Cause Clue Clauses: Error Localization using Maximum Satisfiability *

Manu Jose
University of California, Los Angeles
mjose@cs.ucla.edu

Rupak Majumdar
MPI-SWS, Kaiserslautern &
University of California, Los Angeles
rupak@mpi-sws.org

Abstract

Much effort is spent by programmers everyday in trying to reduce long, failing execution traces to the *cause* of the error. We present an algorithm for error cause localization based on a reduction to the maximal satisfiability problem (MAX-SAT), which asks what is the maximum number of clauses of a Boolean formula that can be simultaneously satisfied by an assignment. At an intuitive level, our algorithm takes as input a program and a failing test, and comprises the following three steps. First, using bounded model checking, and a bound obtained from the execution of the test, we encode the semantics of a bounded unrolling of the program as a Boolean *trace formula*. Second, for a failing program execution (e.g., one that violates an assertion or a post-condition), we construct an *unsatisfiable* formula by taking the formula and additionally asserting that the input is the failing test and that the assertion condition does hold at the end. Third, using MAX-SAT, we find a maximal set of clauses in this formula that can be satisfied together, and output the complement set as a potential cause of the error.

We have implemented our algorithm in a tool called BugAssist that performs error localization for C programs. We demonstrate the effectiveness of BugAssist on a set of benchmark examples with injected faults, and show that in most cases, BugAssist can quickly and precisely isolate a few lines of code whose change eliminates the error. We also demonstrate how our algorithm can be modified to automatically suggest fixes for common classes of errors such as off-by-one.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging—Fault-localization; F.3.2 [*Logics and Meaning of Programs*]: Semantics of Programming Languages—Program Analysis

General Terms Verification,Reliability

Keywords Debugging, Fault localization, Maximum Satisfiability

* This research was sponsored in part by the NSF grant CCF-0546170 and the DARPA grant HR0011-09-1-0037.

PLDI'11, June 4–8, 2011, San Jose, California, USA.

1. Introduction

A large part of the development cycle is spent in debugging, where the programmer looks at a long, failing, trace and tries to localize the problem to a few lines of source code that elucidate the cause of the problem. We describe an algorithm for fault localization for software that automates this process. The input to our algorithm is a program, a correctness specification (either a post-condition, an assertion, or a "golden output"), and a program input and corresponding execution (called the *failing execution*) that demonstrates the violation of the specification. The output is a minimal set of program statements such that there exists a way to replace these statements such that the failing execution is no longer valid.

Internally, our algorithm uses symbolic analysis of software based on Boolean satisfiability, and reduces the problem to *maximum Boolean satisfiability*. It takes as input a program and a failing test case and performs the following three steps. First, it constructs a symbolic *trace formula* for the input program. This is a Boolean formula in conjunctive normal form such that the formula is satisfiable iff the program execution is feasible (and every satisfiable assignment to the formula corresponds to the sequence of states in a program execution). The trace formula construction proceeds identically to symbolic execution or bounded model checking algorithms [2, 6, 15].

Second, it extends the trace formula by conjoining it with constraints that ensure the initial state satisfies the values of the failing test and the final states satisfy the program post-condition that was failed by the test. The extended trace formula essentially states that starting from the test input and executing the program trace leads to a state satisfying the specification. Obviously, the extended trace formula for a failing execution must be unsatisfiable.

Third, it feeds the extended trace formula to a *maximum satisfiability solver*. Maximum satisfiability (MAX-SAT) is the problem of determining the maximum number of clauses of a given Boolean formula that can be satisfied by any given assignment. Our tool computes a maximal set of clauses of the extended trace formula that can be simultaneously satisfied, and takes the complement of this set as a candidate set of clauses that can be changed to make the entire formula satisfiable. Since each clause in the extended trace formula can be mapped back to a statement in the code, this process identifies a candidate localization of the error in terms of program statements. Note that there may be several minimal sets of clauses that can be found in this way, and we enumerate each minimal set as candidate localizations for the user. In our experiments, we have found that the number of minimal sets enumerated in this way remains small.

More precisely, our algorithm uses a solver for *partial* MAX-SAT. In partial MAX-SAT, the input clauses can be marked *hard* or *soft*, and the MAX-SAT instance finds the maximum number of soft

clauses that can be satisfied by an assignment which satisfies every hard clause. In our algorithm, we mark the input constraints (that ensure that the input is a failing test) as well as constraints arising from the specification are hard. This is necessary: otherwise, the MAX-SAT algorithm can trivially return that changing an input or changing the specification can eliminate the failing execution. In addition, in our implementation, we group clauses arising out of the same program statement together, thus keeping the resulting MAX-SAT instance small.

We have implemented our algorithm in a tool called BugAssist for fault localization of C programs.[1] BugAssist takes as input a C program with an assertion, and a set of failing test cases, and returns a set of program instructions whose replacement can remove the failures. It builds on the CBMC bounded model checker for construction of the trace formula and an off-the-shelf MAX-SAT solver [20] to compute the maximal set of satisfied clauses. We demonstrate the effectiveness of BugAssist on 5 programs from the Siemens set of benchmarks with injected faults [8]. The TCAS program in the test suite is run with all the faulty versions in detail to illustrate the completeness of the tool. In each case, we show that BugAssist can efficiently and precisely determine the exact (to the human) lines of code that form the "bug". The other 4 programs are used to show the scalability of the tool when used in conjunction with orthogonal error trace reduction methods.

We can extend our algorithm to suggest *fixes* for bugs automatically, by noticing that the MAX-SAT instance can be used not only to localize problems, but also to suggest alternate inputs that will eliminate the current failure. In general, this is an instance of Boolean program synthesis, and the cost of the search can be prohibitive. However, we have experimentally validated that automatic suggestions for fixes is efficient when we additionally restrict the search to common classes of programmer errors, such as replacement of comparison operators (e.g., $<$ by $\leq$) or off-by-one arithmetic errors. For these classes of systems, BugAssist can automatically create suggestions for program changes that eliminate the current failure.

Error localization is an important step in debugging, and improved automation for error localization can speed-up manual debugging and improve the usability of automatic error-detection tools (such as model checkers and concolic testers). Based on our implementation and experimental results, we feel BugAssist is a simple yet precise technique for error localization.

Related Work. Fault localization for counterexample traces has been an active area of research in recent years [1, 12, 13, 22, 23]. Most papers perform localization based on multiple program runs, both successful and failing, and defining a heuristic metric on program traces to identify locations which separate failing runs from successful ones.

Griesmayer et al. [12] gives a fault localization algorithm for C programs that constructs a modified system that allows a given number of expressions to be changed arbitrarily and uses the counter example trace from a model checker. This requires instrumenting each expression e_i in the program with ($\texttt{diag} == i?\texttt{nondet}() : e_i$), where `diag` is a non deterministic variable and $\texttt{nondet}()$ is a new variable with the size equal to that of e_i. The number of diagnosis variables is equal to the number of components that are faulty in the program and need to be analyzed before creating the modified system. So each expression in the program requires a new variable in the modified system along with the diagnosis variables which could blow up the size of the instrumented program under consideration. In this work we avoid these drawbacks using selector variables and efficient MAX-SAT instance formulation using clause grouping technique.

Many existing algorithms for fault localization [1, 13, 23, 33] use the difference between the faulty trace and a number of successful traces. For example, Ball, Naik, and Rajamani [1] use multiple calls to a model checker and compare the counterexamples to a successful trace. The faults are those transitions that does not appear in a correct trace. Our approach does not require comparing the traces or a successful run of the program as benchmark. We report the exact locations where the bug could be corrected instead of a minimal code fragment or a fault neighbor location.

Alternate approaches to reducing the cognitive load of debugging are *delta debugging* [33], where multiple runs of the program are used to minimize the "relevant" portion of the input, and *dynamic slicing* [30], where data and control dependence information is used to remove statements irrelevant to the cause of failure. Our technique is orthogonal to delta-debugging and dynamic slicing, and can be composed profitably. In fact, we demonstrate in our experiments how a combination of dynamic slicing and delta debugging, followed by our technique, can allow us to localize faults in long executions.

While we describe our algorithm in pure symbolic execution terms, our algorithm fits in very well with concolic execution [3, 11, 25], where symbolic constraints are generated while the concrete test case is run. Our motivation for using CBMC was the easy integration with MAX-SAT solvers, but in our implementation, we performed some optimizations (such as using concrete values for external library calls in the trace formula and constant-folding input-independent parts of the constraints) similar to concolic execution.

Unsatisfiability cores and MAX-SAT have been used successfully for design debugging of gate-level hardware circuits [5, 24]. Unsatisfiability cores have also been used to localize over-constrains in declarative models [26].

2. Motivating Example

Program 1 A simple example.

```
int Array[3];
int testme(int index)
  {
  .   ..................
  1  if ( index != 1) /* Potential Bug 2 */
  2     index = 2;
  3  else
  4     index = index + 2; /* Potential Bug 1 */
  .   ..................
  .   ..................
  .
  5  i = index;
  6  return Array[i]; //assert(i >= 0 && i < 3)
  }
```

We start with an informal description of BugAssist. Consider the function `testme` in Program 1 which returns a value at a new location from an array of size 3. The global array `Array` has 3 elements. The function takes in the current index value, does computation on this value (shown in lines 1–4) to find a new index and returns the value in the array at the new index (line 6). The array dereference on line 5 generates implicit assertions about the array bounds shown in line 6.

The program has a bug. If the input `index` is equal to 1, then the else-branch sets `index` to 3, and the subsequent array dereference

[1] The tool, Eclipse plugin, and test cases can be downloaded from our web page http://bugassist.mpi-sws.org.

on line 6 is out of bounds. Testing the program with this input will find the bug, and return a program trace that shows the array bounds violation at the end. But testing or model checking returns a *full execution* path, including details irrelevant to the specific bug, and do not give the reason for failure, or the cause of the bug. The localization algorithm in BugAssist helps to nail down the issue to a few potential bug locations in the program where the correction has to be made.

BugAssist works as follows. Starting with the test input $\texttt{index} = 1$ and the program, it first constructs a symbolic *trace formula* TF encoding the program semantics:

$$\begin{aligned} \mathsf{TF} \equiv \texttt{guard}_1 = (\texttt{index}_1 \neq 1) \wedge \\ \texttt{index}_2 = 2 \wedge \texttt{index}_3 = \texttt{index}_1 + 2 \wedge \\ \texttt{i} = \texttt{guard}_1 ? \texttt{index}_2 : \texttt{index}_3 \end{aligned}$$

Every satisfying assignment to the trace formula gives a possible execution of the program, and conversely. We assume that integers and integer operations are encoded in a bit-precise way, and without loss of generality, the trace formula is a Boolean formula in conjunctive normal form. In case there are loops in the program, we unroll loops up to a bound computed from the execution of the test input on the program (roughly, we take the bound as the maximum number of times any loop gets executed along the execution, taking nesting into account). We omit the details of the standard encoding from imperative programs to Boolean formulas (see, e.g., [6]).

Clearly, at the end of the trace, the assertion

$$\texttt{i} < 3$$

does not hold for all inputs to the program. Consider now the formula

$$\Phi \equiv \underbrace{\texttt{index}_1 = 1}_{\text{test input}} \wedge \underbrace{\mathsf{TF}}_{\text{trace formula}} \wedge \underbrace{\texttt{i} < 3}_{\text{assertion}}$$

which is unsatisfiable. Intuitively, the formula captures the execution of the program starting with the error-inducing test input, and asserts that the assertion holds at the end (a contradiction, by choice of the input).

We convert Φ to conjunctive normal form (CNF) and feed it to a partial MAX-SAT solver [20]. A partial MAX-SAT solver takes as input a Boolean formula in CNF where each clause is marked "hard" or "soft," and returns the maximum number of soft clauses (as well as a subset of clauses of maximum cardinality) that can be simultaneously satisfied by an assignment satisfying all the hard clauses. In case of Φ, we make the constraints coming from the test input ($\texttt{index}_1 = 1$) and the assertion ($\texttt{i} < 3$) as hard, and leave the clauses in the trace formula soft. Intuitively, we ask, given that the input and the assertion are fixed, which parts of the trace formula are consistent with the input and the assertion? The partial MAX-SAT solver then tries to find a set of soft clauses of maximum cardinality which can be simultaneously satisfied while satisfying all the hard clauses. The complement of a set of maximally satisfiable clauses (CoMSS) gives a set of soft clauses of minimum cardinality whose removal would make Φ satisfiable, i.e., consistent with the view that the test input does not break the assertion. By tracing the origins of the clauses in this set to the program, we get a set of program locations that are potential indicators of the error. Using clause grouping, described in Section 3, each line in the program is mapped to a bunch of its soft clauses which are enabled and disabled simultaneously.

In our example, the hard and soft clauses are:

$$\begin{aligned} \text{Hard} &: \texttt{index}_1 = 1 \wedge \texttt{i} < 3 \\ \text{Soft} &: \mathsf{TF} \end{aligned}$$

MAX-SAT returns that a possible CoMSS maps to the line 4 in the program. This is the unsatisfiable core whose removal or correction can satisfy the formula Φ. We claim that is a potential error location for the program and a fix would be to change the constant to any integer less than 2 and greater than -2.

If the programmer decides this is not a correct localization, we can generate additional localization candidates as follows. We iterate by making another call to MAX-SAT, but this time make clauses arising out of line 4 hard, i.e., asking the MAX-SAT for possible CoMSS where line 4 is kept unchanged. This reveals another potential bug location in the code. We repeat this process until MAX-SAT finds the formula to be unsatisfiable and such that no clauses can be removed to make the instance satisfiable. The error locations reported by BugAssist are underlined in Program 1. On a closer look, these are all the places where the correction can be made. Either changing the constant value at line 4 or the conditional statement at line 1 can fix the program. Our method is limited by the existing code: we cannot "localize" errors that can only be fixed by adding additional code.

Notice that our technique is stronger than simply taking the backward slice of the program trace, and gives fine-grained information about potential error locations. The backward slice for this trace contains all the lines 1, 4, and 5. Our algorithm returns lines 1 and 4 separately as potential error locations. However, slicing is an orthogonal optimization which can be applied before applying our technique.

So far we have focused on error localization. The methodology can be modified to suggest program repairs as well. Intuitively, the fault localization returns a set of program commands that are likely to be wrong. One can then ask, what are potential replacements to these commands that fixes the error? In general, the space of potential replacements is large, and searching this space efficiently is a difficult problem of program synthesis [27, 29]. Instead, we take a pragmatic approach and look for possible fixes for common programmer errors.

Specifically, we demonstrate our idea by fixing "*off by one*" errors. In this example, the error occurs due to accessing an out of bound array element by one. When BugAssist comes back with line 4 as a potential bug location, we try to "fix" the bug by changing the constant whose new value is one off its current value. So we change the value 2 in this line to 3 or 1 and check if either of these values satisfy the properties. This involves modifying the trace formula appropriately and checking if the failing program execution becomes infeasible with either change. So in this case we create two programs with new constants at line 4 as follows.

$$\begin{aligned} Program1: \;\; & index = index + 3 \;\times \\ Program2: \;\; & index = index + 1 \;\surd \end{aligned}$$

The new value 1 ensures that the error path is infeasible, and this can be used as a suggestion for repair for the program. The same procedure can be used to check for operator errors like use of plus instead of minus, division instead of multiplication, performing assignment instead of equality test, etc., which are common programmer error patterns.

3. Preliminaries

3.1 Programs: Syntax and Semantics

We describe our algorithm on a simple imperative language based on control-flow graphs. For simplicity of description, we omit features such as function calls or pointers. These are handled by our implementation.

A *program* $G = (X, \mathcal{L}, \ell_0, \mathcal{T})$ consists of a set X of Boolean-valued variables, a set $\mathcal{L}$ of *control locations*, an initial location $\ell_0 \in \mathcal{L}$ and a set $\mathcal{T}$ of *transitions*. Each transition $\tau \in \mathcal{T}$ is a tuple

(ℓ, ρ, ℓ') where ℓ and ℓ' are control locations and ρ is a constraint over free variables from $X \cup X'$, where the variables from X' denote the values of the variables from X in the next state.

For a constraint ρ, we sometimes write $\rho(X, X')$ to denote that the free variables in ρ come from the set $X \cup X'$.

Our notation is sufficient to express common imperative programs (without function calls): the control flow structure of the program is captured by the graph of control locations, and operations such as assignments $x := e$ and assumes $\mathsf{assume}(p)$ captured by constraints $x' = e \wedge \bigwedge \{y' = y \mid y \in X \setminus \{x\}\}$ and $p \wedge \bigwedge \{x' = x \mid x \in X\}$ respectively. A program is *loop-free* if there is no syntactic loop in the graph of control locations.

A *state* of the program $\mathcal{P}$ is a mapping from variables in X to Booleans. We denote the set of all program states by $v.X$. A *computation* of the program is a sequence $\langle m_0, s_0\rangle\langle m_1, s_1\rangle \ldots \in (\mathcal{L} \times v.X)^*$, where $m_0 = \ell_0$ is the initial location, and for each $i \in \{0, \ldots, k-1\}$, there is a transition $(m_i, \rho_i, m_{i+1}) \in \mathcal{T}$ such that (s_i, s_{i+1}) satisfies the constraint ρ_i.

An *assertion* p is a set of program states. A program *violates* an assertion p if there is some computation $\langle m_0, s_0\rangle \ldots \langle m_k, s_k\rangle$ such that s_k is not in p. Typically, assertions can be given as language-level correctness requirements (e.g., "no null pointer dereference"), as programmer-specified `asserts` in the code, or as post-conditions.

3.2 Trace Formulas

Given a program and a bound $k > 0$, we can unwind the graph of control locations to get a simplified program without loops whose computations all have length at most k and such that each computation of the simplified program is also a computation of the original program. From such a loop-free program, we can derive a (quantifier-free) Boolean formula, called the *trace formula*, such that the set of satisfying assignments to the formula correspond exactly to computations of the program. We briefly describe the construction; see [6] for details.

The construction of the trace formula takes a loop-free program $\mathcal{P}$, all of whose computations have length at most k, and recursively constructs a Boolean formula as follows. Let $X_0, \ldots, X_k$ be independent copies of the set of variables X. For each $\ell \in \mathcal{L}$ and $i \in \{0, \ldots, k-1\}$, let z_i^ℓ be a Boolean variable, and define a constraint $\phi(\ell, i)$ as follows:

$$z_i^\ell \leftrightarrow \bigvee_{(\ell,\rho,\ell') \in \mathcal{T}} \rho(X_i, X_{i+1}) \wedge z_{i+1}^{\ell'} \tag{1}$$

The trace formula is then defined to be the conjunction over $\ell \in \mathcal{L}$ and $i \in \{0, \ldots, k-1\}$ of the constraints in Equation (1), together with the conjunct $z_0^{\ell_0}$:

$$z_0^{\ell_0} \wedge \bigwedge_{\ell \in \mathcal{L}, i \in \{0,\ldots,k-1\}} \phi(\ell, i) \tag{2}$$

The construction is well-defined because $\mathcal{P}$ is loop-free.

While we have described trace formulas for our simple programs, a C program with finite bit width data, e.g., 32-bit integers, can be converted into an equivalent Boolean program by separately tracking each bit of the state, and by interpreting fixed-width arithmetic and comparison operators as corresponding Boolean operations on each individual bit. In particular, our implementation handles all features of ANSI-C, including function calls and pointers. We omit the (standard) details, see e.g., [6, 32].

3.3 Partial Maximum Satisfiability

Given a Boolean formula in conjunctive normal form, the *maximum satisfiability* (MAX-SAT) problem asks what is the maximum number of clauses that can be satisfied by any assignment [16]. The MAX-SAT decision problem is NP-complete; note that a formula is satisfiable iff all its clauses can be satisfied by some assignment.

The *partial maximum satisfiability* (pMAX-SAT) problem takes as input a Boolean formula Φ in conjunctive normal form, and a marking of each clause of Φ as *hard* or *soft*, and asks what is the maximum number of soft clauses which can be satisfied by an assignment to the variables which satisfies all hard clauses. Intuitively, each hard clause must be satisfied, and we look for the maximum number of soft clauses which may be satisfied under this constraint.

Recent years have seen a tremendous improvement in engineering efficient solvers for MAX-SAT and pMAX-SAT. The widely used algorithm for MaxSAT is based on branch-and-bound search [17], supported by effective lower bounding and dedicated inference techniques. Recently, unsatisfiability based MaxSAT solvers by iterated identification of unsatisfiable sub-formulas was proposed in [10]. This approach consist of identifying unsatisfiable sub-formulas and relaxing clauses in each unsatisfiable sub-formulas by associating a relaxation variable with each such clause. Cardinality constraints are used to constrain the number of relaxed clauses [19, 20].

In addition to solving the decision problem, MAX-SAT solvers also give a set of clauses of maximum cardinality that can be simultaneously satisfied. The complement of these maximum satisfiable subsets (MSS) are a set of clauses whose removal makes the instance satisfiable (CoMSS). Since the maximum satisfiability subset is maximal, the complement of this set is minimal [18].

In this work we make use of these CoMSS which refers to the clauses whose removal can make the system satisfiable. Since we represent a C program as a boolean satisfiability problem with constraints and properties, the CoMSS are oracles for potential bug locations.

3.4 Efficient Compilation to MAX-SAT

A single transition can lead to multiple clauses in the conjunctive normal form of the trace formula. In this section we provide a method to simplify the MAX-SAT problem by grouping together clauses arising out of a single transition in the program.

For each transition $\tau = (m, \rho, m') \in \mathcal{T}$, we introduce a new Boolean variable λ_τ. Let $\Lambda = \{\lambda_\tau \mid \tau \in \mathcal{T}\}$. Let $\mathsf{CNF}(\rho)$ be a conjunctive normal form representation of ρ. We augment each clause in $\mathsf{CNF}(\rho)$ with λ_τ. For example, suppose $(c_1^1 \vee \ldots) \wedge (c_1^2 \vee \ldots)$ is a conjunctive normal form representation of ρ, then the augmented representation is $(\neg\lambda_\tau \vee c_1^1 \vee \ldots) \wedge (\neg\lambda_\tau \vee c_1^2 \vee \ldots)$.

The augmentation with λ_τ has the following effect. When λ_τ is assigned `true`, the original clauses in the CNF representation of ρ must be satisfied, while when λ_ρ is assigned `false`, each augmented clause is already satisfied. This helps to enable and disable the clauses corresponding to each transition by setting and unsetting the λ_τ variable respectively. The λ-variables are called *selector variables*.

We now augment trace formulas with selector variables. Let $\phi'(\ell, i, \Lambda)$ be the formula in which each clause arising out of $\tau = (\cdot, \rho, \cdot)$ is augmented with λ_τ. Instead of Equation (2) for the trace formula, we use the form:

$$\underbrace{z_0^{\ell_0} \wedge \bigwedge_{\ell \in \mathcal{L}, i \in \{0,\ldots,k-1\}} \phi'(\ell, i, \Lambda)}_{\mathsf{TF}_1} \wedge \underbrace{\bigwedge_{\tau \in \mathcal{T}} \lambda_\tau}_{\mathsf{TF}_2} \tag{3}$$

where we label the two parts of the formula TF_1 and TF_2 for later reference. Intuitively, clauses from TF_1 will be marked as hard clauses to the MAX-SAT solver, and clauses from TF_2 will be marked soft. Thus, the MAX-SAT solver will explore the space of possible program statements whose replacement will cause the error to go away.

Algorithm 1 Localization Algorithm

Input: Program $\mathcal{P}$ and assertion p
Output: Either p holds for all executions or potential bug locations

```
1:  (test, σ) = GenerateCounterexample(P, p)
2:  if σ is "None" then
3:     return "No counterexample to p found"
4:  else
5:     Φ_H = [[test]] ∧ p ∧ TF_1(σ)
6:     Φ_S = TF_2(σ)
7:     while true do
8:        BugLoc = CoMSS(Φ_H, Φ_S)
9:        if BugLoc = ∅ then
10:          return "No more suspects"
11:       else
12:          output "Potential bug at CoMSS BugLoc"
13:          β = ⋁ {λ_i | λ_i ∈ BugLoc}
14:          Φ_S = Φ_S \ β and Φ_H = Φ_H ∪ β
```

Notice that we allocate a selector variable for each transition of the program, so the number of selector variables is bounded by the size of the program. However, in a trace, the same program transition may occur multiple times (e.g., on unrolling a loop), and there is a distinct clause for each of these occurrences all tagged with the same selector variable.

We use the abstraction technique on transitions, which correspond to line numbers of code in our implementation, but it is also possible to group the clauses from modules and recursively narrow down the problem to a module, and then to a line.

4. Algorithm

We now describe the algorithm for BugAssist. There are two phases of the algorithm: first, generate a failing execution (and a test demonstrating a failing execution), and second, find a minimal set of transitions that can render the failing execution infeasible.

4.1 Generating Failing Tests

In our implementation, we use either failing test cases from a test suite as a starting point. If there are no available tests, we use *bounded model checking* [2, 6] to systematically explore program executions and look for potential assertion violations. Once a failing execution is found, the bounded model checking procedure can generate a concrete initial state that leads to the assertion violation as well as the trace formula.

4.2 The Localization Algorithm

Algorithm 1 shows the BugAssist localization algorithm. Line 1 calls the procedure to generate failing executions for the assertion. If no failing executions are found, the procedure returns. Otherwise, we get a concrete test case test as well as a trace σ demonstrating the failure of the assertion.

Using the test, the failing execution, and the assertion, we construct two formulas (lines 5,6). The formula Φ_H consists of three parts. The first part, $[\![\mathsf{test}]\!]$, is a formula asserting that the initial state coincides with the test case that caused the failure. Formally, for a program state s, the constraint $[\![s]\!]$ is defined as $\bigwedge \{x = s(x) \mid x \in X\}$. The second part is the assertion p. The third part is the first part $\mathsf{TF}_1(\sigma)$ of the trace formula from Equation (3). The formula Φ_S is the second part $\mathsf{TF}_2(\sigma)$ of the trace formula from Equation (3).

Notice that $\Phi_H \wedge \Phi_S$ is unsatisfiable. (Intuitively, it says that if the program is run with the test input test, then at the end of the execution trace σ, the assertion p holds.)

In subsequent calls to pMAX-SAT, clauses in Φ_H are treated as hard clauses, and clauses in Φ_S are treated as soft clauses. Intuitively, treating Φ_S as soft clauses enables us to explore the effect of changing each subset of transitions to see if the failing transition can be made infeasible.

The search for localizations is performed in the **while** loop of lines 7–14. During each iteration of the while loop, we call the pMAX-SAT solver and get a CoMSS for the current (Φ_H, Φ_S) pair. Each of these clauses returned by CoMSS gives potential bug locations in the code, and is output to the programmer.

Whenever we report a potential bug, we add a hard blocking clause for the corresponding CoMSS, so that in subsequent iterations, this CoMSS is not explored again as a potential cause of error. In many of our experiments, the CoMSS returns a single λ_ρ clause as the indicator of error. In general, it returns more than one selector variable which indicates that the program cannot be fixed by changing any one line but must be changed at multiple locations. (This does happen in experiments.) Adding each of these λ_ρ variables as a new hard clause blocks the occurrence of these clauses in a different clause combination. To avoid this problem, we compute a blocking clause β (lines 13) and make the blocking clause hard. For example, suppose the coMSS returned is, $BugLoc = \{\lambda_1, \lambda_2, \ldots, \lambda_k\}$. This means that the bug can be fixed by making simultaneous changes to these k locations. In the next iteration, we add a new hard clause $(\lambda_1 \vee \ldots \vee \lambda_k)$ which ensures that this particular CoMSS is not encountered again, but other combinations of these locations are still allowed.

4.3 Dealing with Multiple Locations

BugAssist may return multiple locations where a correction is possible. The experimental results in section 6.1 shows that the number of potential error locations returned is quite small and, in most cases, the exact bug location is reported using a single failing execution. However, for reliability and further refinement of bug locations, we use a ranking mechanism for bug locations by running the localization algorithm repeatedly with different failing program traces and ranking the bug locations based on their frequency of appearance in each of these runs. For test cases, this requires access to a set of tests that all fail the assertion. For counterexample generation using bounded model checking, we take the Boolean formula constructed by the bounded model checker and generate multiple satisfying assignments by changing the order of variables in the SAT algorithm [7] or by doing a random restart of the solver. Running BugAssist with these new values gives a another set of potential bug locations. Repeating this process and ranking the bug locations can narrow down the search to a few lines in the program.

5. Extensions

We now describe two extensions to the basic algorithm.

5.1 Extension 1: Automated Repair

BugAssist can be extended to suggest potential repairs automatically. In general, program repair reduces to program synthesis. We sacrifice generality for practicality by focusing on specific program repairs inspired by program mutation testing and checking if there is a possible repair from this class that can remove the current test failures. For example, if there is a constant used in a potential error location, we try to synthesize a new constant which can fix the code [12], or if there is an operator used in a potential error location, we try to generate a repair by mutating the operator to a different one. We demonstrate this capability by generating suggestions for fixing *off-by-one errors* [31] in the program. These are a common class of logical errors in programs arising when programmers use an expression e in the program when they should be using $e \pm 1$. For

example, this can happen if programmers forget that a sequence starts at zero rather than one (e.g. array indices in many languages like C, C++). It is also caused during boundary check conditions by using a $<$ instead of $\leq$ or vice versa.

During the code parsing phase, we mark lines which have constants in them. After running BugAssist on the code, we look at potential error locations, and for each constant c in this set, we introduce an indicator variable i_c that takes values in the set $\{-1, 0, 1\}$. We replace the constant c with the expression $c + i_c$ in the code. Now, we ask if there exist values to the indicator variables that ensures that the new trace is infeasible. This is a Σ_2 query (does there exist values of i_c such that for all inputs the trace formula is infeasible), and cannot be directly solved by a SAT solver (see [28, 29] for extensions to SAT solvers to solve this problem). In our implementation, we restrict the number of non-zero indicator variables and iteratively call a SAT solver for each assignment of the indicator variables.

5.2 Extension 2: Debugging Loops

Bugs within loop bodies can be particularly hard to debug as they might be hidden in initial iterations and only visible afterwards. The usual bounded model checking methodology to verify properties is by unwinding loops by duplicating the loop body K times for a limit K on the number of unwindings. The programmer would be interested in knowing the iteration at which the assertion is violated to get a better idea about the cause of the error. We suggest a method to catch the potential iteration of the loop where the bug appeared first.

We can do this by grouping clauses and assigning weights to the soft clauses in the pMAX-SAT instance, and using a *weighted* version of the pMAX-SAT algorithm. Each time a loop body is duplicated (till the bound K), we create a new selector variable. For example, for a transition $\tau = (m, \rho, m') \in \mathcal{T}$ in the loop body, during the i^{th} unwinding, we augment each clause arising out of ρ with λ_τ^i. We add these selector variables as soft clauses to the pMAX-SAT instance as before, but additionally assign a weight as follows:

$$Weight(\lambda_\tau^i) = \alpha + K - i \tag{4}$$

for each $i = 1, \ldots, K$, where α is some default weight for soft clauses. This makes sure that the clauses corresponding to the initial iterations of the loop gets a higher weightage. The weights assigned to the soft clauses in the pMAX-SAT can be thought of as the penalty that has to be paid to falsify the clauses. The solver extracts the CoMSS in such a way that clauses from initial iterations are preferred over clauses from later iterations, since the former have higher weights. This helps to localize the first iteration of the loop which can reproduce the failure.

6. Experimental Results

We now demonstrate the capability of the tool by showing the results from running programs from the Siemens test suite [8]. The Siemens test suite is widely used in the literature for bug localization studies [12, 23]. In Section 6.1, we analyze a simple TCAS program from the Siemens suite [14] in depth and in Section 6.2 we illustrate the scalability of our method using more complex examples.

In our implementation, we used the bounded model checker CBMC [6] to generate failing test inputs as well as to construct the trace formula for an unrolling of the program. For solving the pMAX-SAT instances, we used the *Maximum Satisfiability with UNsatisfiable COREs* (MSUnCORE) tool [20], which can handle large and complex weighted partial MAX-SAT problems. Fixes for off-by-one errors were synthesized using the MiniSAT2 [9] SAT

```
1 int Inhibit_Climb () {
2   return (Climb_Inhibit?Up_Sep+300:Up_Sep);
3   /*return (Climb_Inhibit?Up_Sep+100:Up_Sep);*/
4 }
5 int Non_Crossing_Climb() {
6    upward_preferred=Inhibit_Climb()>Down_Sep;
7    if (upward_preferred) {
8         result = !(Own_Below_Threat()) ||
                  (!(Down_Sep >= ALIM()));  }
9      else{
10       result =  (Cur_Vertical_Sep >= 100)
                  && (Up_Sep >= ALIM());  }
11    return result;
12 }
13 int Non_Crossing_Descend() {
14    upward_preferred=Inhibit_Climb()>Down_Sep;
15    if (upward_preferred) {
16       result = Own_Below_Threat() &&
              (Cur_Vertical_Sep >= 100) &&
                    (Down_Sep >= ALIM());}
17     else{
18       result = !(Own_Above_Threat()) ||
              ((Own_Above_Threat()) &&
              (Up_Sep >= ALIM())); }
19     return result;
20 }
21  int alt_sep_test() {
22      enabled = true; /*conditions omitted*/
23      alt_sep = UNRESOLVED;
24      if (enabled) {
25         need_upward_RA=Non_Crossing_Climb()&&
                              Own_Below_Threat();
26         need_downward_RA=Non_Crossing_Descend()
                              && Own_Above_Threat();
27         if (need_upward_RA && need_downward_RA)
28             alt_sep = UNRESOLVED;
29         else if (need_upward_RA)
30             alt_sep = UPWARD_RA;
31         else if (need_downward_RA)
32             alt_sep = DOWNWARD_RA;
33      }
34      return alt_sep;
35 }
36 int main() {/*inputs omitted*/
37    assert(alt_sep_test() == DOWNWARD_RA);
38 }
```

Figure 1. Sample TCAS code. Potential bug locations identified by BugAssist are underlined. Original code on line *3*; mutation on line *2*

engine. All our experiments are preformed on an 3.16 GHz Intel Core 2 Duo CPU with 7.6 GB RAM.

6.1 TCAS Experiments

The TCAS task of the Siemens test suite implements an aircraft collision avoidance system. It consists of 173 lines of code. The authors have created 41 versions of the program by injecting one or more faults. Their goal was to introduce faults that were as realistic as possible, based on their experience with real programs. We refer to the versions as "v1" to "v41". The Siemens test suite also contains 1600 test cases which are valid inputs for the program.

We created the golden outputs for these 1600 test cases by running the original version of the program. Then for each of the faulty versions, we ran those 1600 test vectors and matched with the golden outputs to segregate the failing test cases. Since the program does not contain a specification, we use the failing test cases as counterexamples and the correct value as its specification.

Table 1 shows the result of running BugAssist on the TCAS test suite. BugAssist ran 1440 times over all versions and 1367 of these runs pin-pointed the exact bug location, i.e., in 95% of the total runs. The "$TC\#$" in the table is the number of failed test cases

Version	TC#	Error#	Detect#	Size Reduc%	Run Time	Error Type	–	Version	TC#	Error#	Detect#	Size Reduc%	Run Time	Error Type
v1	132	1	132	8.6	0.016	op		v21	16	1	16	8.6	0.108	op
v2	69	1	69	4.6	0.068	const		v22	11	1	11	5.7	0.056	code
v3	23	1	13	9.8	0.096	op		v23	42	1	41	6.3	0.100	code
v4	26	1	26	9.2	0.104	op		v24	7	1	7	8.6	0.092	op
v5	10	1	10	8.6	0.120	assign		v25	3	1	3	6.9	0.068	code
v6	12	1	12	8.6	0.108	op		v26	11	1	11	9.2	0.108	addcode
v7	36	1	36	9.2	0.072	const		v27	10	1	10	10.9	0.108	addcode
v8	1	1	1	8.6	0.112	const		v28	76	1	58	5.7	0.080	Branch
v9	9	1	9	5.2	0.092	op		v29	18	1	14	5.7	0.092	code
v10	14	2	14	9.2	0.136	op		v30	58	1	58	5.7	0.064	code
v11	14	2	14	6.3	0.080	op		v31	14	2	14	10.9	0.008	addcode
v12	70	1	48	9.2	0.164	op		v32	2	2	2	10.9	0.004	addcode
v13	4	1	4	9.2	0.080	const		v34	77	1	77	8.6	0.100	op
v14	50	1	50	8.1	0.028	const		v35	76	1	58	5.7	0.060	code
v15	10	3	10	7.5	0.104	const		v36	126	1	126	2.9	0.024	op
v16	70	1	70	9.2	0.104	init		v37	93	1	93	8.6	0.040	index
v17	35	1	35	9.2	0.096	init		v39	3	1	3	6.9	0.088	op
v18	29	1	29	6.9	0.124	init		v40	126	2	126	6.3	0.088	assign
v19	19	1	19	9.2	0.112	init		v41	20	1	20	8.6	0.120	assign
v20	18	1	18	9.2	0.120	op		–	–	–	–	–	–	

Table 1. Results of running BugAssist on the TCAS task of the Siemens Test Suite

Error Type	Explanation for the error
op	Wrong operator usage e.g.: $<=$ instead of $<$
code	Logical coding bug
assign	Wrong assignment expression
addcode	Error due to extra code fragments
const	Wrong constant value supplied e.g.: off-by-one error
init	Wrong value initialization of a variable
index	Use of wrong array index
branch	Error in branching due to negation of branching condition

Table 2. Type of error

for each version. We ran BugAssist with each of these failing test cases as failing program executions and the golden output as the assertion to be satisfied. The column "$Error\#$" shows the number of errors injected in to each version. Most versions have only 1 error but some have 2 and 3 errors. "$Detect\#$" is the number of runs of BugAssist which detected the correct (human-verified) bug location. "$SizeReduc\%$" is the percentage reduction in the code size given by the tool to locate the bug, the ratio of bug locations returned by the tool to the total number of lines in the code. The "$RunTime$" shows the run time for each run of BugAssist in seconds and they are negligible. The last column is the type of bug which is explained in Table 2. For example, the version v2 has one error injected and has 69 failing test cases. We collected the bug locations reported during these 69 runs of the tool which gave 8 potential bug locations, which is 4.6% of the total line number's in the program. The exact location of the fault is contained in the localizations obtained from all the 69 runs.

Except for a few versions like v12, v28 and v35, BugAssist detected the correct bug location for all the runs. For the remaining ones, when we rank locations based on frequency of being reported as bugs, exact bug locations had a count more than half of the total number of runs. The runs in which exact location was not reported did give clues about the real bug. For example, some test cases had wrong constant value assignment to an array element, for which the tool reported the fault at places where that array is accessed rather than the line at which the bad assignment occurred. By analyzing the error locations it is quite evident that the error is due to a wrong value in that array location. On average the number of lines to check for potential bug is reduced to 8% of the total code. It should be noted that most of the single runs of the faulty version have captured the exact bug location.

Figure 1 gives an overview of a version of tcas (v2). The bug is in line 2 (and the original code is shown commented out in line 3). The bug is injected in function *Inhibit_Biased_Climb* at line 2 by changing the constant value. The declaration and initialization of variables, functions, and conditional statements that are not relevant to this bug are omitted in the figure. The program needs to satisfy the safety property *alt_sep_test()* should return *DOWNWARD_RA* and is given as assertion at line 37. There were 69 failing test cases for this version. We ran all these test cases and the tool returned 8 potential bug locations which are shown underlined in Figure 1.

There is no error reported in function *Non_Crossing_Climb*() because the call for that function at line 25 needs the function *Own_Below_Threat()* to be true, but that is false based on a comparison on the input parameters which are made hard clauses. Now lets take a closer look at the reported errors.

1. Line 34 is too weak for a fix because changing the return value can make the assertion always true and that does not serve as a suitable fix.
2. In line 26, setting the *need_downward_RA* variable to true can pick the right value for *alt_sep*. This decision is made by an evaluation of the two functions in that statement. The function *Own_Above_Treat()* returns true based on the input and it is clear that the correction needs to be done to the function call *Non_Crossing_Decend()*.
3. The function *Non_Crossing_Decend()* has a call for the actual faulty function at line 14. It also shows that the repair could be done by changing the return value of this function at line 19 (which we ignore as in case 1 above), or where the wrong evaluation happens (lines 15,16).

4. The actual bug at line 2 is reported as a potential bug location in all the runs.

6.2 Larger Examples

To show the performance and applicability of our approach for larger programs and in the presence of complex pointers and loops, we chose a set of other test cases with function calls, recursion, dynamic memory allocation, loops, and complex programming constructs.

The TCAS test cases were small enough to allow the MAX-SAT solver to deal with the Boolean trace formula without additional optimizations. However, for larger programs, the trace formula obtained by unrolling a program, and the corresponding MAX-SAT instance, was beyond the capacity of the MAX-SAT solver. Therefore, we combine our technique with existing trace reduction techniques like program slicing (S) [30], concolic execution (C) [11], and isolating failure-inducing input using delta debugging (D) [34].

Table 3 shows the result of running BugAssist on 4 other programs from the Siemens suite, each with one injected fault. "*Program*" shows the name of the program from the Siemens testsuite. "*LOC#*" is the total lines of code in the program and "*Proc#*", the number of procedure calls. The kind of reduction technique is specified in "*Reduc*" and "*assign#*" shows the size of the dynamic error trace as the number of assignment expressions before and after performing the reduction technique. The "*var#*" and "*clause#*" is the number of boolean variables and clauses in the MAX-SAT representation of the error trace both before and after the reduction step (the unit "m" denotes million). The number of potential fault locations returned by the tool is given under "*Fault#*". The column "*Time*" shows the runtime in seconds (s) or (in one case) hours (h).

We picked a faulty version of the program and one test input that reveals the bug. The golden output from the non-faulty program with the same input is given as a post condition on the return value of the faulty version. Trace reduction techniques are applied to the program execution with this input to generate a smaller trace formula and given as input to BugAssist. The tool reported the exact bug location in all programs except one (Program 2: `print_token`). Trace reduction techniques significantly reduced the resulting trace and the size of the MAX-SAT instance, as shown in "Before" and "After" sizes in Table 3. The cardinality of the potential fault location set for each of these programs turns out to be small. In all cases, the run time of the tool was smaller than our human effort required to isolate the fault on the original trace. This shows the applicability of the approach.

- The error inducing input to Program `totinfo` was the rows and columns of a matrix. The bug was in the constant value of a conditional operator on checking the product of rows and columns after a few other operations. A simple program slicing removed the assignments irrelevant to the assertion being checked and reduced the number of assignments to 21 and the run time to less than a second.
- Program `print_token` contained a recursive function "next_token" and the input to the program required the loops to be unrolled 8 times in the symbolic trace formula generation. This made the recursive function to have 64 instances in the symbolic trace and the number of assignments went up to 65K without concolic execution. Using concrete execution for the recursive function and variables, the number of assignment statements was bought down to 239. It should be noted that the limitation in using a concrete execution would be to assume that the bug is not present in the functions and loops which are concretized. However, this methodology fits well in programs using functions from a reliable library or for functions which are already verified to be bug free. This program did not show error at the exact location, which was a comparison on a variable which got the value from the concrete execution. This was because the constant propagation used by the symbolic trace generator abstracted away the variable since its value was a constant. Instead, the error was shown in the assignment of the variable to the constant.
- The priority scheduler program 3 and 4, contained a large error inducing input which called a number of procedures before deviating from the golden output of the original program. The trace size was significantly reduced after isolating the error inducing input using delta debugging, but was still quite big (about 400 and 5400 assignment operations respectively). In program 3, the off-by-one error on flushing the number of processes was detected by the presence of a single process creation (leading to a trace of about 400 assignments). However, program 4 required a much larger input and more procedures to expose the failure, resulting in a longer trace. It took BugAssist almost 11 hours to find the exact location (excluding the time taken for input minimization using delta debugging). Each execution of MAX-SAT took around 30 minutes to identify one potential fault location.

Program 2 The strncpy program with an off-by-one error

```
1   #define SIZE 15
2   void MyFunCopy (char *s)
3   {
4       char buf[SIZE];
5       memset(buf, 0, SIZE);
6       strncat(buf, s, SIZE);
7      /*Last argument should be: SIZE-1 */
8       return;
9   }

/*Standard C implementation of strncat*/
10   char *strncat(char *dest, const char *src,
                                    size_t n)
11   {
12         char *ret = dest;
13          while (*dest)
14             dest++;
15          while (n--)
16              if (!(*dest++ = *src++))
17                  return ret;
18          *dest = 0; /*Problem cause*/
19          return ret;
20    }
```

6.3 Fixing Off-By-One Errors

We demonstrate the repair capability of BugAssist by synthesizing fixes for off-by-one errors in the use of standard C library routines. We focus on off-by-one errors arising out of the misuse of the C `strncat` string manipulation function [21]. A common misconception with `strncat` is that the guaranteed null termination will not write beyond the maximum length. In reality, `strncat` can write a terminating null character one byte beyond the maximum length specified.

The Program 2 shows an instance of the bug in the function `MyFunCopy`, which takes a string `s` and uses the `strncat` routine to copy the contents to a string `buf` of length `SIZE`. The lines 10–20 shows a standard C implementation of `strncat`. Note that after

	Program	LOC#	Proc#	Reduc	assign#		var#		clause#		Fault#	time
					Before	After	Before	After	Before	After		
1	totinfo	565	7	S	734	21	0.797m	400	1.822m	1225	2	0.19s
2	print_tokens	726	18	C	65698	239	5.507m	7439	53.483m	22634	13	25s
3	schedule	564	21	DS	5914	391	5.173m	0.053m	15.379m	0.142m	13	28s
4	schedule	564	21	DS	41942	5412	78.982m	4.517m	239.385m	13.788m	25	11h
5	totinfo	565	7	CS	865	454	0.862m	0.734m	4.156m	3.728m	3	225s
6	schedule2	374	16	S	398	275	0.021m	0.015m	0.062m	0.048m	9	20s

Table 3. Running BugAssist on larger benchmark programs from the Siemens Test Suite

copying the n characters at line 17, the implementation writes to the $(n+1)$st location of the string `dest` on line 18. This implies that the function `MyFunCopy` should be using `SIZE` $-$ 1 as the last argument to `strncat`.

We ran BugAssist on this function, checking whether array accesses are within bounds. We made the assumption that library functions cannot be modified, and the error lies in the client code. That is, in the pMAX-SAT problem formulation, we made constraints arising out of library functions (`strncat` in this case) hard clauses. BugAssist located line 6 as a potential bug location in the code. This location is marked during preprocessing as a statement with a constant; so BugAssist now tries to fix it by changing the value to `SIZE` $-$ 1 and `SIZE` $+$ 1 as explained before. This creates two SAT instances with the new constant values, and we use a SAT solver to check if the error is still feasible. In this example, the change to `SIZE` $-$ 1 eliminated the bug.

Program 3 The nearest integer square root function with a bug at line 12

```
1    int squareroot()
2    {
3        int val = 50;
4        int i =1;
5        int v =0;
6        int res =0;
7        while(v < val)
8        {
9            v = v + 2*i +1;
10           i = i+1;
11       }
12       res = i;
13       /* res = i - 1; */
14       assert( (res*res <= val) &&
                    ((res+1)*(res+1) > val);
15         return res;
     }
```

6.4 Finding Faulty Loop Iterations

Program 3 contains a function to find the nearest integer square root of a value. The post condition is specified as an assertion, and states that the result `res` should be the closest square root for `val`. The bug locations reported by BugAssist are underlined. The correct code is given as a comment on line 13. Even though the actual bug is not in the loop body, it requires an analysis of the loop body to conclude that the right fix is at line 12. We gave the unwinding limit 50 to CBMC and BugAssist reports a potential repair at line 10 in the 8^{th} iteration of the loop. This gives clue to the programmer that the error occurs if the loop is iterated atleast 8 times.

7. Discussions

Program analysis based on Boolean satisfiability has been extremely successful in detecting subtle errors in large software programs [4, 6, 32]. We show that techniques based on Boolean MAX-SAT can be similarly effective in *localizing* program errors (as well as in identifying potential fixes).

Our fault localization algorithm depends on the underlying Boolean transform of the program to clauses, and is limited by the scalability of bounded model checking tools. In most cases, a single failing input was sufficient to locate the exact error location. Each of the potential error locations are the unsatisfied clauses in each iteration of the MAX-SAT solver. Our fault correction algorithm works by changing existing clauses. We cannot detect or correct code omission faults.

Our experimental results show that trace reduction techniques are crucial in making our implementation scale to reasonably large examples. Trace reduction techniques, such as delta-debugging or slicing, are orthogonal to our approach, and we can build on the extensive literature in these fields. Additionally, the performance can be improved by using an incremental SAT solver for iterative applications of MAX-SAT. While we have described error localization at the line-number (or program statement) level, our reduction to pMAX-SAT is general, and can be used at different levels of granularity. For example, to localize bugs at the function or module level, we can group clauses coming from the same function or module in the pMAX-SAT instance.

To improve the usability of our tool, we have built an Eclipse plugin to use BugAssist interactively during the development process. The plugin marks potential bugs in the code under development and assists in analyzing the right fix.

References

[1] Thomas Ball, Mayur Naik, and Sriram K. Rajamani. From symptom to cause: localizing errors in counterexample traces. In *POPL '03: Principles of Programming Languages*, pages 97–105, 2003. ACM.

[2] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using sat procedures instead of bdds. In *DAC '09: Design Automation Conference*, pages 317–320, 1999. ACM.

[3] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI '08: Operating Systems Design and Implementation*, pages 209–224, 2008. USENIX Association.

[4] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. Exe: automatically generating inputs of death. In *CCS '06: Computer and Communications security*, pages 322–335, 2006. ACM.

[5] Yibin Chen, Sean Safarpour, Andreas Veneris, and Joao Marques-Silva. Spatial and temporal design debug using partial maxsat. In *GLSVLSI '09: Great Lakes Symposium on VLSI*, pages 345–350, 2009. ACM.

[6] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In *TACAS '04: Tools and Algorithms*

for the Construction and Analysis of Systems, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176, 2004. Springer.

[7] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, 7:201–215, July 1960.

[8] Hyunsook Do, Sebastian Elbaum, and Gregg Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Softw. Engg.*, 10(4):405–435, 2005.

[9] Niklas Eén and Niklas Sörensson. Minisat v2.0 (beta). In *SAT-Race*, 2006. http://fmv.jku.at/sat-race-2006/.

[10] Zhaohui Fu and Sharad Malik. On solving the partial MAX-SAT problem. In *SAT '06: Theory and Applications of Satisfiability Testing*, volume 4121 of *Lecture Notes in Computer Science*, pages 252–265, 2006. Springer.

[11] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: directed automated random testing. In *PLDI '05: Programming Language Design and Implementation*, pages 213–223, 2005. ACM.

[12] Andreas Griesmayer, Stefan Staber, and Roderick Bloem. Automated fault localization for C programs. *ENTCS*, 174:95–111, 2007.

[13] Alex Groce, Sagar Chaki, Daniel Kroening, and Ofer Strichman. Error explanation with distance metrics. *Int. J. Softw. Tools Technol. Transf.*, 8:229–247, 2006.

[14] Monica Hutchins, Herb Foster, Tarak Goradia, and Thomas Ostrand. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *ICSE '94: International Conference on Software Engineering*, pages 191–200, 1994. IEEE Computer Society.

[15] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19:385–394, July 1976.

[16] Chu Min Li and Felip Manyà. MaxSAT, hard and soft constraints. In *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, chapter 19, pages 613–631. IOS Press, 2009.

[17] Chu Min Li, Felip Manyà, and Jordi Planes. New inference rules for MAX-SAT. *J. Artif. Int. Res.*, 30:321–359, October 2007.

[18] Mark H. Liffiton and Karem A. Sakallah. On finding all minimally unsatisfiable subformulas. In *SAT '05: Theory and Applications of Satisfiability Testing*, volume 3569 of *Lecture Notes in Computer Science*, pages 173–186, 2005. Springer.

[19] Joao Marques-Silva. Minimal unsatisfiability: Models, algorithms and applications (invited paper). In *ISMVL '10: International Symposium on Multiple-Valued Logic*, pages 9–14, 2010. IEEE Computer Society.

[20] Joao Marques-Silva and Jordi Planes. Algorithms for maximum satisfiability using unsatisfiable cores. In *DATE '08: Design, Automation and Test in Europe*, pages 408–413, 2008. ACM.

[21] Todd C. Miller and Theo de Raadt. strlcpy and strlcat: consistent, safe, string copy and concatenation. In *USENIX Annual Technical Conference*, pages 175–178, 1999. USENIX Association.

[22] Dawei Qi, Abhik Roychoudhury, Zhenkai Liang, and Kapil Vaswani. Darwin: an approach for debugging evolving programs. In *ESEC/FSE '09: European Software Engineering Conference and Foundations of Software Engineering*, pages 33–42, 2009. ACM.

[23] Manos Renieres and Steven P. Reiss. Fault localization with nearest neighbor queries. In *ASE '03: Automated Software Engineering*, pages 30 – 39, 2003. IEEE Computer Society.

[24] Sean Safarpour, Hratch Mangassarian, Andreas Veneris, Mark H. Liffiton, and Karem A. Sakallah. Improved design debugging using maximum satisfiability. In *FMCAD '07: Formal Methods in Computer-Aided Design*, pages 13–19, 2007. IEEE Computer Society.

[25] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: a concolic unit testing engine for C. In *ESEC/FSE '05: European Software Engineering Conference and Foundations of Software Engineering*, pages 263–272, 2005. ACM.

[26] Ilya Shlyakhter, Robert Seater, Daniel Jackson, Manu Sridharan, and Mana Taghdiri. Debugging overconstrained declarative models using unsatisfiable cores. *ASE '03: Automated Software Engineering*, pages 94–105, 2003. IEEE Computer Society.

[27] Armando Solar-Lezama, Rodric Rabbah, Rastislav Bodík, and Kemal Ebcioğlu. Programming by sketching for bit-streaming programs. *PLDI '05: Programming Languages Design and Implementation*, pages 281–294, 2005. ACM.

[28] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. Combinatorial sketching for finite programs. In *ASPLOS '06: Architectural Support for Programming Languages and Operating Systems*, pages 404–415, 2006. ACM.

[29] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. VS3: SMT solvers for program verification. In *CAV '09: Computer-Aided Verification*, volume 5643 of *Lecture Notes in Computer Science*, pages 702–708, 2009. Springer.

[30] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3:121–189, 1995.

[31] Wikipedia. Off-by-one error, the free encyclopedia, 2004. [Online; accessed 28-March-2010].

[32] Yichen Xie and Alex Aiken. Scalable error detection using boolean satisfiability. In *POPL '05: Principles of Programming Languages*, pages 351–363, 2005. ACM.

[33] Andreas Zeller. Isolating cause-effect chains from computer programs. In *FSE '10: Foundations of Software Engineering*, pages 1–10, 2002. ACM.

[34] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Trans. Softw. Eng.*, 28:183–200, 2002.