "$\mid=$", "$*=$" by another legal assignment operator
EORO	Replace a binary operator by another legal operator
EVRV	Replace a variable by another variable of the same type
ERRV	Replace a reference by a variable of the same type
EVRC	Replace a variable or a constant by a positive value, a negative value, and 0. If it is a string constant, replace it by a constant string and an empty string
EURU	Replace a unary operator by another unary operator
EADV	Add or delete a variable
EADP	Add or delete a pair of parenthesis in an arithmetic expression
EADU	Add or remove a unary operator
EACE	Add a positive constant and a negative constant to the end of an expression
ELCN	Negate the whole logical expression
EEAI	Exchange the index of an array with multiple dimensions

experiments, a *mutation score* was defined as the number of killed mutants divided by the *total number of mutants* (rather than non-equivalent mutants). This revised definition does not affect the comparison results. Table III shows the numbers of mutants created for these three functions using the mutant generator.

Table III
NUMBER OF MUTANTS FOR EACH FUNCTION

Function	Number of Mutants
Commission	333
Bonus	147
Level	252

Our experiments showed that none of the mutation scores were changed after the transformations for $Bonus$ and $Level$. For the $Commission$ function, however, the mutation scores of all three testing methods increased significantly after the transformations (Table IV). This owed to the fact that the test cases that were missed before the transformation were found after the transformations. Among the three testing methods, the basic meaningful impact strategy is considered to be the strongest with the partition method being the weakest. However, the advantage of the basic meaningful impact strategy over the partition method was not clear prior to the transformation.

B. Efficiency

In addition to the "test cases missing" problem, treating functions as variables can cause several other problems. Firstly, a tool is needed to check if a function has constant parameters. If all the parameters are constants, the function cannot be treated as a variable. Secondly, an expression evaluator is required to evaluate the functions with constant parameters. Thirdly, since two equivalent functions must take the same value, a semantic checking tool is required to find equivalent functions. Unfortunately, it is for practical purposes impossible in most cases to determine whether two functions have equivalent actual parameters. Without semantic checking, a constraint solver can assign two equivalent functions different values. The process of searching input values of the functions cannot succeed as it is impossible to produce different outputs for two equivalent functions with the same inputs.

These problems are avoided in the implementation of our approach, where duplicated functions are replaced independently, and functions with constant actual parameters are handled in the same way as other functions. This greatly simplifies the implementation, and the mechanics of function substitution is reduced to pure text processing. No expression evaluator or semantic checker is needed. The transformation itself is independent of other tools or techniques. Although the resultant constraints can be long, further simplifications are usually possible. Using the *evalb* and *BooleanSimplify* logic functions supplied by *Maple*, long constraints can be considerably shortened. For example, the long constraint obtained from $Price(20, VA) < 19$ was simplified to *false* after being processed through *Maple*.

In most cases, the time needed for the simplification of the constraints was acceptable. Initially there was an exponential growth in the execution time of the algorithm that was related to an increase in primitive constraints. For example, on an *iMac* with a 2 GHz *PowerPC G5* CPU and 2GB RAM, it took less than 1 second to process a constraint involving 10 primitive predicates, 4 minutes to process a constraint involving 271 primitive predicates, and 4 hours to process a constraint involving 910 primitive predicates. However, an optimization of the *Maple* code reduces the processing time for the constraint involving 910 primitive predicates to 21 seconds. The sizes of the constraint expressions and the complexity of the embedded functions shown in the case study are indeed comparable to those of medium-sized real-world specifications. This means that the method is feasible for real-world software.

VII. CONCLUSION AND FUTURE WORK

This paper discussed the practical issues of finding solutions to test case constraints that contain functions as well as variables. These issues have been largely ignored in the

Table IV
MUTATION SCORES BEFORE AND AFTER TRANSFORMATIONS (COMMISSION)

Testing method	Before transformation	After transformation
The partition method	0.177	0.649
Decision table-based testing	0.177	0.703
The basic meaningful impact strategy	0.177	0.763

past but they are always present in test case generation. They should be considered regardless of the testing strategy used, otherwise the effectiveness and efficiency of the testing strategy can be compromised. It is for this reason that we have developed a function substitution method that 1) preserves the space of constraint solutions after transformation, 2) does not require semantic checking, and 3) uses only text processing tools. Although this technique can generate long constraints, such constraints can be greatly simplified through the use of commercial tools. The substitution technique can be incorporated into many testing methods even if they are not based on tabular expressions (e.g., [37]). This approach helps improve the completeness of test cases and, hence, enhance confidence in the test results. Although the focus of this research is on unit testing, the approach can be readily applied to other levels of testing.

The execution time of the presented technique can be greatly improved through *Maple* code optimization. It is expected that more problems can be met when the system under test becomes larger. Further improvement can be made by working closely with Maple's technical team. The improvement helps apply the method to programs that have more complicated and deeper levels of function invocations. Moreover, we have started research on the processing of test case constraints for non-deterministic programs.

APPENDIX

Fig. 3 and Fig. 4 are the specifications of $Bonus$ and $Level$, respectively.

$Bonus(int\ c,\ REGION\ r) \equiv$

	$T[2]$		
	$c < 1000$	$1000 \leq c < 1500$	$c \geq 1500$
$r \neq EU$	0	$c \times 1.5\%$	$c \times 2\%$
$r = EU$	0	30	50
$T[1]$	$T[0]$		

Figure 3. Specification of Bonus

ACKNOWLEDGEMENTS

This research was supported by UIC under grant UIC2010-S-01.16 and the Science Foundation Ireland under grants 01/P1.2/C009 and 03/CE3/1405. Zhou was supported in part by a linkage grant of the Australian Research Council (Project ID: LP100200208).

REFERENCES

[1] M. Pezzè, M. Young, *Software Testing and Analysis: Process, Principles, and Techniques*, Wiley, New York, NY, 2008.

[2] D. L. Parnas, J. Madey, M. Iglewski, Precise documentation of well-structured programs, *IEEE Transactions on Software Engineering* 20 (12), pp. 948–976, 1994.

[3] D. L. Parnas, J. Madey, Functional documents for computer systems, *Science of Computer Programming* 25 (1), pp. 41–61, 1995.

[4] R. Janicki, D. L. Parnas, J. Zucker, Tabular representations in relational documents, in D. M. Hoffman and D. M. Weiss (Eds.), *Software Fundamentals: Collected Papers by David L. Parnas*, Addison-Wesley, 2001.

[5] R. Janicki, A. Wassyng, Tabular expressions and their relational semantics, *Fundamenta Informaticae* 67 (4), pp. 343–370, 2005.

[6] K. Heninger, Specifying software requirements for complex systems: new techniques and their application, *IEEE Transactions on Software Engineering* SE-6 (1), pp. 2–13, 1980.

[7] S. D. Hester, D. L. Parnas, D. F. Utter, Using documentation as a software design medium, *The Bell System Technical Journal* 60 (8), pp. 1941–1977, 1981.

[8] D. L. Parnas, S. A. Vilkomir, Precise documentation of critical software, in *Proceedings of the 10th IEEE High Assurance Systems Engineering Symposium (HASE'07)*, pp. 237–244, 2007.

[9] D. L. Parnas, G. J. K. Asmis, J. Madey, Assessment of safety-critical software in nuclear power plants, *Nuclear Safety* 32 (2), pp.189–198, 1991.

[10] R. Baber, D. L. Parnas, S. Vilkomir, P. Harrison, T. O'Connor, Disciplined methods of software specifications: a case study, in *Proceedings of the International Conference on Information Technology Coding and Computing (ITCC 2005)*, 2006.

[11] C. Quinn, S. A. Vilkomir, D. L. Parnas, S. Kostic, Specification of software component requirements using the trace function method, in *Proceedings of the International Conference on Software Engineering Advances (ICSEA)*, 2006.

[12] R. W. Scheifler, An analysis of inline substitution for a structured programming language, *Communications of the ACM* 20 (9), pp. 647–654, 1977.

[13] D. R. Chakrabarti, S.-M. Liu, Inline analysis: Beyond selection heuristics, in *Proceedings of the International Symposium on Code Generation and Optimization (CGO'06)*, 2006.

$Level(int\ qa,\ int\ qc,\ int\ qe,\ REGION\ r) \equiv$

$T[1]$	$T[2]$: $Bonus(Commission(qa, qc, qe), r) < 30$	$T[2]$: $30 \leq Bonus(Commission(qa, qc, qe), r) < 50$	$T[2]$: $Bonus(Commission(qa, qc, qe), r) \geq 50$
$r \neq EU$	0	0	1
$r = EU$	0	1	2

$T[0]$

Figure 4. Specification of Level

[14] J. C. King, Symbolic execution and program testing, *Communications of the ACM* 19 (7), pp. 385–394, 1976.

[15] T. Y. Chen, T. H. Tse, Z. Q. Zhou, Semi-proving: An integrated method for program proving, testing, and debugging, *IEEE Transactions on Software Engineering* (to appear).

[16] L. A. Clarke, A system to generate test data and symbolically execute programs, *IEEE Transactions on Software Engineering* 2 (3), pp. 215–222, 1976.

[17] S. Khurshid, C. S. Păsăreanu, W. Visser, Generalized symbolic execution for model checking and testing, in *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2003)*, 2003.

[18] K. Sen, D. Marinov, G. Agha, CUTE: A concolic unit testing engine for C, in *Proceedings of the Joint 10th European Software Engineering Conference and 13th ACM SIGSOFT International Symposium on Foundation of Software Engineering (ESEC 2005/FSE-13)*, pp. 263–272, 2005.

[19] L. K. Dillon, Using symbolic execution for verification of Ada tasking programs, *ACM Transactions on Programming Languages and Systems* 12 (4), pp. 643–669, 1990.

[20] Y. Kannan, K. Sen, Universal symbolic execution and its application to likely data structure invariant generation, in *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'08)*, pp. 283–294, 2008.

[21] J. Douglas, R. A. Kemmerer, Aslantest: A symbolic execution tool for testing Aslan formal specifications, in *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'94)*, pp. 15–27, 1994.

[22] A. Coen-Porisini, F. De Paoli, C. Ghezzi, D. Mandrioli, Software specialization via symbolic execution, *IEEE Transactions on Software Engineering* SE-17 (9), pp. 884–899, 1991.

[23] M. P. Gerlek, E. Stoltz, M. Wolfe, Beyond induction variables: Detecting and classifying sequences using a demand-driven SSA form, *ACM Transactions on Programming Languages and Systems* 17 (1), pp. 85–122, 1995.

[24] B. Burgstaller, B. Scholz, J. Blieberger, Symbolic analysis of imperative programming languages, in *Proceedings of the 7th Joint Modular Languages Conference (JMLC'06)*, Lecture Notes in Computer Science Vol. 4228, Springer, pp. 172–194, 2006.

[25] M. Harman, L. Hu, R. Hierons, J. Wegener, H. Sthamer, A. Baresel, M. Roper, Testability transformation, *IEEE Transactions on Software Engineering* 30 (1), pp. 3–16, 2004.

[26] C. C. Michael, G. MxGraw, M. A. Achatz, Generating software test data by evolution, *IEEE Transactions on Software Engineering* 27 (12), pp. 1085–1110, 2001.

[27] X. Feng, S. Marr, T. O'Callaghan, ESTP: An experimental software testing platform, in *Proceedings of Testing: Academic and Industrial Conference - Practice and Research Techniques (TAIC-PART'08)*, pp. 59–63, 2008.

[28] P. R. Halmos, *Naive Set Theory*, Princeton, NJ: D. Van Nostrand Company, 1960, reprinted by Springer-Verlag, NY, 1974.

[29] H. Shen, *Implementation of Table Inversion Algorithms*, Master's thesis, McMaster University, Canada, 1995.

[30] T. Fu, Structured decision table $\Leftrightarrow$ generalized decision table conversion tool, SERG Report 380, 1999.

[31] S. Liu, *Generating Test Cases from Software Documentation*, Master's thesis, McMaster University, Canada, 2001.

[32] M. Clermont, D. L. Parnas, Using information about functions in selecting test cases, in *Proceedings of the ICSE Workshop on Advances in Model-Based Software Testing (A-MOST)*, 2005.

[33] K.-C. Tai, Theory of fault-based predicate testing for computer programs, *IEEE Transactions on Software Engineering* 22 (8), pp. 552–562, 1996.

[34] P. C. Jorgensen, *Software Testing: Craftsman's Approach*. Boca Raton, FL: Auerbach Publications, 2008.

[35] E. J. Weyuker, More experience with data flow testing, *IEEE Transactions on Software Engineering*, 19 (9), pp. 912–919, 1993.

[36] X. Feng, D. L, Parnas, T. H. Tse, A comparison of tabular expression-based testing strategies, *IEEE Transactions on Software Engineering*, 37 (5), pp. 616–634, 2011.

[37] P. Samuel, R. Mall, A. K. Bothra, Automatic test case generation using unified modeling language (UML) state diagrams, *IET Software* 2 (2), pp. 79–93, 2008.