A Constraint-based Test Suite Reduction Method for Conservative Regression Testing

Chang-ai Sun
School of Information Engineering, University of Science and Technology Beijing, Beijing 100083, China
Email: casun@ustb.edu.cn

Abstract— **In regression testing, an important problem is how to select a smaller size of test set for execution. We present a novel constraint-oriented test suite reduction method for conservative regression testing by which we mean that all bugs discovered must be revealed by the reduced test suite. A test constraint for a bug is Boolean formulas defined over the input variables of program under test. The reduced test constraints for a pool of bugs are constructed using the subsumption relationship between test constraint conditions. Test case selection is based on the reduced test constraint set. A test case is selected into the test suite when and only when it satisfies one or more test constraints. The selection process is completed when all test constraint conditions are satisfied by the selected test cases. An empirical study is conducted and the experimental results show that our method can significantly save efforts for the conservative regression testing.**

Index Terms— **software testing, regression testing, test case reduction, test case selection**

I. Introduction

In regression testing, one concern is to verify whether the detected bugs have been removed, and the other is to check whether new bugs are introduced during the modification [10]. This requires testers to re-execute a huge number of test cases developed in the previous stages. Software testing is a kind of an engineering activity, and must be conducted within the limited schedule, budget and human power, thus an important issue in regression testing is how to efficiently select test cases from a test set that have been developed using various test case generation strategies [13]. Lots of test suite reduction techniques have been developed, and they usually select test cases based on some criteria, such as control flow coverage [14], requirement coverage [3], dependency analysis [2] and so on [1], [10]–[13].

Conservative regression testing pays much attention to confirm that the reported bugs are removed. We describe a common scenario of conservative regression testing below. When a failure is detected, testers often record the inputs that cause the failure, the functional domain where the failure takes place, and the steps necessary for repeating this failure. With the reported information, programmers debug the relevant modules. This is an extremely challenging and time-consuming process, since the failure-causing input doses not reveal the true reason of the failure. Furthermore, more than one test case may trigger the same failure, programmers need to figure out all possible inputs that can trigger this failure even they have solved the failure with the reported inputs. In this situation, the problem arises that given a set of bugs reported by testers, how to select the minimum set of test cases that can trigger all of them. We view this as a kind of *conservative regression testing*, which is especially important for the hurry-up software release while the limited budget and schedule is allocated.

In this paper, we present a novel constraint-based test case reduction method for conservative regression testing. This method makes use of test constraint for test suite reduction. For a given bug of program *p*, a test constraint that is defined over input parameters of *p* specifies the necessary conditions that the bug can be detected. In other words, to answer whether a test case can detect a specific bug, we only need to check whether the test case satisfies the test constraint of the bug. Test constraints can be derived through program analysis techniques. For a pool of bugs, we calculate the hierarchy of their test constraints and keep the stronger test constraints. Then the test case selection is based on the reduced constraint set. A test case is selected into the test suite when and only when it satisfies one or more test constraints. The selection process is completed when all constraint conditions are satisfied by the selected test cases. In this way, our method only selects a small subset of test suite for conservative regression testing. Our method does not need to run the program, because that a test constraint for a bug is derived through program analysis techniques; both test constraint reduction and test case selection for a constraint are conducted in the level of class rather than instance.

The main contributions of this work include:

- a method for deriving test constraint for a given bug,
- a test constraint oriented test case reduction strategy, and
- a case study on test suite reduction for a real life program using test constraint-oriented strategy.

The remaining of the paper is organized as follows. Section II presents the concept of test constraints and their construction. Section III discusses the construction of the test constraint hierarchy for a pool of bugs. Section IV

This work was partially supported by the Scientific Research Foundation for the Returned Overseas Chinese Scholars, State Education Ministry, China (Grant No. 2008[890]), the Research Fund for the Doctoral Program of Higher Education of China (Grant No. 20080004010051), and the National Natural Science Foundation of China (Grant No. 60903003).

proposes to reduce test suite based on the test constraint hierarchy. Section V demonstrates the proposed method with a real-life program and reports the experimental results. Section VI discusses related work and compares our method with the exiting approaches. Section VII concludes the paper with pointing out future work.

II. TEST CONSTRAINTS AND THEIR CONSTRUCTION

All inputs of a program constitute the input domain of the program. If there is a bug with a program, it means there must be some inputs that can be used to detect the bug. These inputs are called failure-causing inputs and are part of the whole input domain. Then, how can we restrict the input domain into failure-causing inputs? We call such a restriction as a test constraint. A test constraint answers the question "why and how do the beginning statement influences the faulty statements, and why and how do the faulty statements influences to the statements which can produce different observable outputs".

A. Test constraints

Definition 1 *(program under test).* A program under test p is a three-tuple $\rho =< I, O, S >$ where I is a set of inputs, which can be represented by the parameter vector $< V_1, V_2, \cdots, V_n >$, O is a set of operations, and S is a set of states. A state s_i of p is an instance of input vector $< v_1, v_2, \cdots, v_n >$ where $v_1 \in V_1$, $v_2 \in V_2$ and $v_n \in V_n$. $S_{init} \in S$ and $S_{final} \subseteq S$ are initial state and final states of p, respectively. An operation $o_i \in O$ is the transformation between states $s_i \in S$ and $s_j \in S$, namely $s_i \xrightarrow{o_i} s_j$. Note that a final state refers to the one that produces observable outputs, including returning a value or printing out a message.

Definition 2 *(faulty version)* A mutant f of a program p is said to be a faulty version, when there exist inputs x s.t. $x \in I^p$ and $x \in I^f$ where I^p and I^f are the inputs of p and f, such that $S^p_{final} \neq S^f_{final}$ where S^p_{final} and S^f_{final} are the final states of p and f, respectively.

An example of a faulty version is illustrated in Figure 1 where an operator fault occurs in line 4. For the program p, its input consists of four parameters: the first two are of *integer*, while the last two are of *bool*. The operations of p are a set of *assignment*, *logic* and *relation* operations. The final state of p is represented by the output, i.e. the value of variable alt_sep. Obviously, not all test cases can reveal the bug illustrated in Figure 1. For example, when the test case $< 1, 1, 1, 1 >$ is used as an input, both the original program and the faulty version produce an output of 0. The bug is not revealed because the faulty operation is not activated.

Definition 3 *(trigger-condition).* The input set $bti \subseteq I^f$ which can trigger the bug b is called *bug-trigger inputs*, and the condition which can restrict the whole inputs I^f to bti is called *trigger-condition*.

The *trigger-condition* for the bug shown in Figure 1 is "*!preflag*", and its bug-trigger inputs can be expressed as

```
int alt_test (int own_alt, int other_alt, bool preflag, bool
postflag) {
1    bool upward, downward;
2    int alt_sep = 0;
3    if (!preflag ) {
4       upward = own_alt <= other_alt?1:0;
/*The correct one should be "own_alt<other_alt?1:0".*/
5       downward=other_alt<own_alt ?1:0;
6       if(postflag) {
7          if (upward && downward)
8             alt_sep = 0;
9          else if (upward)
10            alt_sep = 1;
11         else if (downward)
12            alt_sep = 2;
13         else
14            alt_sep = 0;
15         }
16      }
17   return alt_sep;
18   }
```

Figure 1. *An example of a faulty program with one operator mutation.*

$\{< x, y, 0, z > | x \in all\ possible\ values\ of\ own\_alt, y \in all\ possible\ values\ of\ other\_alt, z \in \{0, 1\}\}$. It intends to specify that $preflag$ is restricted to be 0, and there are no constraints defined on $postflag$, own_alt and $other\_alt$.

Definition 4 *(Propagation-Condition).*The input set $fpi \subseteq I^f$ which can guarantee different outputs for f and p after the bug b is triggered is called *fault-propagation inputs*, and the condition which can restrict the whole inputs I^f to *fpi* is called *propagation-condition*.

As to the bug shown in Figure 1, the *propagation-condition* for the bug is "$own\_alt == other\_alt$" and "$postflag$". Its *fault-propagation inputs* are $\{< x, y, z, 1 > | x \in all\ possible\ values\ of\ own\_alt\ \wedge\ y \in all\ possible\ values\ of\ other\_alt\ \wedge x == y \wedge z \in \{0, 1\}\}$.

Definition 5 *(Test Constraint).* Given a faulty version f of a program p, i.e. a bug b is seeded into p, the test constraint of b is a set of constraints that are used to identify all possible inputs that can guarantee the detection of the bug b. The intersection of $btpc(b)$ and $fppc(b)$ is necessary parts of test constraint of the bug b. Here, $btpc(b)$ and $fppc(b)$ are the *trigger-condition* and *propagation-condition* of the bug b, respectively.

The test constraint for the bug in Figure 1 is "*!preflag*", "$own\_alt == other\_alt$" and "*postflag*". The test suite satisfying the test constraint is $\{< x, x, 0, 1 > | x \in\ all\ possible\ values\ of\ own\_alt\ \cap\ all\ possible\ values\ of\ other\_alt\}$. $< 5, 5, 0, 1 >$ is a test case that satisfies test constraint of the bug.

B. Constructing test constraints

We employ program analysis techniques, including slicing [16], chopping [6], and path condition [8], to obtain the trigger-conditions and propagation-conditions of a test constraint.

Definition 6 *(Program Slicing)*. Given a statement *t* in a program *p*, a set of statements *slicing (t)* =$\{s_1, s_2, \cdots, s_n\}$ is extracted to form the slice of the statement *t*, where *t* is called the slicing criteria, s_i is a sentence which potentially has an influence onto the statement *t* (i.e. its execution affects the state s_t^p of program *p* at the statement *t*), denoted as $s_i \xrightarrow{*} t$.

The slices of statements 10 and 14 in the program in Figure 1 are *slicing(10)*=$\{2, 3, 4, 5, 6, 7, 9\}$ and *slicing(14)*=$\{2, 3, 4, 5, 6, 7, 9, 11, 13\}$, respectively. Note that the program slicing defined here is a kind of static sclicing, and thereinafter the line number is used for reference to a statement.

Definition7 *(Program Chopping)*. Given a source criteria *s* and target criterion *t* in a program *p*, the *chopping(s,t)*=$\{s_i | s_i \in s \xrightarrow{*} t \wedge s \in p \wedge t \in p\}$, where $s \xrightarrow{*} t$ is referred to as the path from *s* to *t*.

The choppings of statement 5 to statements 10 and 14 in Figure 1 are *chopping(5,10)*=$\{5, 6, 7, 9\}$ and *chopping(5,14)*=$\{5, 6, 7, 9, 11, 13\}$, respectively. Statements in source criterion or target criterion are often replaced by the nodes of PDGs (Program Dependency Graph [4]) to provide the higher abstraction.

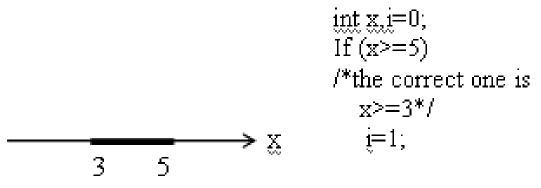
Definition 8 *(Path Conditions)*. Path conditions give necessary conditions under which a transitive dependence between source criterion *s* and target criterion *t* of the program *p* exists. *Path_Condition(s,t)*=$\{ec | s_i \in chopping(s, t) \wedge ec \mapsto s_i\}$, where $ec \mapsto s_i$ denotes that the execution condition of s_i is ec.

Execution conditions for statements 3-14 of the program in Figure 1 are as follows:

$$
\begin{aligned}
&ec(3) := true,\\
&ec(4) := preflag == false,\\
&ec(5) = ec(6) = ec(4),\\
&ec(7) := preflag == false \wedge postflag == true,\\
&ec(8) := preflag == false \wedge postflag == true\\
&\qquad \wedge own\_alt \leq other\_alt \wedge other\_alt < own\_alt,\\
&ec(9) = ec(7),\\
&ec(10) := preflag == false \wedge postflag == true\\
&\qquad \wedge own\_alt \leq other\_alt,\\
&ec(11) := preflag == false \wedge postflag == true\\
&\qquad \wedge own\_alt > other\_alt,\\
&ec(12) := preflag == false \wedge postflag == true\\
&\qquad \wedge own\_alt > other\_alt \wedge other\_alt < own\_alt,\\
&ec(13) := preflag == false \wedge postflag == true\\
&\qquad \wedge own\_alt > other\_alt \wedge other\_alt \geq own\_alt,\\
&ec(14) = ec(13).
\end{aligned}
$$

Based on $chopping(5, 10) = \{5, 6, 7, 9\}$ *and chopping* $(5, 14) = \{5, 6, 7, 9, 11, 13\}$, we can further obtain their path conditions: $path\_condition(5, 10) := ec(5) \wedge ec(6) \wedge ec(7) \wedge ec(9) := preflag == false \wedge postflag == true$ and $path\_condition(5, 14) := ec(5) \wedge ec(6) \wedge ec(7) \wedge ec(9) \wedge ec(11) \wedge ec(13) := preflag == false \wedge postflag == true \wedge own\_alt > other\_alt \wedge other\_alt \geq own\_alt$.

In order to exclusively show the propagation effect of the bug *b*, we need to restrict the inputs to fall in the offset



```
int x,i=0;
If (x>=5)
/*the correct one is
   x>=3*/
    i=1;
```

Figure 2. *An illustration of input offset caused by the bug.*

caused by the fault in terms of input domain, denoted as *Offset(b)*. Figure 2 illustrates such an offset. When $x < 3$ or $x \geq 5$, the variable *i* is assigned to the same value in both the faulty version and the correct version. When $3 \leq x \leq 5$, the variable *i* is assigned to 0 in the faulty version, while 1 in the correct version. Thus, the *offset(b)* is $3 \leq x \leq 5$. As to the bug in 1, the *upward* is 0 in the original program when own_alt is equal to $other\_alt$, while 1 in the faulty version. *Offset(b)* is that $other\_alt == own\_alt$ needs to be evaluated to be true.

Since the test constraint of a bug *b* is the intersection of bug-triggering *trigger-conditions* and fault-propagation *propagation-conditions*, the test constraint *ts(b)* of the bug *b* is equivalently the combination of $path\_conditions(BS, s)$, $offset(b)$ and $path\_conditions(s, ES_i)(i = 1..n)$, where *BS* is the beginning statement of program *p*, ES_i is one of the end statement set *ES*, *s* is the statement where the bug *b* occurs. In order to be efficient, the calculation of path conditions can be executed based on their slicings and choppings. *Slicing(s)* indicates the statements which affect on the execution of *s*, *Chopping (s,t)* indicates the possible fault propagation paths from the source *s*. The test constraint for the bug in Figure 1 can be calculated using $path\_conditions(1, 3)$, $path\_conditions(5, 10)$ and $offset(b)$, and the result is $preflag == false \wedge postflag == true \wedge other\_alt == own\_alt$.

We propose algorithm 1 in Figure 3 to construct test constraints. It makes use of slicing, chopping and path conditions in an integrated way. The algorithm assumes that the program under test is a *C* program with only one single function and with only one single faulty statement, and consists of *assignment*, *branch*, *goto* and *return* statements. The algorithm first constructs a PDG of program *p*, where the entry statement, faulty statement and output statements can be mapped into different nodes; it then constructs *bug trigger chopping* (between entry statement and faulty statement) and *fault propagation choppings* (between faulty statement and output statements); finally, it calculates trigger-conditions and propagation-conditions. Since how to construct a PDG of a program *Construct_PDG (p)*, program slices *Construct_Slice(s, t)*, program chops *Construct_Chopping(s, t)* and path conditions *Construct_PathConditions(s)* are well discussed, we will not extensively discuss these issues. For details, the interested can refer to [6], [8], [13].

The procedure *get_OffsetCondition(sc)* as illustrated in Figure 4 returns the *Offset(sc)* of a mutant *sc*(i.e. a bug).

```
Algorithm 1 Test Constraint Construction for a Single Fault TS_Construction(p,sc, ts)
{
INPUT
p : $\{s_i | type(s_i) \in \{assignment, branch, goto, return\} \wedge 1 \le i \le n$; Note that type $(s_i)$ is type of statement $s_i$;
sc: $s_i \overset{mutation}{\longrightarrow} s'_i$, where $s_i \in p$, $s'_i$ is a mutated statement;
OUTPUT
ts:$\{cs_j | \, |cs_j| \in \{true, false\} \wedge vars(cs_j) \subseteq paras(p) \wedge 1 \le j \le m\}$; Note that $vars(cs_j)$ denotes a set of variables in $cs_j$; $paras(p)$ denotes a set of input parameter variables of program $p$.
PROCEDURE
  1) Initialise ts to $\emptyset$, the beginning statement of $p$ to $s_{init}$ , the output statement set to $s_{final}$, the input parameter variables to $paras(p)$.
  2) Construct a PDG of $p$ using the procedure Construct_PDG (p), where $PDG = \{Nodes, Edges\}$.
       - Note that for each $s_i \in p$, there exists a node $n_k \in Nodes$. For $s_i \in p \wedge s_j \in p \wedge s_i \in n_k \wedge s_j \in n_m \wedge n_k \neq n_m$, if there exists an edge $t \in Edges$ between $n_k$ and $n_m$, then there exists dependency between $s_i$ and $s_j$. When constructing the PDG of a program, the mutated statement $s'_i$ should be included in a separate node.
  3) Map $s_{init}$, $s_i$ and $s_{final}$ to the nodes $N_{init}$, $N_{mutant}$ and $N_{final}$ in PDG, where $N_{init} \in Nodes$, $N_{mutant} \in Nodes$, and $N_{final} \subseteq Nodes$.
  4) Construct program slices $slice$ using the procedure Construct_Slice($N_{init}, N_{mutant}$), and $ts \leftarrow ts \cap Construct\_PathConditions(slice)$.
  5) $ts \leftarrow ts \cap get\_OffsetCondition(sc)$.
  6) For each $n_i \in N_{final}$, construct chops using the procedure Construct _Chopping ($N_{mutant}, n_i$) and $ts \leftarrow ts \cap$ Construct_PathConditions($chops - \{s_i\}$).
  7) Return ts.
END
}
```

Figure 3. *A sketch of Algorithm 1*

Here, $ts(var, vs_i, vs'_i)$:= $|var| \rightarrow vs_i \cap vs'_i$, where vs_i and vs'_i satisfy *assume*($s(vs_i/var)$) $\oplus$ *assume*($s(vs'_i/var)$). Note that $s(y/x))$ is referred to as that all the occurrences of x in statement s are substituted by y which is a set of feasible values of variable x. $assume(s)$ is defined as follows.

$$assume(s) = \begin{cases} var == exp; & \text{if } s \text{ is an assignment statement like } var = exp \\ var\ op\ exp; & \text{if } s \text{ is a } branch \text{ statement like } if(var\ op\ exp) \text{ where } op \in \{>, \ge, \le, <, ==, != \} \end{cases}$$

Algorithm 1 can apply to normal C programs through pre-processing. Like slicing execution [19], we can inline the function body at every function call site to get an equivalent C program with only one function, and use *if* and *goto* statements to rewrite all loops in C program.

```
Procedure get_OffsetConditions(sc)
{
INPUT
sc: $s_i \overset{mutation}{\longrightarrow} s'_i$.
OUTPUT
dts:$\{cs_j | \, |cs_j| \in \{true, false\} \wedge vars(cs_j) \subseteq paras(p) \wedge 1 \le j \le m\}$;
PROCEDURE
  1) $dts \leftarrow \emptyset$.
  2) foreach var in $vars(s_i) \cap paras(p) \cap vars(s'_i)$, where $vars(s_i)$ and $vars(s'_i)$ are variables in statements $s_i$ and $s'_i$, respectively, $dts \leftarrow dts \cap ts(var, vs_i, vs'_i)$.
  3) Return dts.
}
```

Figure 4. *The sketch of Procedure get_OffsetConditions.*

After rewriting, the C program has only one function and is composed of *assignment*, *branch*, *goto* and *return* statements.

We assume the faulty version contains only one bug when the faulty version is compared with the original one, as illustrated in Figure 1. Actually, we can extend the test constraint of a single fault to the one containing multiple faults. A faulty version *f* of the program *p* contains bugs $b_1, b_2, \ldots, b_n$, $tc_1, tc_2, \ldots, tc_n$ is the test constraint of bugs $b_1, b_2, \ldots, b_n$, respectively. Each test constraint tc_i for a single fault can be derived using Algorithm 1. Then test constraint for this faulty version (composite of bugs $b_1, b_2, \ldots, b_n$) is $tc_1 \cup tc_2 \cup, \ldots, \cup tc_n$. Hereinafter test constraints may be referred to as test constraint for single or multiple faults unless otherwise indicated.

III. CONSTRUCTING TEST CONSTRAINT HIERARCHY

For a given bug of a program, we can obtain its test constraints using program slicing, chopping and path conditions as discussed in Section II. Repeatedly, we can obtain a set of test constraints for a pool of reported bugs. Each test constraint specifies the conditions that can guarantee the detection of the targeted bug in a program under test. In other words, test constraints restrict the selection of test cases to the particular area of input domain of the program. Sometimes, test cases *a* and *b* have overlapping in the input domain. This means that test constraints of different bugs may have hierarchical constraint conditions.

A test constraint is a Boolean formula. The operators between two constraint conditions in a test constraint are disjunctive ($\vee$), conjunctive ($\wedge$) and not (!) and parentheses. We can transform a test constraint ts in a general form to one ts' in disjunctive normal form (DNF) using the *distributive law*. Each term in the resulting test constraint ts' is a feasible test case *schema*, which is referred to as that a test case satisfying this schema must be able to detect the fault on which ts is constructed. Each literal in a term is an *atomic constraint* condition. For example, the atomic constraints defined on input parameter *x* of program *p* may be $x \geq a$ where *a* is a constant, or $x \leq y$ where *y* is another variable or input parameter. If more than one literal is defined on the same input parameter *x*, then these literals are the *composite constraint condition* for *x*. For example, $x \geq a \vee x \geq b$ and $x \geq a \wedge x \leq b$ (*a* must be less than *b*; otherwise, it is an unsatisfiable constraint) are two composite constraints for *x*.

Definition 9 *(null constraint condition).* For an input parameter *x* of a program *p*, if there does not exist a test constraint condition defined on *x*, we say *x* has a *null constraint.*

Definition 10 *(stronger constraint condition).* c_1 and c_2 are two constraints defined on the input parameter *x*, c_1 is said to be stronger than c_1 (denoted as $c_1 \succ c_2$), if and only if, any value *v* satisfying c_1 must satisfy c_2.

A *null constrain* is the weakest constraint. A *stronger constraint* restricts the qualified values to smaller scope. For example, $x > 5$ and $x > 7$ are two constraints on *x*, then $x > 7$ is stronger than $x > 5$.

Definition 11 *(test constraint subsumption)* ts_1, ts_2 are two test constraints of program *p*, and $V_1, V_2, \cdots, V_n$ is a set of input parameters of *p*, for all $V_i (i = 1..n)$, $C(V_i)$ and $C'(V_i)$ are constraints on input parameter V_i in test constraints ts_1 and ts_2, respectively, if $C(V_i) = C'(V_i)$ or $C(V_i) \succ C'(V_i)$, then we say ts_1 subsumes ts_2 (denoted as $ts_1 \supset_s ts_2$).

Based on the hierarchy among test constraints, we propose the following strategies for test constraint reduction.

1) Strategy-I: If constraint condition c_1 is stronger than constraint condition c_2 (i.e. $c_1 \succ c_2$), c_1 will restrict input domain to a smaller input domain than c_2, c_1 is selected as the reduced constraint.
2) Strategy-II: If test case schema (a term in a test constraint) tcs_1 subsumes test case schema tcs_2 (i.e. $tcs_1 \supset_s tcs_2$), tcs_1 is selected as the reduced test constraint.

We propose Algorithm 2 in Figure 5 to reduce test constraints based on the above reduction strategies. The algorithm assumes that a set of test constraints of all known bugs has been derived in DNF. The body of the algorithm is composed of two passes: the first pass reduces each test constraint by the concept of *stronger constraint*, and the second pass reduces the set of test constraints by the concepts of *test constraint subsumption*. The algorithm returns a reduced test constraint set. Note that the algorithm cannot guarantee that the output is a smallest size of test constraints, $constraint(v_k, t_j)$ denotes the constraint conditions of term t_j defined on the variable v_k, $t_j(-/c_2)$ denotes all occurrences of c_2 in t_j are replaced by *null.*

```
Algorithm 2. Test Constraint Reduction TS_Reduction ($ts_0, ts_r$)
{
INPUT
$ts_0 : \{ts_i | ts_i$ is a test constraint in DNF $\wedge 1 \leq i \leq n\}$;
OUTPUT
$ts_r : \{ts'_j | ts'_j$ is a constraint in DNF $\wedge 1 \leq j \leq m\}$;
PROCEDURE
  1) foreach test constraint $ts_i \in ts_0$ using Strategy-I.
       - foreach term $t_j$ in $ts_i$ {
         foreach variable $v_k$ in $paras(p)$ {
         foreach constraints $c_1 \in constraint(v_k, t_j) \wedge$
         $c_2 \in constraint(v_k, t_j) \wedge c_1 \neq c_2$ {
         If$(c_1 \succ c_2)$
         $t_j \leftarrow t_j(-/c_2)$.
         } } }
       - foreach terms $t_1 \in ts_i$ and $t_2 \in ts_i \wedge t_1 \neq t_2${
         If$(t_1 \subset_s t_2)$
         $ts_i \leftarrow ts_i(-/t2)$.
         }
  2) $ts_r \leftarrow ts_0$.
  3) foreach term $ts'_i \in ts_r$ and $ts'_j \in ts_r$ {
     If $\exists t_1, t_2 (t_1 \in ts'_i \wedge t_2 \in ts'_j \wedge t_1 \supset_s t_2)$
     $ts_r \leftarrow ts_r - \{ts'_j\}$.
     }
  4) Return $ts_r$.
END
}
```

Figure 5. *A sketch of Algorithm 2.*

IV. TEST SUITE REDUCTION VIA CONSTRAINT HIERARCHY

If there are common test constraints between two bugs, we can merge the test constraints. The test cases that satisfy the reduced test constraints can still guarantee the detection of the two bugs. In practice, a software bug library usually consists of a number of reported bugs and test cases. Ideally, all those test cases should be re-executed to verify whether all reported bugs are removed from the program under test. This is often impractical. Instead, we need to select test cases for execution from the pool of test cases that have been developed before regression testing.

We propose the following procedure to select test cases based on the reduced test constraint set ts_r.

1) *Initialise* an empty test case set *TestSuite*, and an empty satisfied test constraint set *SatTC*.
2) For all test cases tc_i in the test case pool *tcp*, figure out the number sts_i of test constraints ts'_j in ts_r which tc_i can satisfy.
3) *Select* the test case tc_m whose sts_m is the largest among the remaining test cases in *tcp*, $tcp \longleftarrow tcp - \{tc_m\}$, $TestSuite \longleftarrow TestSuite \cup \{tc_m\}$, $SatTC \longleftarrow SatTC \cup GetSatTC(tc_m)$; Note that $GetSatTC(tc_m)$ denotes the set of test constraints ts'_j in *tcp* which test case tc_m can satisfy.
4) *Repeat* Steps 2 and 3 until *tcp* is null or the size of *SatTC* does not increase any longer.
5) *Return TestSuite.*

When the procedure stops, it means that either all test constraints has have been satisfied by the selected test cases, or all test cases in the current test cases can not satisfy some test constraints. If former, it means that all reported bugs are detected by the selected test cases; if latter, it means that new test cases need to be constructed to satisfy the remaining test constraints.

Although the proposed method is intended to reduce the size of test cases and selects test cases from a test case library, it can be also used to guide the generation of high-quality test cases for regression testing. This is totally different from those generating test cases from software code or specifications because our method generates test cases on the basis of test constraints. This is particularly useful to design test case for those hard-to-detect bugs provided that test constraint hierarchy exists.

V. CASE STUDY

In this section, we report a case study which is used to validate feasibility and efficiency of the proposed method.

A. Experiment Settings

Subject program. TCAS is an aircraft collision avoidance system developed by the researchers at Siemens. TCAS consists of 138 executable lines of C code in 9 modules. It has 12 input parameters: 5 of them are of Boolean type and 7 are of Integer. We select TCAS as subject program since it has been widely used for empirical study in several literature [5], [7], [9], [14], [15], [17].

Faulty versions. For TCAS, 41 faulty versions have been created by manually seeding "realistic" faults into the base program that is considered correct, and each fault involves single or multiple line changes when compared with the base program [5], [14].

Test suite. The test case pool has 1608 test cases which has been constructed with two steps: a test suite first generated by employing Category Partition Method, and then additional test cases are appended to the test suite to ensure that several kinds of unit coverage in the base program and faulty versions were exercised by at least 30 tests [5], [14].

B. Test suite reduction via test constraints

For each faulty version, we first locate the place where the fault is seeded, and then develop the test constraint using the proposed method in Section II. Finally, we obtain 41 test constraints.

As an illustration, faulty version 6 has an operator error in line 104 where the *less than* operator (i.e. "$<$") is mistakenly replaced by the *less than or equal to* operator (i.e."$\leq$"). The corresponding test constraint is illustrated in Table I. Atomic constraint condition is represented by a Boolean literal, which usually defines a relationship over one or more input parameters. Composite constraint conditions are represented by a Boolean expression consisting of one or more atomic constraint conditions. The test constraint of faulty version 6 is composed of 4 atomic constraint conditions (i.e. No. 1, 4, 5 and 6 in Table I) and 3 composite constraint conditions (i.e. No. 2, 3 and 7 in Table I).

TABLE I.
TEST CONSTRAINT OF THE FAULTY VERSION 6

No	Atomic/Composite constraint conditions
1	$Other\_Tracked\_Alt = Own\_Tracked\_Alt$
2	$Down\_Separation < 400 \wedge Alt\_Layer\_Value = 0 \vee$ $Down\_Separation < 500 \wedge Alt\_Layer\_Value = 1 \vee$ $Down\_Separation < 640 \wedge Alt\_Layer\_Value = 2 \vee$ $Down\_Separation < 740 \wedge Alt\_Layer\_Value = 3$
3	$Climb\_Inhibit = 1 \wedge$ $Up\_Separation + 100 > Down\_Separation \vee$ $Climb\_Inhibit = 0 \wedge$ $Up\_Separation > Down\_Separation$
4	$High\_Confidence = 1$
5	$Own\_Tracked\_Alt\_Rate \leq 600$
6	$Cur\_Vertical\_Sep > 600$
7	$Other\_Capacity != 1 \vee Other\_Capacity == 1 \wedge$ $Two\_of\_Three\_Reports\_Valid = 1 \wedge Other\_RAC = 0$

For the 41 faulty versions of TCAS, we achieved 17 atomic constraint conditions (such as No. 1, 4, 5 and 6 in Table I and 37 composite constraint conditions (such as No. 2, 3 and 7 in Table I), respectively. We further construct the test constraint hierarchy using the concept and strategy in Section III and then employ the test constraint reduction algorithm to reduce the size of test constraint set and obtain 26 reduced test constraints.

Among these 1608 test cases, 1076 valid test cases are accepted and processed by the program. We use the procedure discussed in Section IV to select 26 test cases from 1076 valid test cases. The selected test cases can satisfy the reduced test constraints. For each faulty version, at least one test case can detect the seeded fault. This means that if one wants to confirm whether all seed faults are removed, he just needs to execute these 26 test cases.

C. Results and threats

Experimental results are very exciting since only about 2 percent of test cases are selected for execution for the purpose of conservative regression testing. Besides the test suite reduction efficiency, we also discover that there exists test constraint hierarchy among these 41 seeded faults. For example, test constraints of the faulty version 6 subsume test constraints of the faulty versions 10, 11, and 31. In other words, it is more difficult to detect the bug in the faulty version 6 than in the faulty versions 10, 11 and 31. This preliminary empirical study has shown the feasibility and efficiency of the proposed method. The number and size of the subject program may be the limitation of the empirical evaluation. In addition, the subject program is a numeric application and there are not very complex cascades of if-then-else branches, this could be another limitation of effectiveness evaluation of the proposed method.

VI. RELATED WORK

Various test suite reduction methods have been developed [13]. Below, we describe several typical work and compare them with our method.

Program-based test suite reduction methods select a subset of test cases from original test set using some criteria on programs. Harrold and Rothermel [14] presented a test suite reduction technique using the control-flow coverage as selection criteria. Wu et al. [18] presented a regression testing technique that selects test cases by utilizing static information from the analysis of the program structure and dynamic information by tracing the function-calling sequences.

Specification-based test suite reduction methods use system requirements to select test cases for regression testing. Chittimalli and Harrold [3] presented a regression test selection approach using system requirements along with their associate test cases and their criticality. Paul et al. [12] presented a scenario-based functional regression testing technique.

Model-based test suite reduction methods first model programs or specifications and then select test cases for regression testing using some criteria over the model. Chen et al. [2] proposed a test suite reduction technique using extended dependency analysis. Ali et al. [1] developed a test case selection technique that is based on an extended concurrent control flow graph generated from UML class diagrams and sequence diagrams.

Architecture-based test suite reduction methods make use of architecture information to guide test case selection for regression testing. Muccini et al. [10] explore how regression testing can be systematically applied at the software architecture level in order to reduce the cost of retesting modified systems. A model differencing technique that is used to implement architecture-level regression testing is reported in [11].

The test suite reduction method presented in this paper belongs to the program-based category. However, our method is completely different from existing program-based test suite reduction approaches in that our method is based on test constraints that are Boolean formulas defined on the input parameters of a program; thus our method itself does not concern the structure or data flow of the program although calculating test constraints needs to analyze the program, our method does not need to run the program in order to select test cases for regression testing, and our method reduces test suite at the level of constraints instead of test case instances.

VII. CONCLUSIONS AND FUTURE WORK

We have presented a test constraint-oriented test suite reduction method for selecting a smaller size of test suite for conservative regression testing. The method consists of construction of test constraints for each bug, reduction of test constraints according to their hierarchy for a pool of bugs, and selection of test cases based on the reduced test constraints. A test constraint can be derived via program slicing, chopping, path conditions and sensitive data analysis techniques. A test case is selected into the test suite only when it satisfies one or more reduced test constraints. In this way, our method doesn't need the execution of program under test, and selects the smaller size of test suite for regression testing. A case study has been conducted and the experimental results show the feasibility and efficiency of the propose approach.

For future work, we will seek the latest results from the area of symbolic executions and dataflow analysis, and leverage them for more experiments. The test constraint method analyzes the source code of programs under test and its success relies heavily on tool support for slicing, chopping and path conditions analysis. Another interesting topic is to verify the effectiveness when the proposed method is applied to unrestricted regression testing.

ACKNOWLEDGMENTS

We thank all the contributors of TCAS, and the anonymous reviewers for their comments which greatly improve the presentation of this paper.

REFERENCES

[1] A. Ali, A. Nadeem, M. Iqbal, M. Usman."Regression Testing Based on UML Design Models", *Proceedings of the 13th Pacific Rim International Symposium on Dependable Computing (PRDC 2007)*, IEEE Computer Society, 2007, pp.85-88.

[2] Y. Chen, R. Probert, H. Ural, "Regression test suite reduction using extended dependence Analysis", *Proceedings of the 4th international workshop on Software quality assurance, in conjunction with the 6th ESEC/FSE*, ACM Press, 2007, pp.62-69.

[3] P. Chittimalli and M.J. Harrold, "Regression test selection on system requirements", *Proceedings of the 1st India software engineering conference*, ACM Press, 2008, pp.87-96.

[4] J. Ferrante, K.J. Ottenstein, and J.D. Warren, "The program dependence graph and its use in optimization". *ACM Transactions on Programming Languages and Systems*, 1987, 9(3):319-349.

[5] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand, "Experiments on the Effectiveness of Dataflow- and Control flow- based Test Adequacy Criteria", *Proceedings of 16th International Conference on Software Engineering (ICSE 1994)*, 1994, pp.191-200.

[6] D. Jackson and E.J. Rollins, "A new model of program dependences for reverse engineering", *Proceedings of 2nd ACM SIGSOFT symposium on Foundations of software engineering (FSE 1994)*, 1994, pp.2-10.

[7] J.A. Jones and M.J. Harrold, "Test-Suite Reduction and Prioritization for Modified Condition/Decision Coverage", *IEEE Transactions on Software Engineering*, 2003, 29(3):195-209.

[8] J. Krinke, "Slicing, chopping and path conditions with barriers", *Software Quality Control*, Kluwer Academic Publishers, 2004, 12(4): 339-360.

[9] M. Marre and A. Bertolino, "Using Spanning Sets for Coverage Testing", *IEEE Transactions on Software Engineering*, 2003, 29(11):974-984.

[10] H. Muccini, M. Dias, D. Richardson, "Towards software architecture-based regression testing", *ACM SIGSOFT Software Engineering Notes*, ACM Press, 2005, pp.1-7.

[11] H. Muccini, "Using Model Differencing for Architecture-level Regression Testing", *Proceedings of the 33rd EUROMICRO Conference on Software Engineering and Advanced Applications(EUROMICRO SEAA 2007)*, IEEE Computer Society, 2007, pp.59-66.

[12] R. Paul, L. Yu W. Tsai, X. Bai, “Scenario-Based Functional Regression Testing”, *Proceedings of the 25th International Computer Software and Applications Conference (COMPSAC 2001)*, IEEE Computer Society,2001,pp.496.
[13] G. Rothermel, and M.J. Harrold, “Analysing Regression Test Selection Techniques”, *IEEE Transactions on Software Engineering*, 1996, 22(8):529-551.
[14] G. Rothermel, and M.J. Harrold, A Safe, Efficient Regression Test Selection Technique, *ACM Transactions on Software Engineering and Methodology*, 1997, 6(2):173-210.
[15] G. Rothermel, R.H. Untch, C. Chu, and M.J. Harrold, “Prioritizing Test Cases For Regression Testing”, *IEEE Transactions on Software Engineering*, 2001, 27(10):929-948.
[16] M. Weiser, “Program slicing”, *IEEE Transactions on Software Engineering*, 1984, 10(4):352-357.
[17] E. Weyuker, T. Goradia, A. Singh, “Automatically generating test data from a Boolean specification”, *IEEE Transactions on Software Engineering*, 1994, 20(3):353-363.
[18] Y. Wu, M. Chen, H. Kao, “Regression Testing on Object-Oriented Programs”, *Proceedings of the 10th International Symposium on Software Reliability Engineering (ISSRE 1999)*, IEEE Computer Society, 1999, pp.270.
[19] X. Yi, J. Wang, and X. Yang, “Verification of C programs using slicing execution”, *Proceedings of the 5th International Conference on Quality Software (QSIC 2005)*, IEEE Computer Society, pp.109-116.

Chang-Ai Sun was born at Jiangsu, China in 1974. He received his Ph.D. degree in Computer Software and Theory from Beihang University in 2002; a bachelor degree in Computer Science from University of Science and Technology Beijing in 1997. His research areas are software testing, software architecture and service-oriented computing.

He is currently an Associate Professor in School of Information Engineering, University of Science and Technology Beijing, China. Before he joined University of Science and Technology Beijing, he was an Assistant Professor in School of Computer and Information Technology, Beijing Jiaotong University. From 2003 to 2007, he was a postdoctoral research fellow at the Faculty of Information and Communication Technologies, Swinburne University of Technology, Australia, and at Faculty of Mathematics and Natural Sciences, the University of Groningen, the Netherland, respectively. In 2003, he worked as a Research Associate at Department of Computing, HongKong Polytechnic University. He has published more than 40 referred journal or conference papers in software engineering and service-oriented computing.

Dr. Sun is a member of IEEE, a senior member of China Computer Federation (CCF), an IBM Academic Initiative Member, a member of YOCSEF committee of CCF, a member of Experts Panel of Beijing Haidian District Scientific Committee, an International Reader of Australian Research Council, a Program Chair of TrustCom 2008, and a Program Committee member of more than ten international conferences.