Program Repair Based on SMT

Zhang Ling, Wan Hai, Peng Yi, Guo BoTing

School of Data Science and Computer, Sun Yat-sen University, Guangzhou, China

Abstract

For high-level programming language like C, it is hard to deduce its complicated data dependency and control flow during the progress of automatic repair. By converting C program to boolean program and simulating its execution, we managed to collect all bad paths that lead the program into error states, and deduce the exact boolean repair statement by excluding all the bad routes. The repaired boolean program can be converted back to C program, thus the whole procedure of repairing can be deemed fully automatic. We further give out the design and architecture of the automatic repairing tool that is implemented based on this method, and put it under testing to verify its effectiveness and correctness.

Keywords: Code Repair, Boolean Program, SMT, Reverse Conversion

1. Introduction

1.1. Background

Debugging is the process of finding and resolving bugs or defects that prevent correct operation of computer software or a system[1]. The process of debugging requires *fault localization* and *program repair*[2]. Fault localization means finding the location of faults, while program repair means fixing the faults, mostly done by developing correct statements and removing faulty ones[3]. For most cases, such procedure has to be done manually, which can be notoriously hard and time consuming.

Program repair takes up the most parts of software maintenance. Finding a bug requires manual tracking and analysis, and only after further development and testing can the update patch be published to fix the bug[4]. Fixing a bug in such maintenance fashion takes up too much time, while the company can only choose to terminate the software or keep it running at the risk of exposing its defect, which could bring great threat to the software's availability and security. Both choices can bring great financial cost to the company[5]. Some reports place software maintenance, traditionally defined as any modification made on a system after its delivery, at 90% of the total cost of a typical software project[5]. Modifying existing code, repairing defects, and otherwise evolving software are major parts of those costs[6]. However, the number of outstanding software defects typically exceeds the resources available to address them. Even a mature software projects would be forced to ship with both known and unknown bugs, because they lack the development resources to deal with every defect. For example, in 2005, one Mozilla developer claimed that, everyday, almost 300 bugs appear, far too much for only the Mozilla programmers to handle.

The great cost of program repair comes from the necessity of manual intervene[7]. To solve such problem, research on automatic software repairing has been in the spotlight. Unfortunately, most traditional research focuses on automatic fault localization, while program repair is still done by human. Many effective methods of program repair have been proposed in the recent years, which do make it easier to locate the cause of software defects, but they are still not accurate enough to come up with a concrete advice for program repair: the job of designing and developing update patch still needs to be done by human developers. What developers need is to recover the software system from failure and make sure such error will not happen again[8]. In this circumstance, premeditated repair is generally applied, meaning pre-defined repairing strategies for predicted errors.

The program repairing methods proposed in recent years are mostly based on formal specifications or test cases, but it is hard to deduce the complicated data dependencies and control flows for high-level programming language like *C*. Thus, we need a different language or specification which is easy to deduce its control flow structure as we try to repair the program automatically. In 2000, a new model and process for program analysis was proposed by Microsoft researchers – the boolean program[9]. Boolean program is similar to C program: they both have functions and recursions, global and local variables[10].

The difference is that all variables in boolean program are of boolean type and no additional storage is available. Boolean programs also support assertions, parallel assignments, and nondeterminism.

To some extent, boolean program can be considered as a push-down automaton which accepts context-free language, its reachability and termination are both decidable[11]. Boolean programs can be thought of as an abstract representation of C programs as it explicitly captures the correlations between data and control flow, in which boolean variables can represent arbitrary predicates over the unbounded state of a C program.[12]. Such characteristics of boolean program make it a suitable tool for data and control flow deduction. Boolean program has already been applied in several program repair research[13], and judged by the experimental results, repairing program based on its control flow structure is effective. However, such method is not fully automatic: after repairing the boolean program, the boolean repair statement still needs to be converted back to C language by human, and the results that satisfy the specification after conversion might not be unique, it is still human's job to pick one among them.

Judging by the characteristics of boolean program, it will make a suitable intermediate language for program analysis and repair. The work we present here goes one step beyond the former research: we analyze the program based on its control flow structure to find out the appropriate boolean repair, and convert the repair statement back to readable and executable C statement automatically, totally without human intervene. The process of repairing a C program can be considered as three steps: convert the C program into boolean program, repair the boolean program and convert the repaired result back to C program. Each step shall be done automatically to achieve fully automatic program repair. We focus on automatizing the third step.

1.2. Research situation

In the field of automatic program repair, some intriguing methods have been proposed in recent years. These methods can be categorized into two types: one based on formal specification the user input and the other based on test cases. The advantage of specification-based methods is that once a sound specification is established, the system can locate the error and build up the accurate repairing according to the specification. Unfortunately, the process of establishing such formal specification is notoriously hard, in which case this type of methods can hardly be generally applied. Without the necessity of specification, all the methods of the later type need is sound and reasonable test cases. In recent years, several practical methods of this type have been proposed.

In 2006, Griesmayer presented a way to repair a C program by converting it to a boolean program, and applied such method to repair a real-world operating system driver[13]. Inspired by the concept of Game Theory, Griesmayer considered the repairing problem as a game[14]. The two players of the game are the environment, which provides the inputs, and the system, which provides the correct values for the unknown expression. The game is won if for any input sequence the system can provide a sequence of values for the unknown

expression such that the specification is satisfied. A winning strategy fixes the proper values for the unknown expression and thus corresponds to a repair. One important aspect of such technique is that it repairs the original program based on its equivalent boolean program, which is also the base of our work. However, this technique failed to achieve a fully automatic repairing procedure, as manual intervene is still needed when it comes to converting the boolean repair back to target language and choosing the most suitable one among all those equivalent conversions.

Also in 2006, Demsky proposed a kind of formal specifications that based on data structure consistency[15]. If a conflict between the program's data structure and the given formal specification was detected, the data structure repairing algorithm can change the program's data structure to make sure it satisfy the specification. Such technique makes it possible for the program to keep running even if there is a fatal data structure error. It is quite suitable for those systems with non-volatile data structure, as if such data structure was compromised, the whole system will be unavailable. By applying such technique, the inconsistency error of the data structure can be repaired while the system is still running, which can greatly reduce the cost penalties brought by system reboot. However, this technique has its limitation, as it can only deal with the error that comes from data structure, without supporting other kinds of logical error.

In 2008, Arcuri proposed a method called ABF[1], which was also the first time the evolutionary computation was applied to program repair[16]. What this method needs is the program's formal specification and a test suite which contains test cases that can reflect the location of the error. This method uses evolutionary computation technique to change the original program, ultimately acquiring the program variant that can pass all test cases by mutations and crossovers. However, such technique is only applicable for programs with limited length. To really scale up to real-world software, it will be compulsory to apply addtional techniques to reduce the search space of the evolution, hence the method as such is still limited.

In 2009, Forrest and Weimer took one step further, for the first time applied evolutionary computation to repairing a real-world software system[17, 18]. To use such technique, one must provide a suite of test cases that describes the correct behavior of the program. The algorithm represent each program variant as an AST[2] paired with a weighted program path. New variants are generated by modifying existing variants using crossover and mutation, with each modification producing a new AST and weighted program path. The fitness of each variant is evaluated by compiling the abstract syntax tree and running it on the test cases. Its final fitness is a weighted sum of the positive and negative test cases it passes. The algorithm stops until a variant that can pass all of the test cases are found. This paper demonstrates the possibility and potential of

[1] Automatic bug fixing
[2] Abstract syntax tree

applying evolutionary computation to automatic program repair. But there is still limitation. A significant impediment for such technique is the potentially infinite search space it must sample to find a correct program. Even if auxiliary techniques are applied to reduce the search space, it will still grow exponentially as the size of the software grows. Secondly, one must provide a huge number of test cases to fully describe all use cases of the software, which is difficult to produce manually as it is hard to know if a test suite is complete. Such test suite is necessary, as providing an insufficient test suite might result in a variant that fail to achieve some important functionality that the suite fails to describe.

Further research on applying boolean program on program repair can hardly be found since 2006. Statistically, research in recent years concentrates on specification-based fault localization and test-case-based GP[3] program repair. We believe there is more for boolean program. There are mature tools like SLAM[19], presented by Microsoft, and SATABS[20], presented by researchers from Oxford University and Carnegie Mellon University, that can be used to convert a C program to its equivalent boolean program for further verification and other process. The tools can help with the conversion to boolean programs, but converting a given boolean program back to C language might result in more than one valid programs. For example, there are variables $p1$ and $p2$ in a boolean program, representing the predicates $a == 0$ and $a > 1$ of the original C program. If there is a statement $p1 == false \&\& p2 == false$ in the boolean program, it could be $a == 1$ or $a < 0$ when it is converted back to C language. So far, there is no such tool for converting a boolean program back to C language and choosing appropriately among valid options.

1.3. Structure of the Paper

We shall present the basic algorithm of converting boolean program back to C language in this paper, and implement a complete program repair tool by using this algorithm. The paper will be presented in seven sections: Section 1 introduces the basic research situation of boolean program repair, analyzes the limitations of some well-known program repair methods and drops a hint of the necessity of a boolean program reverse conversion tool. Section 2 is for preliminaries, introducing the basic syntax of boolean programs and other basic concepts which are used in later sections. Section 3 first will present the basic process of computing a boolean repair statement, and then focus on converting the statement back to C language and verifying its effectiveness. Section 4 introduces the basic architecture of the program repair tool and analyzes the computational complexity of the tool, with the basic implementation of the tool given in pseudo-code. Section 5 applies the TCAS test suite from Siemens Suit to verify the correctness and effectiveness of the tool. Section 6 is for conclusion, discussing further works.

[3]Genetic Programming

2. Preliminaries

2.1. Syntax and Semantics of Boolean Program

A boolean program can be considered as a predicate abstraction of its corresponding C program, as all variables in boolean program are of boolean type, representing predicates in the C program. The variables in boolean program can be global or local. When declaring a variable in boolean program the type of the variable does not need to be explicitly given, as all of them must be of boolean type. There are only two literals in boolean program: $0(\mathit{false})$ and $1(\mathit{true})$, but the value of a variable could also be *uncertain*(represented by $*$). The statements in boolean program have similar structures with their couterparts in C Labels can also be assigned to arbitary statements. Boolean program also supports parallel assignment like Python, which can be used to assign a tuple of values to another tuple of variables simultaneously. For instance, $a, b = b, a$ can simply swap the values of variable a and b. There are three kinds of statements that can control the execution path of a boolean program: if, $while$, $assert$, which have similar functionalities with their counterparts in C. Next, we shall present the formal definition of a boolean program.

First, assume V represents a set of variables in a boolean program, including global variables and local variables. Also, we use evaluation $\xi \subseteq V$ to represent the set of accessible *true* variables in a given moment of the program execution, and we have its power set of ξ being $X = 2^V$. At this point of view, it shall be intuitive to consider the control flow graph of boolean program B as a DFA[4] A_B, which includes:

- A finite set of *statement*s $S = \{s_1, s_2, \ldots, s_n\}$
- A finite set of *state*s Q, where each $q \in Q$ consists of a statement and an evaluation, i.e. $q = (s, \xi)$, meaning the program is going to execute statement q, and the values of all variables at this time are described by evaluation ξ. A state can be represented as a node in the control flow graph.
- A transition function $q = next(s, \xi)$, which represent a transition from state $p = (s, \xi)$ to state q. If there is a state $p = (s, \xi)$, then $next(p)$ is such a state $q = (s\prime, \xi\prime)$ that if the program execute statement s with evaluation ξ, the program will step to statement $s\prime$ with evaluation $\xi\prime$. A transition from state p to state q can be represented as an edge from node p to node q in the control flow graph.
- An initial state $init \in Q$.

Such definition is sufficient for most boolean programs, but it is a little tricky when we try to simulate an invocation of a function. To address this problem, we use a new automaton to represent the execution of the invoked

[4]Deterministic Finite Automaton

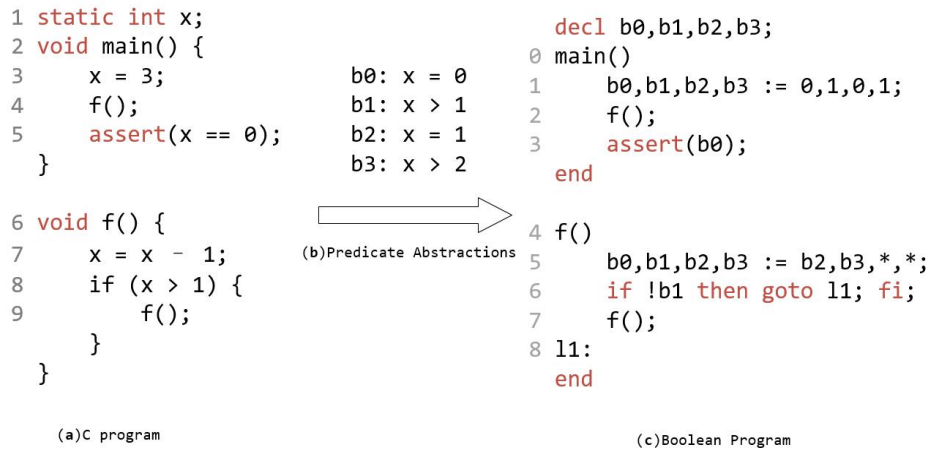

Figure 1: An example of boolean program conversion

function, with its initial state determined by the evaluation of its parameters. The invocation statement and the return of the function trigger transitions from one automaton to another, represented as a dashed edge in the control flow graph. The inter-automata transitions are transparent to the invoker, as the automaton of invoker function will halt while we are simulating the invoked function, and transfer to its next state according to the result returned by the invoked function. Formally speaking, assume the automaton of invoker function being *src* and the automaton of invoked function being *dst*, the edge of invocation would be $\mu : X_{src} \times X_{dst}$. Such mapping assigns the values of arguments to parameters, leaving the values of global variables unchanged, and sets all local variables of the invoked function to *uncertain*. Also, we use function $\rho : X_{src} \times X_{dst} \rightarrow X_{src}$ to represent the change of evaluation produced by the execution of *dst*. Function ρ copies values of all global variables from the evaluation returned by *dst*, copies values of all local variables from the evaluation that *src* use to invoke *dst* and assigns the values returned by *dst* to the variables designated by the invocation statement.

Takes Figure 1 as an example. For the C program in Figure(a), it is obvious that the assertion in line 5 cannot be satisfied. By abstracting the C program using the predicates in Figure(b), the program can be converted to the boolean program in Figure(c). The corresponding control flow graph is shown in Figure 2. Note that the value combinations for $b0$, $b1$, $b2$ and $b3$ can only be 0000, 1000, 0010, 0100 or 0101.

2.2. Constructing Boolean Program

The boolean program used for program repair in this paper is converted from C program by using SATABS[20]. SATABS, being a useful tool for model checking, implements iterative predicate abstraction refinement algorithm[21],

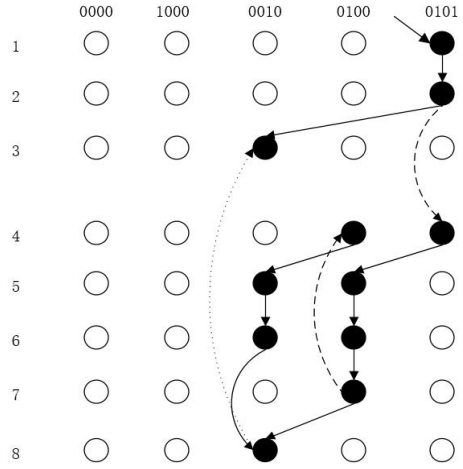

Figure 2: The control flow graph

and can be used to convert ANSI-C program to an equivalent boolean program. For a given C program, SATABS will compute for its abstract representation automatically, which is an effective way to reduce the number of states. Predicate abstraction[22, 23] is one of the most popular abstracting algorithms, which abstracts the program's data by tracking a limited number of predicates. In the abstract model, variables of the original program will be omitted, and boolean variables will be used to represent the abstracted predicates.

2.2.1. Converting an Assignment Statement

Based on the state transitions of the given program, SATABS can calculate the abstracted state transitions between basic blocks and produce a set of abstracted predicates P, with every boolean variable corresponds to one predicate in P. When the program is being converted, if there used to be an assignment statement, for example $x = e$, at location L in the original program, there will also be an assignment statement for the corresponding boolean variable at location L in the boolean program[24]. Value for the boolean variable is computed based on its corresponding predicate. If the value of the predicate cannot be determined at location L, the statement in B will look like $b_i = *$, meaning *uncertain*.

For example, given the C program in Figure 1(a), the predicate set computed by SATABS is $\{b0 : x = 0, b1 : x > 1, b2 : x = 1, b3 : x > 2\}$. Then it is easy to know the boolean assignment statements for C statements $x = 3$ and $x = x - 1$ will be $b0, b1, b2, b3 := 0, 1, 0, 1$ and $b0, b1, b2, b3 := b2, b3, *, *$.

2.2.2. Converting a Control Flow Statement

Except for basic blocks of assignment statements, there are also control flow statements like `if`, `while` and `for` in boolean program. Some of them can have

Table 1: Examples of Boolean Statement Conversions

Type	C	Predicate	Boolean Program

(1)*if*	`if(x>1){`
	`  f();`
	`}`
		`b1 : x>1`	`  if !b1`
			`  then`
			`  goto l1;`
			`  fi;`
			`  f();`
			
			`l1:`

(2)*while*	`while(x>1){`
	`  x=x-1;`
	`}`
		`b0 : x==0`
		`b1 : x>=1`
		`b2 : x==1`
		`b3 : x>=2`
			`l1:`
			`  if !b2`
			`  then`
			`  goto l2;`
			`  if;`
			`  b0,b1,b2,b3 :=`
			`  b2,b3,*,*;`
			`  goto l1;`
			`l2:`

(3)*for*	`a=2;`
	`for(i=0;i<a;i++){`
	`  x=x-1;`
	`}`
		`b0 : x==0`
		`b1 : i>=a`
		`b2 : 1+i>=a`
		`b3 : x==1`
		`b4 : a<=0`
		`b5 : a<=1`
			`  b1,b2,b4,b5 :=`
			`  *,*,0,0;`
			`  b1,b2 := b4,b5`
			`l1:`
			`  if b1`
			`  then`
			`  goto l2;`
			`  if;`
			`  b0,b3 := b3,*;`
			`  b1,b2 := b2,*;`
			`  goto l1;`
			`l2:`

conditional expression as their parameter, but they will not change the values of variables: all they do is changing the direction of control flow. Table 2.2.2 shows examples of conversions of C control flow statements, one can easily sum up the basic rules behind these conversions by these examples.

For conditional statement `if`, we assume there is such an `if` statement at location L, evaluating a boolean expression: if the expression is `true`, the control flow goes to location L_T, otherwise it goes to L_F. During the conversion, we need to abstract the boolean expression. First of all we need to scan the syntactic structure of this expression, examining whether the sub-expressions belong to predicate set P: if they are, we replace the sub-expressions with their corresponding boolean variables b_i. What SATABS does is negating the converted boolean expression c: the exact expression that is used in the converted program would be $\overline{c}$. If $\overline{c}$ is `true`, the program will go to location L_F, being the `l1` in example (1), and ends the `if` statement; otherwise, the program will go to location L_T, being the statements right after the `if` statement in the example.

As for `while` loop, they are converted into `if` statements in boolean program. As usual, we assume there is a `while` statement at location L in the original program, evaluating boolean expression c. The converted statement follows such rule that: if c is `false`, the program will goto location L_T, exiting the `while`

loop; otherwise, the program goes to location L_T, execute the body of the loop, and goes back to location L for the next loop.

`for` loops are also converted into `if` statements in boolean program. Essentially, `for` loop and `while` loop are similar, the difference is one can declare local variables in the initialization of `for` loop, being the variable `i` in example (3). Hence, it is necessary for us to abstract these local variables. The initialization of the original `for` loop will be placed above the converted `if` statement, while the afterthought being converted into assignment statement and placed above the `goto` statement, which stands for the end of the loop body.

2.2.3. Converting a Function Invocation

Converted boolean program will reserve the function invocations from the original C program, by which the boolean program can reflect the correlations of functions in the original C program accurately. For C language, there are invocations with and without parameters, and also functions with and without result returned. For parameter, all functions will be converted into its no-parameter equivalence in boolean program.

For invocation of function with no result returned (`void` function), generally such function will only be used to manipulate global variables, hence it would be equivalent to convert them all to invocation of function without parameter, meaning whether they are `f()` or `f(x)` in the original program, they will all be `f()` in the converted boolean program.

It would be a little tricky when we are dealing with functions with result returned. During the conversion, we need to declare such a boolean variable additionally: it does not stand for any predicates abstracted from the original variables in the C program, but only stands for the returned result of the invoked function.

Figure 3 shows an example boolean program, in which variable `b1` stands for the result returned by function `f`. Before the invocation statement, the converted program will initialize the local variables declared in the invoked function's body, and assign the returning result to the additionally-declared variable at the end of the body.

2.3. Conjunctive and Disconjunctive Normal Form

In mathematical logic, a *well-formed formula*, often simply *formula*, is a word (i.e. a finite sequence of symbols from a given alphabet) that is part of a formal language. The formulas of propositional calculus, also called propositional formulas[25], are expressions such as $(A \wedge (B \vee C))$. Their definition begins with the arbitrary choice of a set V of propositional variables. The alphabet consists of the letters in V along with the symbols for the propositional connectives and parentheses "(" and ")", all of which are assumed to not be in V. The formulas will be certain expressions (that is, strings of symbols) over this alphabet.

The formulas are inductively defined as follows:

- Each propositional variable is, on its own, a formula.

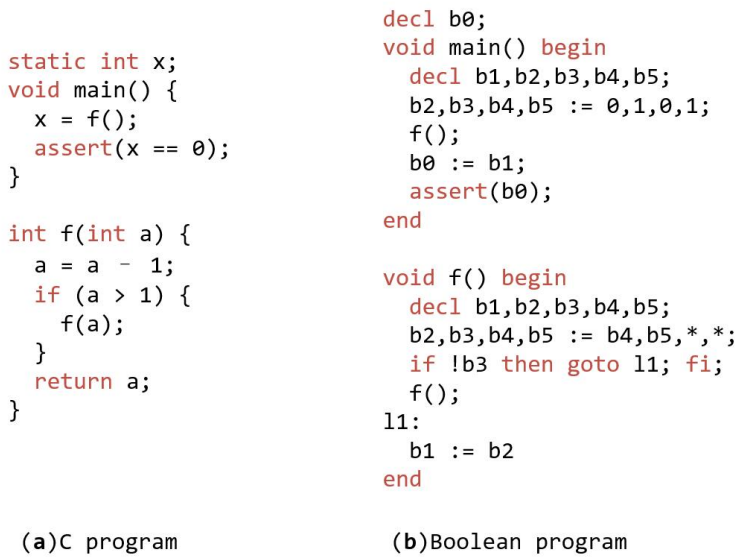

Figure 3: Another example of boolean program conversion

- If φ is a formula, then $\neg\varphi$ is a formula.
- If φ and ψ are formulas, and $\bullet$ is any binary connective, then $\varphi \bullet \psi$ is a formula. Here $\bullet$ could be (but is not limited to) the usual operators $\vee$, $\wedge$, $\rightarrow$, or $\leftrightarrow$.

In Boolean logic, a formula is in *conjunctive normal form* (CNF) if it is a conjunction of clauses, where a clause is a disjunction of literals; otherwise put, it is an *AND* of *OR*s. As comparison, a *disjunctive normal form* (DNF) is a standardization (or normalization) of a logical formula which is a disjunction of conjunctive clauses, which can also be described as an *OR* of *AND*s. For conjunctive and disconjunctive normal form, the only propositional operators are *and*($\wedge$), *or*($\vee$) and *not*($\neg$), where the *not* operator can only be used as part of a literal, meaning it can only precede a propositional variable.

The formal grammar for DNF and CNF can be expressed as follows:

$$
\begin{array}{rcl}
\textit{literal} & \rightarrow & \textit{variable} \\
\textit{literal} & \rightarrow & \neg \textit{variable} \\
\textit{conjunctive clause} & \rightarrow & \textit{literal} \\
\textit{conjunctive clause} & \rightarrow & (\textit{literal} \wedge \textit{conjunctive clause}) \\
\textit{DNF} & \rightarrow & \textit{conjunctive clause} \\
\textit{DNF} & \rightarrow & (\textit{conjunctive clause} \vee \textit{DNF}) \\
\textit{disjunctive clause} & \rightarrow & \textit{literal} \\
\textit{disjunctive clause} & \rightarrow & (\textit{literal} \vee \textit{disjunctive clause}) \\
\textit{CNF} & \rightarrow & \textit{disjunctive clause} \\
\textit{CNF} & \rightarrow & (\textit{disjunctive clause} \wedge \textit{CND})
\end{array}
$$

where *variable* can be any variable.

2.4. Satisfiability Modulo Theories

Variables in converted boolean program are somehow different from general boolean variables: they all are bound to specific predicates abstracted from the original C program, which provide them *meanings*. With all these meanings behind them, some value combinations would be impossible for them to take, as such combinations result in conflicts on their meanings. For example, consider a predicate abstraction where p_1 stands for `x == 0` and p_2 stands for `x == 1`. Combination `p1 : true, p2 : true` is impossible as it is impossible for `x == 0` and `x == 1` both being `true` at the same time. It is important to consider which combinations are possible for a boolean program, as the set of all possible combinations would be generally smaller than 2^V. Considering other problems in a smaller set can bring great reduction to the search space. Hence, we introduce satisfiability modulo theories.

In computer science and mathematical logic, the satisfiability modulo theories (SMT) problem[26] is a decision problem for logical formulas with respect to combinations of background theories expressed in classical first-order logic with equality. Formally speaking, an SMT instance is a formula in first-order logic, where some function and predicate symbols have additional interpretations, and SMT is the problem of determining whether such a formula is satisfiable. In other words, SMT problem can be considered as a SAT[5] problem in which some of the binary variables are replaced by predicates over a suitable set of non-binary variables[27].

We assume a countable set of variables X, function symbols F and predicates P.[28] A first-order signature Σ is a partial map from $F \cup P$ to the natural numbers corresponding to the *arity* of the symbol. A Σ-*term* τ has the form

$$\tau := x | f(\tau_1, \ldots, \tau_n) \tag{1}$$

where $f \in F$ and $\Sigma(f) = n$. For example, if $\Sigma(f) = 2$ and $\Sigma(g) = 1$, then $f(x, g(x))$ is a Σ-term. A Σ-*formula* has the form

$$\psi := p(\tau_1, \ldots, \tau_n) | \tau_0 = \tau_1 | \neg\psi_0 | \psi_0 \vee \psi_1 | \psi_0 \wedge \psi_1 | (\exists x : \psi_0) | (\forall x : \psi_0) \tag{2}$$

where $p \in P$ and $\Sigma(p) = n$, and each τ_i is a Σ-term. For example, if $\Sigma(<) = 2$ for a predicate symbol $<$, then $(\forall x : (\exists y : x < y))$ is a Σ-formula. A Σ-*structure* M consists of a nonempty domain $|M|$, for each $p \in P$ such that $\Sigma(p) = n$, $M(p)$ is a subset of $|M|^n$, and for each $x \in X$, $M(x) \in |M|$. The interpretation of a term a in M is given by $M[\![x]\!] = M(x)$ and $M[\![f(a_1, \ldots, a_n)]\!] = M(f)(M[\![a_1]\!] \ldots M[\![a_n]\!])$. For a Σ-formula ψ and a Σ-structure M, satisfaction $M \models \psi$ can be defined as

[5] Boolean satisfiability problem

$$
\begin{array}{rcl}
M \models a = b & \iff & M[\![a]\!] = M[\![b]\!] \\
M \models p(a_1, \ldots, a_n) & \iff & (M[\![a_1]\!], \ldots, M[\![a_n]\!]) \in M(p) \\
M \models \neg\psi & \iff & M \not\models \psi \\
M \models \psi_0 \vee \psi_1 & \iff & M \models \psi_0 \text{ or } M \models \psi_1 \\
M \models \psi_0 \wedge \psi_1 & \iff & M \models \psi_0 \text{ and } M \models \psi_1 \\
M \models (\forall x : \psi) & \iff & M\{x \mapsto a\} \models \psi, \text{ for all } a \in |M| \\
M \models (\exists x : \psi) & \iff & M\{x \mapsto a\} \models \psi, \text{ for some } a \in |M|
\end{array}
$$

A first-order Σ-formula ψ is *satisfiable* if there is a Σ-structure M such that $M \models \psi$, and it is valid if in all Σ-structures M, $M \models \psi$.

```
1 static int x;
2 void main() {
3     x = 3;
4     f();
5     assert(x == 0);
  }

6 void f() {
7     x = x - 1;
8     if (x > 1) {
9         f();
      }
  }
```

Figure 4: The C program from Figure 1

3. Boolean Program Repair

3.1. Repairing by Bad Routes

To locate fault accurately, establishing an appropriate model of *fault* is essential. In the following sections, we assume there is only one line of faulty code in the original C program.

3.1.1. Constructing Fault Model

In this paper, we focus on one single error in the original C program. To be more specific, one *line* of faulty statement is all we concern. After being converted to boolean program, one variable in C program can correspond to multiple boolean variables, one line of statement can also be converted into multiple lines of statements in boolean program, but only those kinds of errors that maps to one single line of statement in boolean program we try to achieve automatic repair. Thus, the fault model used in this paper can only be applied to one single line of statement after being converted to the corresponding fault model for boolean program. Formally speaking, our fault model can be defined as follows:

Definition 3.1. If there is a fault in the conditional expression of the control flow statement c in the original C program, the corresponding faulty statement in the boolean program is control flow statement c_b; if the fault is in the C assignment statement s, the corresponding fault is in the boolean assignment statement s_b. In a word, one line of faulty statement in C will only correspond to one line of faulty statement in boolean program.

We used Figure 1(a) again, as Figure 4. In this faulty C program, variable `x` is initialized to be `3`, and the program uses statement `assert(x == 0)` to assert its state after the invocation of function `f`. Actually, such operations make up the two basic elements of unit test case, which are "setting the program to a specific initial state" and "asserting the program's final state", or "a known input" and

"an expected output" if you prefer. Here, we give out the basic definition of a faulty program:

Definition 3.2. For boolean program B, if a given initial state makes B fail to pass some `assert` statements, then we say program B doesn't pass the unit test case made up of this initial state and `assert` statement, meaning program B has a fault.

Also, the definition of a correct program is pretty intuitive:

Definition 3.3. A *correct* program can always walk from initial state to expected termination state, without passing through any wrong state that makes an assertion failed.

In section 2.1, we describe the control flow graph of boolean program as a DFA[6]. No doubt that the concept of "path" for DFA should be familiar. In this sense, we define the path of such boolean program DFA as follows:

Definition 3.4. A *path* r of boolean program B is an ordered sequence of states $Q_r = (q_r^{(0)}, q_r^{(1)}, \ldots, q_r^{(t)})$, in which for any pair of elements $q_r^{(i)}$ and $q_r^{(i+1)}$ with $i \geq 0$, we have $next(q_r^{(i)}) = q_r^{(i+1)}$. In the control flow graph, a path is represented by a sequence of end-to-end connected edges.

With definition 3.2 and 3.3, the definition of bad path can be described as follows:

Definition 3.5. For a path r, if there is such state $q_r^i = (s, \xi)$ that statement s is an assertion statement and the current evaluation ξ doesn't satisfy the assertion, we say path r is a *bad* path.

3.1.2. Collecting Bad Paths

For a located faulty statement, we need to produce a correct repairing statement to replace it. Assume $*rep*$ is an arbitrary function of all boolean variables, and consider $*rep$ (or the invocation of it) as the repairing statement. Replace the falty statement with an invocation of the unknown function $*rep$. After the replacement, the concrete implementation of $*rep$ is still unknown, hence we need to consider every possible path branches off at this point.

According to the fault model we estabilished, there will be only one invocation of $*rep$ in boolean program B. To produce the appropriate implementation of $*rep$, we need to re-construct the state transition graph of this program and every path branches off at this point that might lead to wrong termination state. When we confronts $*rep$, we can move from one state to another based on one of its possible implementations. In this case, different implementations lead to different states.

An implementation of $*rep$ can be considered as a mapping from the current evaluation to a boolean value (the result of $*rep$). Assume we have a set of evaluations $c \subseteq X$, which satisfies:

[6]Deterministic finite automaton

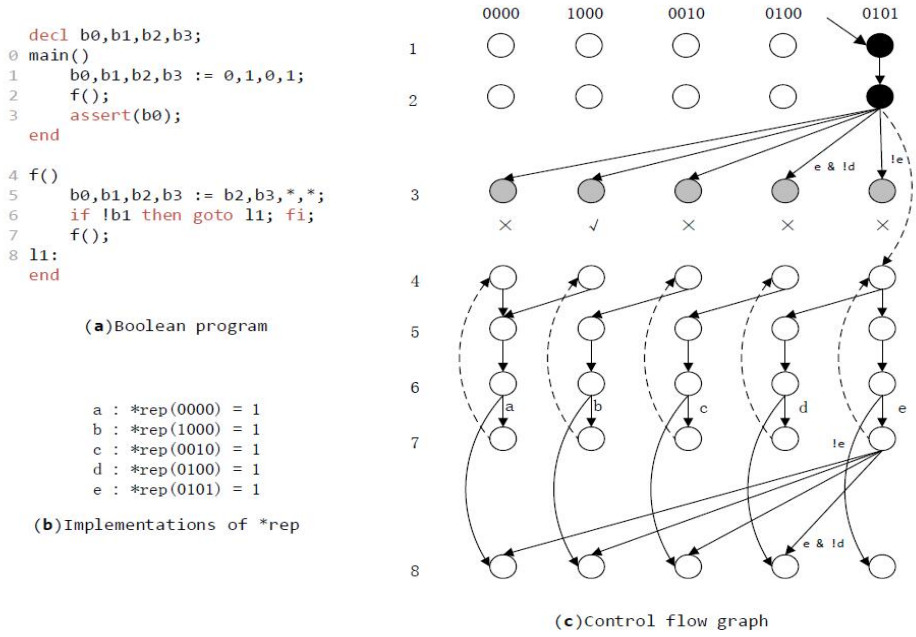

Figure 5: An example of finding all bad routes

Definition 3.6. For any $\xi \in X$, $*rep(\xi) = 1$ if and only if $\xi \in c$. $C = 2^c$ is the set of all implementations of $*rep$.

In other word, c is the set of evaluations which make $*rep$ return 1. For each element of c, it would be easy to imagine its corresponding $*rep$ implementation (the mapping).

Takes Figure 5 as an example, where Figure(a) is basically the same as Figure 1(b). After replacing the faulty statement with $*rep$, by considering every possible transition corresponding to different $*rep$ implementation, one can easily draw the control flow graph in 5(c). In this example, we give out predicates a, b, c, d and e, representing different elements in evaluation set c. Consider state $(6, 0101)$. In this state, the program will examine the result of function $*rep$: if the function return 1, meaning predicate e is satisfied, the program will result in state $(7, 0101)$ and invoke function `f` again; otherwise, if predicate e cannot be satisfied, the program will result it state $(8, 0101)$ and return from function `f`. Assume we enter state $(7, 0101)$ and then state $(6, 0100)$, the program will examine the result of function $*rep$ again: if the function return 0, meaning predicate d is not satisfied, the program will enter state $(8, 0100)$ and return from function `f`. By repeating this, one can calculate the different requirements for the program to enter the 5 different states in line 3. Among them, state $(3, 0000)$, $(3, 0010)$, $(3, 0100)$ and $(3, 0101)$ cannot pass the assertion, meaning they are wrong termination states, and the paths they belong to are

bad paths.

3.1.3. Producing Repairing Statement

By constructing all possible paths that lead the program into wrong termination states, we actually find out all wrong implementations of $*rep$. Then it would be intuitive that the correct implementation of $*rep$ is the complementary of all these wrong implementations.

Definition 3.7. Assume for every implementation c of $*rep$ there is a concrete execution path r_c, the correct implementation of $*rep$ is $I = C - \{c | \exists q \in r_c \wedge q \in bad\}$, where bad is the set of all wrong states.

In the above example, we collected four wrong implementations of $*rep$ corresponding to states $(3, 0000)$, $(3, 0010)$, $(3, 0100)$ and $(3, 0101)$: $bcde$, $\bar{c}de$, $\bar{d}e$ and $\bar{e}$, meaning if the implementation of $*rep$ satisfies $bcde \vee \bar{c}de \vee \bar{d}e \vee \bar{e} = b \vee \bar{c} \vee \bar{d} \vee \bar{e}$, the program will enter wrong state. Based on the above definition, one can safely produce the correct implementation of $*rep$, being $I = \bar{b}cde$, which is also the only path that leads to the correct termination state $(3, 1000)$. After representing I with the corresponding predicates and simplifying it based on first-order logic simplification rules, we have $I = \neg b_0$.

3.2. The Satisfiability of the Repairing Statement

After producing the repairing statement, we also need to consider if this statement is satisfiable.

Boolean program is abstracted from the original C program. After producing the repairing statement, we also need to convert it back to the corresponding C statement. The repairing statement for the boolean program, or *boolean repair*, is a boolean expression consists of boolean variables and operations, in which all boolean variables are produced by predicate abstraction of the original C program, meaning every boolean variable in the boolean repair has its corresponding *background theory*. In this case, it is uncertain whether the expression itself is satisfiable. For example, in Figure 1, the background theory of predicate b_0 is $x = 0$, while b_2 being $x = 1$. Expression $b_0 \wedge b_2$ is clearly unsatisfiable after considering their background theories.

To address this, we need to refine the definition of boolean repair based on Definition 3.7:

Definition 3.8. Assume the set of all bad paths is FP, and $\neg(fp_1 \vee fp_2 \vee \ldots fp_n$ is satisfiable, then we say there is a boolean repair that can repair the program. The boolean repair is $I = C - \{c | \exists q \in r_c \wedge q \in bad\}$, where bad is the set of all wrong states.

Therefore, existence of a repairing statement for boolean program does not necessarily mean the existence of a corresponding repairing statement for the

original C program. We need to determine if the boolean repair, being a first-order logic expression, is satisfiable after considering the background theories. Such problem can be reduced to the problem of SMT[7][26].

As the boolean repair is a first-order logic expression, it would be convenient to first convert it to the equivalent disjunctive normal form, meaning if we have a boolean repair φ_{rep}, we have:

$$\varphi_{rep} = \varphi_1 \vee \varphi_2 \vee \cdots \vee \varphi_n \tag{3}$$

Every φ_i is made up of predicate variables and has the following form:

$$\varphi_i := p(\tau_i, \ldots, \tau_n) | \varphi_a \wedge \varphi_b \tag{4}$$

in which each term τ_i is a Σ-term (see Equation 1), representing predicate abstracted from the original C program. Hence, one can easily notice that the formula of φ_{rep} satisfies the general form of Σ-formula (see Equation 2).

Additionally, we assume the background theories of φ_{rep} is T_{rep}. The background theories of boolean program is abstracted from the original C program, hence T_{rep} corresponds to the predicate variable $p(\tau_1, \tau_2)$ in φ_{rep}, having the same assignment model as predicate variable does.

Above all, based on the definition of SMT shown in section 2.4, one can prove the satisfiability problem of boolean repair φ_{rep} can be reduced to SMT. The boolean repair produced by the method describe in section 3.1.2 is actually the correct repairing statement for the boolean program, but the φ_{rep} also has to be satisfiable after conversion. For now, we know the satisfiability problem of boolean repair φ_{rep} can be reduced to SMT, hence we can use the algorithm for SMT to solve the satisfiability program of φ_{rep}

3.3. Boolean Repair Simplification

For a given DNF[8] expression φ, assume there is such model M that $M \models \varphi$, meaning expression φ is satisfiable. Actually, as φ is a DNF expression, all can be inferred from its satisfiability is that it contains satisfiable conjunctive clause, it doesn't mean every clause of it is satisfiable.

Formally speaking, for a repairing statement $\varphi_{rep} := \varphi_1 \vee \varphi_2 \vee \ldots \varphi_n$, it's satisfiability is not sufficient for the satisfiability of every φ_i.

For example, consider an repairing statement $\varphi_{rep} := (p_1 \wedge p_2) \vee p_3$, where $p_1 : x > 2, p_2 : x < 1, p_3 : x = 0$. The expression is satisfiable as clause p_3 is satisfiable, but clause $p_1 \wedge p_2$ is not satisfiable. If we convert the expression given in the example to C statement, we have `x > 2 && x < 1 || x != 0`, where `x > 2 && x < 1` is meaningless, and the statement can be simplified to `x != 0`. Hence, before we convert the boolean repair to C statement, there will be ways to simplify the statement, removing unsatisfiable clauses is one of them.

In the following sections, we will use three different ways to simplify the repairing statement.

[7] Satisfiability modulo theories
[8] Disjunctive normal form

3.3.1. Removing Unsatisfiable Clauses

For a given satisfiable repairing statement $\varphi_{rep} := \varphi_1 \vee \varphi_2 \vee \cdots \vee \varphi_n$, one can proceed the outer-clause simplification by examining the satisfiability of every clause φ_i. If the clause is satisfiable, it should be preserved; otherwise, it can be removed from φ_{rep}. Based on section 3.2, the satisfiability problem of clause φ_i can also be reduced to SMT.

Assume the new repairing statement produced by removing all unsatisfiable clauses are φ'_{rep}. One can prove φ'_{rep} is equivalent to φ_{rep}, meaning $\varphi'_{rep} \iff \varphi_{rep}$.

Proof. Assume φ'_{rep} is produced by removing unsatisfiable clause φ_i from φ_{rep}, meaning

$$\varphi_{rep} = \varphi'_{rep} \vee \varphi_i \tag{5}$$

It should be intuitive that

$$\varphi'_{rep} \Longrightarrow \varphi'_{rep} \vee \varphi_i \tag{6}$$

Thus,

$$\varphi'_{rep} \Longrightarrow \varphi_{rep} \tag{7}$$

We know clause φ_i is unsatisfiable, meaning φ_i is always $false$ and $\neg\varphi_i$ is always $true$. In this sense, we have

$$\neg\varphi_i \wedge (\varphi'_{rep} \vee \varphi_i) \Longrightarrow \varphi'_{rep} \tag{8}$$

Based on equation (5), we have

$$\neg\varphi_i \vee \varphi_{rep} \Longrightarrow \varphi'_{rep} \tag{9}$$

Given the fact of $\neg\varphi_i$ always being $true$, we have

$$\neg\varphi_i \wedge \varphi_{rep} = \varphi_{rep} \tag{10}$$

Combined with equation (9), we have

$$\varphi_{rep} \Longrightarrow \varphi'_{rep} \tag{11}$$

□

3.3.2. DNF Simplification

In this section, we will elucidate how one can proceed inter-clause simplification based on the characteristics of DNF[9].

A DNF expression can be considered as a tree of clauses, while in the field of computer algorithm, tree-like structure is simplified mostly by rule-based methods. Several general and effective simplification rules will be listed and proved below accordingly.

[9]Disjunctive normal form

Theorem 3.1. For a given satisfiable expression $\varphi = p \vee (q \wedge r)$, if $p \vee r$ or $p \vee q$ always being $true$, φ can be simplified to be $p \vee q$ or $p \vee r$ respectively.

One can prove this theorem easily based on the distribution law of first-order logic.

Theorem 3.2. For a given satisfiable expression $\varphi = (\varphi_1 \wedge q) \vee (\varphi_2 \wedge \neg q) \vee \varphi_{other}$, if every predicate in φ_1 is also contained in φ_2, φ can be simplified to be $\varphi = (\varphi_1 \wedge q) \vee \varphi_2 \vee \varphi_{other}$.

Proof. Each clause φ_i is actually made up of conjunctions of predicates. As every predicate in φ_1 is contained in φ_2, we have $\varphi_2 = \varphi_1 \wedge \varphi_3$, where φ_3 consists all predicates that are in φ_2 but not in φ_1. Then we have

$$(\varphi_1 \wedge q) \vee (\varphi_2 \wedge \neg q) = (\varphi_1 \wedge q) \vee (\varphi_1 \wedge \varphi_3 \wedge \neg q) \tag{12}$$

It can be further transformed into the following equation by using distribution law:

$$(\varphi_1 \wedge q) \vee (\varphi_1 \wedge \varphi_3 \wedge \neg q) = ((\varphi_1 \wedge q) \vee (\varphi_1 \wedge \varphi_3)) \wedge ((\varphi_1 \wedge q) \vee \neg q) \tag{13}$$

in which the $(\varphi_1 \wedge q) \vee \neg q$ part on the right hand side can be simplified by further use of distribution law:

$$(\varphi_1 \wedge q) \vee \neg q = (\varphi_1 \vee q) \wedge (q \vee \neg q) = \varphi_1 \wedge \neg q \tag{14}$$

Therefore,

$$\begin{array}{rcl} (\varphi_1 \wedge q) \vee (\varphi_1 \wedge \varphi_3 \wedge \neg q) & = & ((\varphi_1 \wedge q) \vee (\varphi_1 \wedge \varphi_3)) \wedge (\varphi_1 \vee \neg q) \\ & = & ((\varphi_1 \wedge q) \wedge (\varphi_1 \vee \neg q)) \vee ((\varphi_1 \wedge \varphi_3) \wedge (\varphi_1 \vee \neg q)) \end{array} \tag{15}$$

in which the $(\varphi_1 \wedge q) \wedge (\varphi_1 \vee \neg q)$ part on the right hand side can be simplified by using association law:

$$(\varphi_1 \wedge \varphi_3) \wedge (\varphi_1 \vee \neg q) = \varphi_1 \wedge \varphi_3 = \varphi_2 \tag{16}$$

Hence,

$$\begin{array}{rcl} (\varphi_1 \wedge q) \vee (\varphi_1 \wedge \varphi_3 \wedge \neg q) & = & (\varphi_1 \wedge q) \vee \varphi_2 \\ (\varphi_1 \wedge q) \vee (\varphi_2 \wedge \neg q) & = & \end{array} \tag{17}$$

By such mean, we proved the theorem after adding φ_{other} on both sides:

$$(\varphi_1 \wedge q) \vee (\varphi_2 \wedge \neg q) \vee \varphi_{other} = (\varphi_1 \wedge q) \vee \varphi_2 \vee \varphi_{other} \tag{18}$$

□

The simplification rules listed above are just two examples. No doubt, adding more rules can enhance the accuracy of this algorithm, but judging from experimental results, these two rules are already effective enough for most cases.

3.3.3. Simplification based on Background Theories

Let us start this section with a simple example: assume we have a simplified repairing expression $\varphi_{rep} = p_1 \vee p_2$ after applying the methods described in former sections, where p_1 is `x < 2` and p_2 is `x < 3`. The corresponding C statement of φ_{rep} would be `x < 2 || x < 3`, which can still be simplified to be `x < 3`. In this sense, inner-clause simplification based on background theories is necessary to produce a simplified and intuitive C statement.

Theorem 3.3. For a given simplified repairing expression $\varphi_{rep} = p \wedge q \wedge \varphi$ and the corresponding background theory T, if there is a binary predicate $r = p \wedge q$ in T, repairing expression φ_{rep} can be simplified to be $\varphi_{rep} = r \wedge \varphi$.

Theorem 3.4. For a given simplified repairing expression $\varphi_{rep} = p \vee q \vee \varphi$ and the corresponding background theory T, if there is a binary predicate $r = p \vee q$ in T, repairing expression φ_{rep} can be simplified to be $\varphi_{rep} = r \vee \varphi$.

Theorem 3.3 and 3.4 is related to the background theory T, where the predicate r from T may be new from p and q or one of them. As the repairing expression will always be in DNF, these two simplification rules are actually sufficient.

3.4. Summary

In this section, we illustrated the basic procedure of C program automatic repair. In section 3, we introduced the basic concept of producing boolean repair by collecting bad routes for converted boolean program, while section 3.2 focused on reducing the satisfiability problem of the boolean repair to SMT. We demonstrated some basic rules of simplifying the boolean repair in section 3.3, and by such way we eventually produced an effective and intuitive C repairing statement for the original C program.

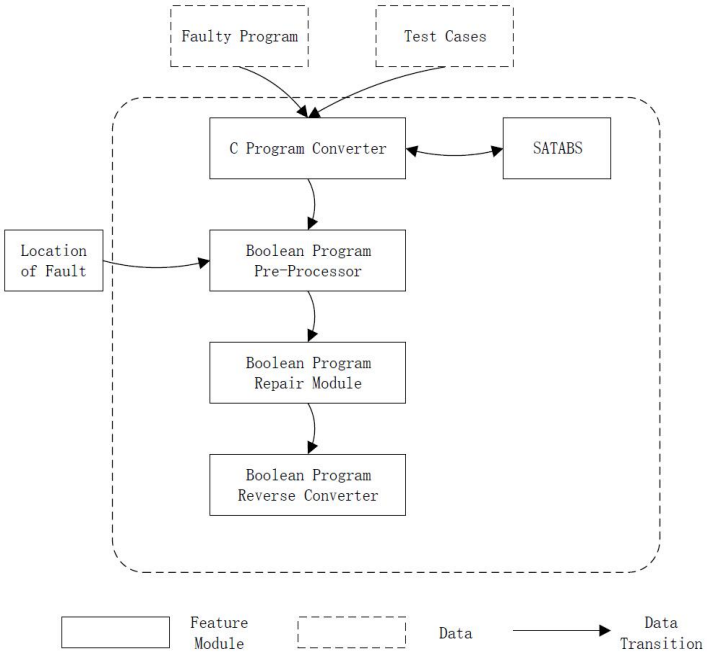

Figure 6: The Structure of the Tool

4. The Basic Design of the Program Repair Tool

The program repair system can be considered as four modules: C program converter, boolean program pre-processor, boolean program repair module and boolean program reverse converter. The correlation of these four parts is shown in Figure 6.

4.1. Modules of the Program Repair Tool

We shall introduce you the details of these four components in this section.

- **C program converter** is responsible for predicate abstraction and converting the given C program to boolean program. We use SATABS[20] to implement this module, which is capable of converting the assignment statements, control flow statements and invocation statements in C program to the corresponding boolean program statements. By doing so, this module achieves the basic conversion from C program to boolean program.

  - **Input**: a faulty C program with given test cases and expected output.
  - **Output**: the corresponding boolean program.

- **Boolean program pre-processor** refines the converted boolean program to some extent. This module is capable of doing some further process on the converted boolean program, including removing useless information, extracting the relations between boolean variables and predicates and the mappings from boolean statements to the original C statements. In this process, the location of the faulty C statement is also passed to the system, leaving for further process in the following repair module.

  - **Input**: a converted boolean program given by SATABS and the line number of the faulty C statement in the original C source file.
  - **Output**: the refined boolean program and other useful related information.

- **Boolean program repair module** constructs and collects all bad paths for the given boolean program and produces the appropriate boolean repair. This module is actually made up of two parts: one for constructing bad paths and the other for producing boolean repair. As the location of the faulty statement is given, bad paths can be constructed by replacing the faulty statement with an unknown repairing function (`*rep`) and enumerating all possible paths that lead to wrong states (see section 3.1.2). Further, the correct boolean repair will be produced based on the set of all bad paths by taking the negation of their respective implementations (see section 3.1.3).

  - **Input**: the refined boolean program and related information.
  - **Output**: boolean repair

- **Boolean program reverse converter** converts the repaired boolean program back to C language and finishes the repairing. First it will replace the boolean variables with their respective predicates and determine the satisfiability of the converted repairing statement based on the SMT reduction described in section 3.2. After this, several simplification rules will be applied on the converted repair statement (section 3.3) to give out an effective and intuitive C repair statement.

  - **Input**: the boolean repair and the mappings produced by boolean program pre-processor.
  - **Output**: the repaired C program.

4.2. Boolean Program Repair Module

First, boolean program repair module will be used to product the appropriate boolean repair. This module can be considered as two parts: one for constructing bad paths and the other for calculating boolean repair. Also, we improve the accuracy of this module by introducing multiple test cases.

For a given faulty program, we need to constructs all possible paths that lead to wrong states. We use test cases as the standard of a correct program

in this paper, and define the specification a program should satisfy by `assert` statements, meaning the expected output of test cases. During the process of constructing bad paths, the location of the faulty statement will be passed as a parameter, and the faulty statement will be replaced by an unknown repairing function. At this point, all possible bad paths will be constructed and recorded.

After collecting all bad paths constructed by the given test cases, we negate the set of bad paths to produce the correct path, and further produce the concrete implementation corresponding to this path.

4.2.1. Multiple Test Cases Repair

More test cases means more bad paths to be constructed. In this sense, one can improve the coverage of bad paths by introducing more test cases, and by which improve the accuracy of the repairing.

The program repair method described in section 3.1.2 can find out all bad paths that lead to wrong states by given input and expected output. For one single test case, the initial state of the program is determined, hence the bad paths can be constructed is determined. Different test cases can have different initial states, and different paths can be constructed. In this sense, introducing only one test case to repair is not completed or sufficient.

For one faulty program, different test cases can be introduced to generate C source files of different versions. We can still construct bad paths for each of them, and produce a more accurate repairing statement by collecting all these bad paths. C source files of different versions can have different predicates and boolean variables, but we can sum up the boolean variables in different bad paths, unifying the names of boolean variables which stand for the same predicate.

Theorem 4.1. Assume we have a set of bad paths FP_a constructed by a test case a. If a new set of bad paths FP_b can be constructed by additionally introducing a new test case b, we have $FP_a \subseteq FP_b$.

As defined in definition 3.7, the correct implementation of `*rep` is calculated by negating the set of bad paths. If more test cases are introduced, based on theorem 4.1, the coverage of bad paths constructed will be improved, meaning more wrong implementations of `*rep` can be found. By doing so, we can improve the accuracy of the resulting boolean repair.

4.2.2. Pseudo Code

The key of constructing bad paths is to simulate the execution of the program until a termination state is reached, and determine if this path is a bad path, judging by `assert` statement and the current evaluation. The set of all possible paths starting from a given state can be considered as a tree, while the given state being the root of the tree. The algorithm collects all possible bad paths by iterating the whole tree in a depth-first manner, guaranteeing every node, or state, will be examined, hence every possible path.

The pseudo code of this algorithm is shown as follows:

Algorithm 1 Function simulating algorithm

```
1: func ← the function being simulated
2: loc ← the location of the statement being simulated
3: eva ← the current evaluation
4: route ← the current route
5: states ← list of all possible next states
6: bad_routes ← the list of all recorded bad routes
7:
8: if loc == 0 then
9:    if loc has already been simulated for func then
10:      return existed result
11:   end if
12: end if
13:
14: push the current state to route
15:
16: st ← func[loc]
17: if st is an assignment statement then
18:   if st is not the unknown repairing statement then
19:      push next state to states
20:   else
21:      push every possible next state to states
22:   end if
23: else if st is an 'if' statement then
24:   if st is not the unknown repairing statement then
25:      push next state to states
26:   else
27:      push every possible next state to states
28:   end if
29: else if st is a 'goto' statement then
30:   push the destination state to states
31: else if st is an invocation statement then
32:   push every possible next state to states
33: else if st is an 'assert' statement then
34:   if st is the unknown repairing statement then
35:      if !check_assert(st, eva) then
36:         add route to bad_route
37:         return the result
38:      end if
39:   end if
40: else
41:   if loc is at the end of func then
42:      return the result of func
43:   end if
44: end if
45:
46: for all state s in states do
47:   simulate s in func
48: end for
49:
50: if loc == 0 then
51:   cache the result of func
52: end if
53:
54: pop the current state from route
```

So far, we collect all bad paths of a boolean program by using the algorithm above. The next thing we need to do is to produce an appropriate boolean repair from these bad paths. In section 3, we have proved that a correct boolean repair should not lead to any wrong state. By negating every bad path and conjuncting all these resulting clauses, the formula we have will represent a correct path.

Algorithm 2 Calculates boolean repair

```
1: good_route ← a empty list
2:
3: for all route in bad_route do
4:    negates any element of route then put it into good_route
5: end for
6:
7: return good_route
```

For now, we have `good\_route`, which stands for the correct implementation of $*rep$, represented as pairs of input and expected output of $*rep$. The last thing to do is to convert it to a readable boolean expression. For example, assume we have something like $*rep(00) = 1$, $*rep(01) = 0$. Apparently, the boolean expression can be $\cap p_0 p_1 || \cap \cap p_0 p_1$, but such representation is complicated when the number of variables goes up, which is also difficult for further simplification. Hence, a better representation is needed. In this case, the concept of Binary decision diagram[29] can be introduced.

The binary decision diagram (BDD) is a data structure for representing boolean functions, which can also be considered as a compressed representation of sets or relations[30, 31]. A boolean function can be represented as a rooted, directed, acyclic graph, which consists of several decision nodes and terminal nodes. There are two types of terminal nodes called 0-terminal and 1-terminal. Each decision node N is labeled by boolean variable V_N and two child nodes, representing setting the variable to $true$ or $false$. A path from the root node to a 1-terminal node means the values of variables represented by this path makes the expression to be $true$. BDD can be used to eliminate duplicate information from a given boolean expression, as equivalent expressions will always produce the same BDD. Also, the representation of BDD is generally smaller than truth table, which can also be used to reduce the search space.

Except for BDD, logic simplification rules described in section 3.3 can also be applied after converting the given expression into DNF.

4.3. Boolean Repair Reverse Converter

By applying the algorithm described in the former section, we have the boolean repair for boolean program. Boolean repair is a boolean expression consisted of boolean variables. To repair the original C program, we still need to convert it to a C statement and determine if the converted repair statement is correct and effective.

Algorithm 3 Conversion of repairing clauses

```
1: clause_str ← the conjunctive clause to be converted
2: vars_map ← mappings from boolean variables to predicates
3: predicates ← the list of converted predicates
4:
5: vars_str ← split(clause_str, "&")
6: for all var_str in vars_str do
7:    isNeg ← if the current variable is a negation
8:    if var_str[0] == '!' then
9:       isNeg ← true
10:      var_str ← var_str.substr(1)
11:   else
12:      isNeg ← false
13:   end if
14:
15:   predicate ← vars_map.map(var_str)
16:   if isNeg then
17:      predicate ← predicate.negate()
18:   end if
19:   predicates.push(predicate)
20: end for
21:
22: return predicates
```

Algorithm 4 Conversion of repairing expression

```
1: expr_str ← the disjunctive expression to be converted
2: clauses ← the list of converted clauses
3:
4: clauses_str ← split(expr_str, " | ")
5: for all clause_str in clauses_str do
6:    clause ← clause_conversion(clause_str)
7:    if clause is always true then
8:       return always true
9:    end if
10:   clauses.push(clause)
11: end for
```

During the conversion, the first thing to do is to convert the boolean repair to its respective predicate formula.

As the repair statement generated by the tool will always be in DNF, simplification can be applied to produce a more readable statement. Several effective simplification rules are described in section 3.3, while applying BDD for simplification is also acceptable.

4.4. Complexities of Algorithms

In the former sections, we described the basic implementation of the program repair tool. In this section, the time complexity of different modules will be analyzed.

The C program converter is implemented by using SATABS to convert a C program to boolean program, hence the complexity of this module will not be considered.

The algorithm used to implement boolean program pre-processor is constant time. This module is responsible for extracting the information of boolean variables and the mappings from C statements to boolean statements. Such intermediate information will be generated when SATABS is converting the C program, constant number of searches on both source files is all we need. If we have to consider the length of source files, let us assume the number of lines in the original C program is LoC and the converted boolean program is $LoBP$, then the complexity of this algorithm should be $O(LoC + LoBP)$.

The boolean program repair module is actually the bottleneck of the whole system. Assume the number of boolean variables is V, then the number of evaluations could be $X = O(2^V)$, which means the number of possible paths is $O(2^X)$, and the number of distinct states is $O(LoBP * X)$. In this sense, the number of paths grows exponentially as the number of boolean variables grows, which also grows exponentially as the number of variables in the original C program grows. It would appear to be the major bottleneck of the whole system, as the complexity of this module is $O(LoBP * X + 2^X)$. In this case, further optimization can be applied on this module.

In our implementation, we use different ways to reduce the number of variables needed to be considered during the simulation, meaning reducing V. As $X = 2^V$, no doubt the reduction on V can greatly facilitate the performance of the algorithm. Such optimization is actually effective, as for each statement, most accessible variables are only of temporary use or irrelevant to the simulating statement. Hence, we only consider the state transitions of those variables which is not temporary and is relevant to the current statement during the simulation. By doing so, the time cost needed for boolean program repair is greatly reduced and become more acceptable.

Boolean repair reverse converter is responsible for converting boolean statement back to respective C statement and apply simplification based on its semantic. Note that the repairing expression will maintain to be in DNF in the whole process. Assume the number of clauses in the DNF expression is N, and the maximum number of predicates in each clause is M. During the process of

inner-clause simplification, predicates will be sorted based on their semantics, and those predicates with similar semantics will be combined as one (see section 3.3.3). The sorting of predicates can be implemented by quick sort, whose complexity is $O(M \log M)$. After the predicates are in order, those predicates with combinable semantics will only appear consecutively, so the complexity of finding combinable predicates is $O(M)$. In this sense, the complexity of simplification for all clauses is $O(NM \log M)$. Each pair of clauses will be checked for whether their predicates have same or exactly opposite meanings in the process of inter-clause simplification (see section 3.3.2), hence the complexity is $O(N^2 M^M)$. As for outer-clause simplification, the satisfiability of each clause will be examined, which can be reduced to SMT problem (see section 3.3.1). SMT program is an NP-complete problem, which can be verified in polynomial time, so the complexity of outer-clause simplification will be $O(2^M)$ in the worst case. But outer-clause simplification actually preforms after inner-clause simplification, which can reduce the number of predicates in clauses, hence reduce the time cost needed for outer-clause simplification. In summary, the time complexity of boolean repair reverse conversion is $O(NM \log M + N2^M + N^2 M^2)$.

Table 2: Statistics of TCAS Test Cases

Version	Fault Type	Detailed Fault Type
5,21,22,23,24,27,41	Assignment Statement Fault	Logical Error in Assignment
1,3,4,6,9,12,20,25,39	Assignment Statement Fault	Wrong Operator
13,14,16,17,18,19,36,38	Declaration Error	Declaration Error
2,28,29,30,35,37	Assignment Statement Fault	Wrong 'return' Statement
7,8	Assignment Statement Fault	Initialization Error
34	Conditional Statement Fault	Wrong Condition Expression

5. Experiment

To verify the effectiveness and correctness of the methods and tool described in former sections, detailed experimental results and analysis will be given in this following section. In the experiment, the program repair tool will be used to repair some programs and evaluated based on its accuracy and efficiency.

The experiment mainly tested the tool on its ability to convert and repair assignment statements and conditional control flow statements (`if`). To be more specific, Several test cases of TCAS program from Siemens Suite, which has been used as a benchmark to evaluate the effectiveness of many testing techniques, are used in this experiment. More customized programs are added to the test cases to test the tool's ability of repairing boolean expression faults in loop statements, which Siemens' TCAS fails to cover.

5.1. TCAS of Siemens Suite

5.1.1. Introduction

The TCAS program used in the experiment comes from Siemens Suite, which has been used by other recent works on fault localization[32, 33]. TCAS, or traffic collision avoidance system, is an aircraft collision avoidance system designed to reduce the incidence of mid-air collisions between aircraft[34]. Siemens Suite provides a correct TCAS program, along with 41 different faulty versions. Additionally, 1608 different test cases are provided in Siemens Suite. The program has 173 lines of statements, including 25 global variables and 8 functions except for `main` function. Among the 41 given faulty versions, $V10$, $V11$, $V11$, $V15$, $V31$, $V31$, $V32$, $V33$, $V40$ contain multiple faults in one file, which is not supported by the tool, hence they will not be considered in the experiment. The statistics of the rest 34 versions is listed in table 2, in which assignment statement fault and conditional control flow statement fault are supported by the tool.

5.1.2. Preprocess on Test Cases

To execute the TCAS program, designated input is needed, but the repair tool we design assumes the given C program already includes test case. In this case, for each faulty version, one or more test cases will be selected to assign to the respective input variables, while new `assert` statement will be added just

before the end of `main` function to examine the program's output. By doing so, the given input and expected output of the test cases give out the specification of the program's behavior.

In the TCAS repairing experiment, we introduced different numbers of test cases to one faulty version, hoping to prove the effectiveness of the multi-test-case repair method we described in section 4.2.1.

5.1.3. Experimental Results

We experimented on the 26 faulty versions listed in table 2 whose faults are supported by the tool. For the faults of these versions, the repairing method which uses multiple test cases managed to locate all of them.

The results of 10 of them are listed in table 3. Meanings of columns in table 3 are listed below:

1. *Version* : the version number.
2. V : the number of global and relevant boolean variables.
3. $LoBP_1$: the length of converted boolean program when only one test case is introduced.
4. $Time_1(s)$: the time in seconds used to repair the given boolean program when only one test case is introduced.
5. $Result_1$: the percentage of passed tests among the given 1608 tests when only one test case is introduced.
6. $LoBP_2$: the length of converted boolean program when multiple test cases are introduced.
7. $Time_2(s)$: the time in seconds used to repair the given boolean program when multiple test cases are introduced.
8. $Result_2$: the number of test cases which cover bad routes effectively and the number of test cases we use.

We shall give you a detailed insight into one of them. Takes $V1$ as an example, we used 6 test cases to find out a correct repair statement, which was $*rep = (b4)|(b3)|(b2)|(!b1)|(b0)$ with respective predicates of each variable listed below:

- b0 : $Down\_Separation - Positive\_RA\_Alt\_Thresh[Alt\_Layer\_Value] == 0$
- b1 : $Down\_Separation - Positive\_RA\_Alt\_Thresh[Alt\_Layer\_Value] <= 0$
- b2 : $Other\_Tracked\_Alt - Own\_Tracked\_Alt <= 0$
- b3 : $Down\_Separation - Positive\_RA\_Alt\_Thresh[Alt\_Layer\_Value] >= 0$
- b4 : $Down\_Separation - Up\_Separation >= 100$

Judging from the experimental result, $*rep$ is satisfiable. After applying the simplification rules we described in former section, we have a converted C repair statement `Down_Separation - Positive_RA_Alt_Thresh[Alt_Layer_Value] < 0 && Down_Separation - Up_Separation < 100 && Other_Tracked_Alt - Own_Tracked_Alt > 0`. After replacing the faulty statement with the above statement, the repaired program passed all test cases.

Table 3: Results of TCAS Test Cases

Version	V	$LoBP_1$	$Time_1(s)$	$Result_1$	$LoBP_2$	$Time_2(s)$	$Result_2$
1	6/22	806	66.6	95.0%	714	474.1	6/10
3	2/15	894	101.0	97.5%	749	717.4	3/10
4	10/16	697	132.9	100%	591	250.7	6/10
5	4/18	803	50.2	100%	506	339.5	5/10
6	5/19	717	43.3	99.8%	514	220.3	2/10
9	7/24	928	96.1	93.5%	815	751.2	6/10
12	4/17	816	358.3	99.9%	852	920.6	4/10
26	4/18	803	51.3	99.9%	52	526.7	6/10
27	4/18	803	50.8	99.9%	520	432.7	5/10
34	3/17	856	105.3	83.2%	835	665.6	4/10

5.2. Cases of Loop Fault

5.2.1. Introduction of Test Cases

Loop statement fault is the kind of fault which the TCAS from Siemens Suite fails to cover. In this case, two groups of customized programs are added to the experiment. Both groups implement the algorithm of finding the maximum value of a given array. The difference is, one group is implemented by `while` loop while the other is `for` loop.

Two faulty versions of both programs are respectively given in Figure 7, in which fault 1 is located in the condition expression of loop statement, while fault 2 is located in the `if` statement in the loop body.

5.2.2. Experimental Results

For loop statement fault, one test case is sufficient for finding the repairing statement. The results are listed in table 4. Meanings of columns in table 4 are listed below:

1. *Version*: the version number of the corresponding version.
2. V : the number of global and relevant boolean variables.
3. $LoBP_1$: the length of converted boolean program when five test cases are introduced.
4. $Time_1(s)$: the time in seconds used to repair the given boolean program when five test cases are introduced.
5. $Result_1$: the percentage of passed tests among the given 2000 tests when five test cases are introduced.
6. $LoBP_2$: the average length of converted boolean program when multiple test cases are introduced.
7. $Time_2(s)$: the time in seconds used to repair the given boolean program when multiple test cases are introduced.
8. $Result_2$: the number of test cases which cover bad routes effectively and the total number of test cases we use.

```
int a[5], i, max, count;

int main() {
  a[0] = 5;
  a[1] = 2;
  a[2] = 10;
  a[3] = 4;
  a[4] = 1;
  count = 5;

  max = a[0];
  i = 1;
  while (i < count) {  // Fault 1
    if (max > a[i])    // Fault 2
      max = a[i]
    i++;
  }

  assert(max >= a[0]);
  assert(max >= a[1]);
  assert(max >= a[2]);
  assert(max >= a[3]);
  assert(max >= a[4]);

  return 0;
}
```

(a) while loop

```
int a[5], i, max, count;

int main() {
  a[0] = 5;
  a[1] = 2;
  a[2] = 10;
  a[3] = 4;
  a[4] = 1;
  count = 5;

  max = a[0];
  for(i = 0; i < count; i++) {  // Fault 1
    if (max > a[i])             // Fault 2
      max = a[i]
  }

  assert(max >= a[0]);
  assert(max >= a[1]);
  assert(max >= a[2]);
  assert(max >= a[3]);
  assert(max >= a[4]);

  return 0;
}
```

(b) for loop

Figure 7: While loop and for loop faulty programs

Table 4: Results of Customized Test Cases

Version	V	$LoBP_1$	$Time_1(s)$	$Result_1$	$LoBP_2$	$Time_2(s)$	$Result_2$
$while_1$	4/40	177	0.38	89.6%	177	14.1	10/100
$while_2$	5/40	178	0.31	58.3%	178	2.7	2/100
for_1	4/40	177	0.26	89.6%	177	14	10/100
for_2	5/40	178	0.30	58.3%	178	2.9	2/100

Judging from the experimental results, the tool is capable of handling loop statement fault and conditional control flow statement fault, which provides convincing evidence of the tool's effectiveness.

In Appendix A, the concrete test cases we used in the experiment and the respective boolean repairs and converted C repairing statements will be listed respectively.

5.3. Conclusions

Based on the experimental results listed in the former sections, one can say the tool is capable of producing an effective and intuitive repairing statement for many kinds of faults, including assignment statement fault and conditional control flow statement fault, which generally exist in most faulty programs.

In 2005, Griesmayer managed to find the boolean repair automatically[13], but manual intervene is still needed to convert the boolean repair back to the original programming language. In this sense, Griesmayer's work failed to achieve complete automation for program repair. Judging from the experimental results, our tool is capable of completing the whole process of program repair automatically and producing an effective repairing statement.

In 2009, Forrest and Weimer for the first time apply evolutionary computation to repair a real-world software system[18]. Their repairing system was capable of repairing faults like infinite loop and stack overflow, but there was no experimental result that proved their tool supports assignment fault or control flow fault. The fault model we discussed in this paper focuses on logical faults of program statements, which makes a difference from Weimer's study.

6. Summary

In this paper, we illustrated the basic conversion rules from C program to boolean program, and by collecting bad routes for the converted program, we managed to produce an appropriate boolean repair for a faulty program. Examination of the satisfiability of the boolean repair is also introduced in the paper. At last, by simplifying the boolean repair, an effective and relatively intuitive C repairing statement is generated. A lot of tests are also performed on the program repair tool, proving that the method described in this paper is capable of repairing assignment statement faults and control flow statement faults of TCAS.

There are still more work to be done on the program repair tool. For the boolean program repair module, though it is capable of finding a correct repairing statement, it comes with significantly high time complexity. The time cost will grow exponentially as the number of variables grows, which limits its availability for real-world programs. In this case, further work can be done on optimizing the search algorithm used in this module. Besides, the fault model defined in this paper only supports those faults lying in one single statement, which include only a few kinds of faults. Hence, supporting more kinds of faults, such as infinite loop and missing statements, can also be considered for further improvement.

[1] E. Y. Shapiro, *Algorithmic program debugging*. MIT press, 1983.

[2] D. Gopinath, M. Z. Malik, and S. Khurshid, "Specification-based program repair using sat," in *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 173–188, Springer, 2011.

[3] H. Cleve and A. Zeller, "Locating causes of program failures," in *Proceedings of the 27th international conference on Software engineering*, pp. 342–351, ACM, 2005.

[4] R. D. Banker, S. M. Datar, C. F. Kemerer, and D. Zweig, "Software complexity and maintenance costs," *Communications of the ACM*, vol. 36, no. 11, pp. 81–94, 1993.

[5] B. Boehm, B. Clark, E. Horowitz, C. Westland, R. Madachy, and R. Selby, "Cost models for future software life cycle processes: Cocomo 2.0," *Annals of software engineering*, vol. 1, no. 1, pp. 57–94, 1995.

[6] E. Börger and A. Cisternino, *Advances in software engineering*. Springer, 2008.

[7] M. R. Lyu and V. B. Mendiratta, "Software fault tolerance in a clustered architecture: Techniques and reliability modeling," in *Aerospace Conference, 1999. Proceedings. 1999 IEEE*, vol. 5, pp. 141–150, IEEE, 1999.

[8] D. Garlan and B. Schmerl, "Model-based adaptation for self-healing systems," in *Proceedings of the first workshop on Self-healing systems*, pp. 27–32, ACM, 2002.

[9] T. Ball and S. K. Rajamani, "Boolean programs: A model and process for software analysis," tech. rep., Technical Report 2000-14, Microsoft Research, 2000.

[10] T. Ball and S. K. Rajamani, "Bebop: A symbolic model checker for boolean programs," in *SPIN Model Checking and Software Verification*, pp. 113–130, Springer, 2000.

[11] J.-M. Autebert, J. Berstel, and L. Boasson, "Context-free languages and pushdown automata," in *Handbook of formal languages*, pp. 111–174, Springer, 1997.

[12] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani, "Automatic predicate abstraction of c programs," in *ACM SIGPLAN Notices*, vol. 36, pp. 203–213, ACM, 2001.

[13] A. Griesmayer, R. Bloem, and B. Cook, "Repair of boolean programs with an application to c," in *Computer Aided Verification*, pp. 358–371, Springer, 2006.

[14] B. Jobstmann, A. Griesmayer, and R. Bloem, "Program repair as a game," in *Computer Aided Verification*, pp. 226–238, Springer, 2005.

[15] B. Demsky, M. D. Ernst, P. J. Guo, S. McCamant, J. H. Perkins, and M. Rinard, "Inference and enforcement of data structure consistency specifications," in *Proceedings of the 2006 international symposium on Software testing and analysis*, pp. 233–244, ACM, 2006.

[16] A. Arcuri and X. Yao, "A novel co-evolutionary approach to automatic software bug fixing," in *Evolutionary Computation, 2008. CEC 2008.(IEEE World Congress on Computational Intelligence). IEEE Congress on*, pp. 162–168, IEEE, 2008.

[17] S. Forrest, T. Nguyen, W. Weimer, and C. Le Goues, "A genetic programming approach to automated software repair," in *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pp. 947–954, ACM, 2009.

[18] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, "Automatically finding patches using genetic programming," in *Proceedings of the 31st International Conference on Software Engineering*, pp. 364–374, IEEE Computer Society, 2009.

[19] T. Ball and S. K. Rajamani, "The s lam project: debugging system software via static analysis," in *ACM SIGPLAN Notices*, vol. 37, pp. 1–3, ACM, 2002.

[20] E. Clarke, D. Kroening, N. Sharygina, and K. Yorav, "Satabs: Sat-based predicate abstraction for ansi-c," in *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 570–574, Springer, 2005.

[21] D. Kroening and E. Clarke, "Checking consistency of c and verilog using predicate abstraction and induction," in *Proceedings of the 2004 IEEE/ACM International conference on Computer-aided design*, pp. 66–72, IEEE Computer Society, 2004.

[22] S. Graf and H. Saïdi, "Construction of abstract state graphs with pvs," in *Computer aided verification*, pp. 72–83, Springer, 1997.

[23] M. A. Colón and T. E. Uribe, "Generating finite-state abstractions of reactive systems using decision procedures," in *Computer Aided Verification*, pp. 293–304, Springer, 1998.

[24] R. C. Seacord, D. Plakosh, and G. A. Lewis, *Modernizing legacy systems: software technologies, engineering processes, and business practices*. Addison-Wesley Professional, 2003.

[25] M. Fitting, *First-order logic and automated theorem proving*. Springer Science & Business Media, 2012.

[26] C. W. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli, "Satisfiability modulo theories.," *Handbook of satisfiability*, vol. 185, pp. 825–885, 2009.

[27] A. Cimatti, "Beyond boolean sat: Satisfiability modulo theories," in *Discrete Event Systems, 2008. WODES 2008. 9th International Workshop on*, pp. 68–73, IEEE, 2008.

[28] L. de Moura, B. Dutertre, and N. Shankar, "A tutorial on satisfiability modulo theories," in *Computer Aided Verification*, pp. 20–36, Springer, 2007.

[29] S. B. Akers, "Binary decision diagrams," *Computers, IEEE Transactions on*, vol. 100, no. 6, pp. 509–516, 1978.

[30] A. J. Hu, *Techniques for efficient formal verification using binary decision diagrams*. PhD thesis, stanford university, 1995.

[31] S. J. Friedman and K. J. Supowit, "Finding the optimal variable ordering for binary decision diagrams," in *Proceedings of the 24th ACM/IEEE Design Automation Conference*, pp. 348–356, ACM, 1987.

[32] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand, "Experiments of the effectiveness of dataflow-and controlflow-based test adequacy criteria," in *Proceedings of the 16th international conference on Software engineering*, pp. 191–200, IEEE Computer Society Press, 1994.

[33] G. Rothermel and M. J. Harrold, "Empirical studies of a safe regression test selection technique," *Software Engineering, IEEE Transactions on*, vol. 24, no. 6, pp. 401–419, 1998.

[34] W. Harman, "Tcas- a system for preventing midair collisions," *The Lincoln Laboratory Journal*, vol. 2, no. 3, pp. 437–457, 1989.