Making the Most of BMC Counterexamples

Alex Groce [1,2] Daniel Kroening [1,3]

Computer Science Department
Carnegie Mellon University
Pittsburgh, PA, USA

Abstract

The value of model checking counterexamples for debugging programs (and specifications) is widely recognized. Unfortunately, bounded model checkers often produce counterexamples that are difficult to understand due to the values chosen by a SAT solver. This paper presents two approaches to making better use of BMC counterexamples. The first contribution is a new notion of counterexample minimization that minimizes values with respect to the type system of the language being model checked, rather than at the level of SAT variables. Greedy and optimal approaches to the minimization problem are presented and compared. The second contribution extends a BMC-based error explanation approach to automatically hypothesize causes for the error in a counterexample. These hypotheses (in terms of relationships between variables) can be automatically checked to determine if a causal dependence exists. Experimental results show that causes can be automatically determined for errors in interesting ANSI C programs.

Key words: model checking, counterexamples, error explanation

1 Introduction

The value of counterexamples [9] in model checking [10] is indisputable: Bounded Model Checking (BMC) [4] can even be seen as a recognition of the centrality of the search for a counterexample to a property. For practical purposes,

[1] This research was sponsored by the Gigascale Systems Research Center (GSRC) under contract no. 9278-1-1010315, the National Science Foundation (NSF) under grant no. CCR-9803774, the Office of Naval Research (ONR), the Naval Research Laboratory (NRL) under contract no. N00014-01-1-0796, the Army Research Office (ARO) under contract no. DAAD19-01-1-0485, and the General Motors Collaborative Research Lab at CMU. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of GSRC, NSF, ONR, NRL, ARO, GM, or the U.S. government.

[2] Email: agroce@cs.cmu.edu

[3] Email: kroening@cs.cmu.edu

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

```
1  void f (int a, int b, int c)
2  {
3    int temp;
4    if (a > b) {
5      temp = a;
6      a = b;
7      b = temp;
8    }
9    if (b > c) {
10     temp = b;
11     b = c;
12     c = temp;
13   }
14   if (a < b) {
15     temp = a;
16     a = b;
17     b = temp;
18   }
19   assert ((a <= b) && (b <= c));
20 }
```

Fig. 1. sort.c

the diagnostic uses of counterexamples in understanding and correcting errors in a system or its specification are obvious.

A model checking counterexample is intended to be read by a person and used for debugging. Ideally, such a counterexample would be the most succinct and easily comprehensible witness to the existence of an error. The utility of small (in various senses, including length) counterexamples is widely recognized. Minimization of counterexamples, both in bounded [23] and explicit-state [14] model checking is a topic of ongoing research.

Previous work on minimization of counterexamples has concentrated either on producing counterexamples of minimal length or on removing irrelevant information from a counterexample. This paper presents a technique that can be used to minimize counterexample length, but focuses on a *semantic* minimization, with respect to the type system of the language (ANSI C, in this case). In particular, this approach minimizes the *values* of variables in the counterexample. As an example, consider the program in Figure 1. Without minimization, the Bounded Model Checker CBMC [19] produces the counterexample shown in Figure 2.

CBMC and similar tools are likely to produce counterexamples with unusually high (or low) values for variables. Bounded model checkers unwind a transition system to produce a propositional formula that is satisfiable if a counterexample of a certain length exists. The SAT solvers used to check these formulas for satisfiability typically return the first satisfying assignment produced. The counterexample values, therefore, are highly dependent on the decision heuristics used by the SAT solver. That these choices may result in needlessly large values for the actual program variables is independent of the issue of unnecessarily complete assignments addressed by other minimization work [23]. The issue is an artifact of the bit-level translation; however, using an integer-based technique such as UCLID [5] would not preserve the proper bit operation and overflow semantics of ANSI C.

```
Counterexample:
Initial State
----------------------------------------------------
State 1
----------------------------------------------------
  a=-8193 (11111111111111111101111111111111)
State 2
----------------------------------------------------
  b=-402 (11111111111111111111111001101110)
State 3
----------------------------------------------------
  c=-2080380800 (10000011111111111110100010000000)
State 4
----------------------------------------------------
  temp=0 (00000000000000000000000000000000)
State 10 file sort.c line 10 function c::f
----------------------------------------------------
  temp=-402 (11111111111111111111111001101110)
State 11 file sort.c line 11 function c::f
----------------------------------------------------
  b=-2080380800 (10000011111111111110100010000000)
State 12 file sort.c line 12 function c::f
----------------------------------------------------
  c=-402 (11111111111111111111111001101110)
Failed assertion: assertion file sort.c line 19 function c::f
```

Fig. 2. Counterexample for sort.c

```
Counterexample:
Initial State
----------------------------------------------------
  temp=-1 (11111111111111111111111111111111)
  a=0 (00000000000000000000000000000000)
  b=0 (00000000000000000000000000000000)
  c=-1 (11111111111111111111111111111111)
State 6 file sort.c line 10 function c::f
----------------------------------------------------
  temp=0 (00000000000000000000000000000000)
State 7 file sort.c line 11 function c::f
----------------------------------------------------
  b=-1 (11111111111111111111111111111111)
State 8 file sort.c line 12 function c::f
----------------------------------------------------
  c=0 (00000000000000000000000000000000)
Failed assertion: assertion file sort.c line 19 function c::f
```

Fig. 3. Minimized counterexample for sort.c

In this case, the decision heuristics used by ZChaff [22] assign 1 to a large number of bits. This results in large values for the program variables, making it difficult to follow what is happening. This problem, already evident in a small example program, is greatly exacerbated when many variable values are involved.

Using the optimization approach to value minimization presented in Section 4.2, we produce a new counterexample with minimal values for the program variables, making it much easier to follow the behavior of the program (Figure 3). Note that both counterexamples are of the same length and contain the same amount of program state.

This paper presents two approaches to value minimization. The first is a greedy approach that makes use of incremental SAT (Section 4.1), while

the second solves an optimization problem in order to guarantee true minimality (Section 4.2). Both approaches are used for counterexample length minimization, as well.

The second issue addressed in this work is that of making better use of counterexamples. Minimization directly improves the usability of counterexamples. *Error explanation* [15] provides information about the causality of errors beyond that contained in the counterexample alone. The `explain` tool [16] automatically generates explanations for CBMC counterexamples, based on the *counterfactual* theory of causality proposed by David Lewis [21]. Previous work [15] presented a notion of *causal dependence*, and noted that `explain` could check whether an error was causally dependent on a predicate. This feature was of limited utility, however, as the user was required to supply a possible cause to be checked. Section 6 presents a new method for automatically hypothesizing possible causes for an error.

2 Related Work

Minimization of counterexample length has been addressed in various contexts, including heuristic approaches [9,14,13]. Other kinds of minimization, based on game-semantics or minimal SAT assignments [18,23] have also appeared. The approach presented here for minimizing counterexample values can also be used to minimize counterexample lengths.

More generally, maximizing the utility of counterexamples has been addressed by the ideas of proof-like and evidence-based counterexamples [8,26].

The automatic causal dependence hypothesis-generation and checking presented in Section 6 is a natural extension of BMC-based *error explanation* [15,16]. Error explanation facilities have been described for MSR's SLAM model checker [3] and NASA's JPF model checker [17]. The game-based minimization approach of Jin, Ravi, and Somenzi [18] also provides an error explanation. The distance metric based approach used in Section 6 is related to Zeller's delta-debugging techniques [28,27] and the fault localization approach taken by Renieris and Reiss [24].

3 Bounded Model Checking for C Programs

CBMC reduces the model checking problem to determining the validity of a bit-vector equation; full details are presented elsewhere [11]. In a process analogous to that used for Bounded Model Checking of Kripke structures, the transition system is unwound by duplicating the loop bodies in the case of `for` and `while` loops, duplicating code in the case of loops build by means of backward `goto` statements, and function inlining in the case of recursive functions. Unwinding assertions ensure that sufficient unwinding has been performed – i.e. that it is not possible that a counterexample can be found by allowing more loop iterations.

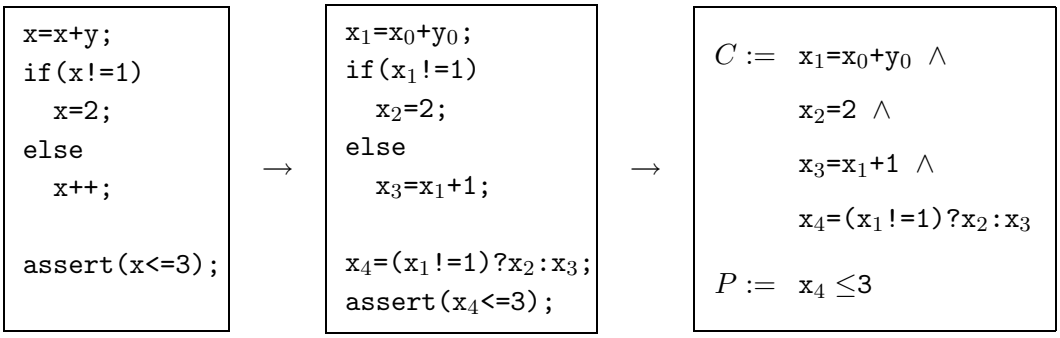

Fig. 4. Transformation into SSA

The program, after unwinding, is composed of only `if` statements and assignments. This program is then transformed into static single assignment (SSA) form [2], which requires a pointer analysis. We omit the full details of this process. For programs in SSA form, each variable is assigned at most once (Figure 4). The SSA form is then transformed into an equation C by replacing the assignments by equalities. The property is denoted by P. In order to check the property, CBMC converts $C \wedge \neg P$ into CNF by adding intermediate variables and passes the CNF to a SAT solver such as Chaff [22]. If the equation is satisfiable, the solution to C represents a counterexample for P. If it is unsatisfiable, P holds.

The conversion of most operators into CNF is straight-forward and resembles the generation of appropriate arithmetic circuits. The tool can also output the bit-vector equation before it is flattened to CNF, for the benefit of circuit-level SAT solvers.

4 Counterexample Value Minimization

4.1 Greedy Minimization

The first method for value minimization is a greedy heuristic approach based on incremental SAT. The first step of the algorithm is to attempt to minimize the length of the counterexample. CBMC generates a Boolean guard variable for each basic block. The variable is 1 if and only if the block is executed in the trace. In hardware BMC, length minimization is not generally an issue if BMC is performed in an incremental fashion. The unwinding used by CBMC, however, includes program statements that may or may not be executed – the unwinding length is *not* the number of execution steps – it is the *potential* number of execution steps, *if* all guards are satisfied, which typically is not the case. Different counterexamples with the same unwinding depth may execute varying numbers of program statements.

The length minimization algorithm sorts the list of guard variables by the number of instructions each guard affects. Starting with the guard that affects the most instructions, the heuristic proceeds as follows: first, a clause is added to the clause data base with the negation of the guard variable as the only

literal (forcing the guard to be false). The algorithm then proceeds depending on the value of the variable in the current satisfying assignment:

- If the value of the variable in the current satisfying assignment is 0, the old satisfying assignment is also a satisfying assignment for the new set of clauses.
- If the value is 1, the SAT solver is restarted. If the new instance is also satisfiable, the new clause remains in the clause database. If it is unsatisfiable, the attempt failed and the new clause is removed.

The algorithm continues with the next guard from the sorted list until all guards have been used. The heuristic approach only then attempts to minimize the variables that are used in the counterexample trace. Because the guard values are now fixed, the only values minimized are those that will appear in the counterexample: assignments guarded by false conditions are not taken into account. Alternatively, one could attempt to minimize first values and then execution steps.

The heuristic begins with the most significant bits of all variables, and then continues towards the least significant bit. In case of unsigned variables, the heuristic attempts to make all the bits zero. Signed variables are encoded as two's complement, and the goal is to minimize the absolute value. Furthermore, positive values are preferred over negative values. Thus, in case of signed variables, the algorithm first tries to set the sign bit to 0. The following bits are then minimized to 0 or 1, depending on the outcome of the SAT instance for the corresponding sign bit. If the sign bit is 1, the heuristic attempts to make the following bits 1 as well, and vice versa.

4.2 Optimal Minimization

The greedy approach to minimization does not always work well. A very unfortunate choice for the initial value to minimize for the program in Figure 1 produces a counterexample (Figure 5) that is not only almost unminimized, but is *longer* than the original counterexample. [4]

True minimization of counterexample values can be considered as a 0-1 ILP problem. PBS [1] is a *pseudo-Boolean* constraint solver which, given a SAT instance in CNF and a set of integer coefficients for SAT variables, will solve optimization problems over the constraints. The length of the counterexample is minimized before values are minimized. Each guard bit is given a weight equal to the number of program statements guarded by that condition. The psuedo-Boolean optimization problem is to minimize the weighted sum, i.e., the number of executed program statements. As with the greedy algorithm, counterexample length minimization is completed and guard values are locked before value minimization begins.

[4] The length increase actually results from the attempt to greedily minimize counterexample length, rather than values.

```
Counterexample:
Initial State
----------------------------------------------------
State 1
----------------------------------------------------
  a=2114977792 (01111110000100000000000000000000)
State 2
----------------------------------------------------
  b=-33554433 (11111101111111111111111111111111)
State 3
----------------------------------------------------
  c=2138989455 (01111111011111100110001110001111)
State 4
----------------------------------------------------
  temp=0 (00000000000000000000000000000000)
State 6 file sort.c line 5 function c::f
----------------------------------------------------
  temp=2114977792 (01111110000100000000000000000000)
State 7 file sort.c line 6 function c::f
----------------------------------------------------
  a=-33554433 (11111101111111111111111111111111)
State 8 file sort.c line 7 function c::f
----------------------------------------------------
  b=2114977792 (01111110000100000000000000000000)
State 14 file sort.c line 15 function c::f
----------------------------------------------------
  temp=-33554433 (11111101111111111111111111111111)
State 15 file sort.c line 16 function c::f
----------------------------------------------------
  a=2114977792 (01111110000100000000000000000000)
State 16 file sort.c line 17 function c::f
----------------------------------------------------
  b=-33554433 (11111101111111111111111111111111)
Failed assertion: assertion file sort.c line 19 function c::f
```

Fig. 5. Greedily minimized counterexample for sort.c

The notion of value minimality used here is to minimize the sum of the absolute values of all variables (appearing in the counterexample), with respect to the type system of the language. Again, consider the program in Figure 1. At first glance, it would appear that the goal is to minimize the sum: `|a| + |b| + |c| + |temp|`. However, each of these variables may take on different values during execution of the program. Therefore, the sum that is minimized is over all program variables after loop unrolling and static single assignment [2], in this case `|a#0| + |a#1| + |a#2| + |a#3| + |a#4| + |b#0| + ... + |b#6| + |c#0| + |c#1| + |c#2| + |temp#0| + ... + |temp#6|`.

For unsigned bit-vectors, the pseudo-Boolean constraints produced simply use values proportional to the place values, i.e., the least significant bit receives a weight of 1, the next least significant receives a weight of 2, up to a weight of 2^{n-1} for the most significant bit of an n-bit vector. Let a_0 denote the least significant bit of the bit-vector a, and a_{n-1} the most significant bit. We denote the integer value of an unsigned bit-vector a by $\langle a \rangle$:

$$(1) \quad \langle a \rangle := \sum_{i=0}^{n-1} a_i \cdot 2^i$$

For signed bit-vectors, such as those in `sort.c`, a different approach is required. CBMC encodes signed bit-vectors using two's complement. Note

that the ANSI C standard also permits other encodings.

The integer value represented by a is in the range from -2^{n-1} to $2^{n-1}-1$ and is denoted by $[a]$:

$$[a] := -2^{n-1} \cdot a_{n-1} + \langle a_{n-2} \ldots a_0 \rangle \quad (2)$$

The bit a_{n-1} is called the sign bit. We denote it by $sign(a)$.

We aim at minimizing the absolute value of a. If a is positive, i.e., if a_{n-1} is 0, then the absolute value of a is equal to a. If the sign bit is 1, the negation of a is

$$-[a] = \langle \overline{a} \rangle + 1 \quad (3)$$

where $\overline{a}$ denotes the bit-wise negation of a.

We implement this computation as follows: For each variable x of a signed bit-vector type, we introduce a new variable x' which is the bit-wise xor of x with its own sign bit:

$$x'_i := x_i \oplus sign(x) \quad (4)$$

If x is positive, then $x' = x$ and it is obvious that x' it is equal to the absolute value of x, i.e.,

$$|[x]| = \langle x' \rangle$$

If x is negative, then $x' = \overline{x}$, and obviously

$$|[x]| = -[x] = \langle x' \rangle + 1$$

Combining both cases results in

$$|[x]| = \langle x' \rangle + x_{n-1}$$

As x'_{n-1} is always zero, we get

$$|[x]| = \langle x'_{n-2} \ldots x'_0 \rangle + x_{n-1}$$

The pseudo-Boolean constraints are then assigned almost exactly as in the unsigned case, with the following exceptions: (i) the constraints for every place value, *not including the sign bit* are based on x' rather than x and (ii) the weight of the sign bit is 1, rather than 2^{n-1}. Thus, we minimize

$$x_{n-1} + \sum_{i=0}^{n-2} 2^i \cdot x'_i \quad (5)$$

5 Experimental Results

Table 1 shows results for minimization of counterexamples for several programs. The first column shows which program is being model checked. The remaining columns give results for the non-minimized counterexample, the greedily minimized counterexample, and the optimally minimized counterexample, in groups of three. In each group, the first column gives the time taken to generate a counterexample (time(s)). The second column ($\Sigma[x]$) in each group gives the sum of the absolute values of the variables and the third

	normal			greedy			optimal		
Program	time(s)	$\Sigma[x]$	l	time(s)	$\Sigma[x]$	l	time(s)	$\Sigma[x]$	l
sort.c	0.70	4.161×10^9	7	1.11	1.073×10^{10}	10	995.72	2	7
TCAS #1	1.30	111,905	73	5.22	111,905	73	13.35	9,734	73
TCAS #11	1.20	747,623	65	4.44	747,623	65	14.55	9,524	65
TCAS #31	1.64	488,241	68	4.05	488,241	68	12.23	8,932	68
TCAS #40	0.87	640,307	63	3.83	640,307	63	5.32	9,526	63
TCAS #41	1.69	937,749	72	4.00	937,749	72	6.05	9,528	72
adpcm_coder	4.42	814	106	39.52	73	91	107.30	391	91
adpcm_decoder	2.47	578	83	41.20	517	78	9.49	574	73
epic_quantize	1.06	18	28	7.58	14	28	3.65	14	28
g721_decode	8.10	1.075×10^9	289	168.63	855,224	298	18.45	855,106	289
gsm_decode	367.60	5257	250	3667.68	3,166	374	2436.08	180,041	225
mpeg2dec	3.82	6.334×10^9	61	141.41	9	60	55.36	9	60

Table 1
Time and minimization results for greedy and optimal strategies.

column (l) gives the length in steps of the counterexample. Sums over 1 billion are given in scientific notation. The benchmarks are taken from the example program `sort.c`, the TCAS suite [25], and the MediaBench benchmarks [20].

For the sort example and TCAS benchmarks, greedy optimization resulted in no improvements in the original counterexamples but in all cases took less time than true optimization.

For the MediaBench benchmarks, the results are mixed. The greedy heuristic is typically slower than the true optimization, but results in smaller values in some cases (the values are a secondary goal, and larger values in the optimal algorithm can be caused by different control flow traces computed in the first stage). On two benchmarks, hardly any minimization is achieved by either algorithm. These benchmarks make heavy use of large lookup-arrays, which are computed at run-time.

6 Hypothesizing and Checking Causal Dependence

Previous work [15] using CBMC to explain errors in programs presented a notion of *causal dependence* derived from David Lewis' counterfactual theory of causality [21]:

Definition 6.1 [causal dependence] A predicate e is *causally dependent* on a predicate c in an execution a iff:

(i) c and e are both true for a (we abbreviate this as $c(a) \wedge e(a)$)

(ii) $\exists$ an execution b . $\neg c(b) \wedge \neg e(b) \wedge (\forall b' \,.\, (\neg c(b') \wedge e(b')) \Rightarrow (d(a,b) < d(a,b')))$

where d is a *distance metric* for program executions. In other words, e is causally dependent on c in an execution a iff executions in which the removal

of the cause also removes the effect are more like a than executions in which the effect is present without the cause.

The previous work did not focus on checking causal dependence, as determining if e depends on c is only useful *after* arriving at likely candidate causes. This would be putting the cart before the horse, as the chief goal of error explanation is to help the user move from awareness of the existence of an error to a small set of candidate causes. The distance metric that allows causal dependence checking was instead used to discover a successful execution that was as similar as possible to a given counterexample. The distance metric used by CBMC is based on the total number of atomic changes (Δs) in variable and guard values between two executions [15]. These differences are presented to the user as causes for the error.

However, differences in actual variable values are often *too specific*. The relevant information is often a change in *relationships* between variables: i.e., not that `x` was 100 and must be changed to 200 to avoid violating an assertion, but that in the failing run `x < y` and in the successful run, `x > y`. The basic explanation approach may, unfortunately, completely omit `y` from an explanation if only the value of `x` is altered in the successful execution. Because the distance metric minimizes the number of changes, such omissions are very likely to occur. A more general notion of Δs would report to the user all predicates whose values are different for the counterexample and the successful execution. As the set of changed predicates is potentially infinite (comparisons of variables with constant values, etc.), only a subset of the potential Δs can be considered. Our implementation only checks basic ordering and equality relations between program variables, e.g. `x == y`, `x < y`, `x > y`, `x <= y`, etc.

Directly presenting the set of changed Δs is not particularly useful: changes in important variables are likely to introduce many accidental and unimportant changes, hiding the relevant differences in a large set of uninteresting results. However, the set of changes can be used a set of *candidate causes* for *checking causal dependence*. Only the Δs on which the error is causally dependent are presented to the user.

The set of predicate Δs that need to be checked is minimized by requiring that one of the variables being compared has changed its value in the successful execution. If neither variable has changed value, the predicate value must be unchanged. Given a possible cause c, the counterexample execution a, and an error (or effect) e, checking causal dependence requires two steps:

(i) Find an execution b such that (1) c does not hold and (2) the distance $d(a, b)$ is minimal. b is an execution that is as similar as possible to the counterexample a, except that the potential cause c is present in a but not in b. If the error e is present in b, it is not causally dependent on c.

(ii) Perform bounded model checking over all executions such that (1) c does not hold and (2) the distance to a is equal to $d(a, b)$. If all such executions are error free (e does not hold), then e is causally dependent on c.

```
Error is causally dependent on these predicates:
  c#0 < a#0
  c#0 < b#0
```

Fig. 6. Causes for sort.c

```
Error is causally dependent on these predicates:
  Input_Down_Separation#0 == Layer_Positive_RA_Alt_Thresh#1
  Input_Down_Separation#0 <= Layer_Positive_RA_Alt_Thresh#1
  Down_Separation#1 == Layer_Positive_RA_Alt_Thresh#1
  Down_Separation#1 <= Layer_Positive_RA_Alt_Thresh#1
```

Fig. 7. Causes for TCAS error #1

```
 100c100
// (correct version)
<       result = !(Own_Below_Threat()) || ((Own_Below_Threat()) &&
          (!(Down_Separation >= ALIM())));
---
// (faulty version #1)
>       result = !(Own_Below_Threat()) || ((Own_Below_Threat()) &&
          (!(Down_Separation > ALIM())));
```

Fig. 8. diff of correct TCAS code and variation #1

Figure 6 shows a subset of the causes discovered for the counterexample shown in Figure 3. In this case, the only causes shown are those which relate two input values. The algorithm actually detects 63 additional causes, relating inputs to intermediate values, or intermediate values to each other. For this reason, an option is provided to only check for relationships between input variables. The high degree of causal dependence in this case derives from the nature of the code: for a faulty sorting routine, ordering relations will obviously be crucial to the occurrence of the error, unless the sorting routine is invariably incorrect. The relationships between intermediate values are somewhat uninteresting in this case, as the set of input values is equivalent to the set of all values computed by the program.

For variation # 1 of the TCAS case study [25,12] examined in earlier work [15,16], however, a much smaller set of causes (Figure 7) is produced without restriction to input values. Figure 8 shows the error in the TCAS code as a diff between correct and incorrect versions. The automatically generated explanation, as described in the earlier work, focuses attention on line 100. The function call to `ALIM()` on this line always returns a value that is equal to `Layer_Positive_RA_Alt_Thresh#1`. Any user familiar with the specification of the TCAS code will be aware of this equivalence. Knowing (i) that the fault can be localized to line 100 and (ii) that the error is causally dependent on the predicate `Down_Separation#1 == Layer_Positive_RA_Alt_Thresh#1`, a user should be able to quickly conclude that the `>` comparison on line 100 should be a `>=` comparison.

6.1 Alternative Approaches for Hypothesis Selection

The particular choice of predicates for which to check causal dependence involves a number of tradeoffs: using too many predicates will increase com-

putation time and may result in redundant results; using too few may miss causal dependencies. An obvious alternative method is to use predicates taken from guards and Boolean assignments in the program source. Such comparisons should be generalized: if `x > y` appears in a guard, checking `x <= y`, `x == y`, and so forth is necessary to catch cases where the choice of comparison operations is incorrect. The primary difference between this generalization and the method implemented in CBMC is that no causality checking is done for (1) comparisons with constants and (2) comparisons with temporary results that are never stored in a variable (i.e. `x > (y + 50)`) are not checked for causality. On the other hand, comparisons between values that do not appear in guards together are checked. Causal dependencies that are directly present in a guard in the source code are generally not as difficult to detect as indirect dependencies: a change in guard value is likely to appear in the explanation. For this reason, it seems at least reasonable to expect that the current tradeoff is often the correct choice. More extensive evaluation will be needed to determine if a source code-mining approach is preferable.

Another alternative approach is to leverage predicate abstraction. The predicate abstraction based model checker MAGIC [6] now supports distance metric based explanation over *abstract* executions [7]. The predicates used in the abstract model could be tested for causal dependence. Checking causal dependence is less important in this case, however, as the explanations are presented in terms of changes in relationships between variables in the first place, and irrelevant Δs are suppressed by the metric and the abstract model.

7 Conclusions and Future Work

This paper presents a new kind of counterexample minimization: in contrast to previous work, the simplification is with respect to the semantic values of program variables. Small values are particularly beneficial for understanding traces of sequential programs, such as ANSI C programs. Conventional BMC implementations suffer from the fact that SAT solvers choose values according to built-in heuristics which do not favor readable counterexamples.

Two approaches are described: a greedy minimization heuristic using incremental SAT, and an algorithm that computes an exact solution using a pseudo-Boolean solver. The experimental results show that the optimal approach not only produces better results in a great many cases, but that it can be *faster* than the greedy approach. However, both algorithms are considerably slower than plain BMC without minimization.

More sophisticated heuristic approaches taken from the optimization community should outperform the naive greedy implementation. In future work, we plan to investigate SAT solvers with decision heuristics that are aware of a metric for counterexample simplicity: the idea being to make favoring simple counterexamples a part of the search algorithm, as opposed to a post-processing step.

The paper also presents a new use of BMC counterexamples, an extension of previous work on error explanation. This algorithm allows a model checker to identify predicates on which an error is causally dependent, in addition to providing a counterexample and fault localization. In future work, we hope to extend the range of predicates considered and consider subsumption or other methods for reducing the number of causes presented to the user.

References

[1] F. Aloul, A. Ramani, I. Markov, and K. Sakallah. PBS: A backtrack search pseudo Boolean solver. In *Symposium on the theory and applications of satisfiability testing (SAT)*, pages 346–353, 2002.

[2] B. Alpern, M. Wegman, and F. Zadeck. Detecting equality of variables in programs. In *Principles of Programming Languages*, pages 1–11, 1988.

[3] T. Ball, M. Naik, and S. Rajamani. From symptom to cause: Localizing errors in counterexample traces. In *Principles of Programming Languages*, pages 97–105, 2003.

[4] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 193–207, 1999.

[5] R. Bryant, S. Lahiri, and S. Seshia. Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In *Computer-Aided Verification*, pages 78–92, 2002.

[6] S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. *IEEE Transactions on Software Engineering*, 30(6):388–402, 2004.

[7] S. Chaki, A. Groce, and O. Strichman. Explaining abstract counterexamples. In *SIGSOFT/Foundations of Software Engineering*, 2004. To appear.

[8] M. Chechik and A. Gurfinkel. Proof-like counter-examples. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 160–175, 2003.

[9] E. Clarke, O. Grumberg, K. McMillan, and X. Zhao. Efficient generation of counterexamples and witnesses in symbolic model checking. In *Design Automation Conference*, pages 427–432, 1995.

[10] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.

[11] E. Clarke, D. Kroening, and K. Yorav. Behavioral consistency of C and Verilog programs using Bounded Model Checking. Technical Report CMU-CS-03-126, Carnegie Mellon University, School of Computer Science, 2003.

[12] A Coen-Porisini, G. Denaro, C. Ghezzi, and M. Pezze. Using symbolic execution for verifying safety-critical systems. In *European Software Engineering Conference/Foundations of Software Engineering*, pages 142–151, 2001.

[13] S. Edelkamp, A. L. Lafuente, and S. Leue. Trail-directed model checking. In *Workshop of Software Model Checking (SoftMC)*, 2001.

[14] P. Gastin, P. Moro, and M. Zeitoun. Minimization of counterexamples in spin. In *SPIN Workshop on Model Checking of Software*, pages 92–108, 2004.

[15] A. Groce. Error explanation with distance metrics. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 108–122, 2004.

[16] A. Groce, D. Kroening, and F. Lerda. Understanding counterexamples with `explain`. In *Computer-Aided Verification*, 2004. To appear.

[17] A. Groce and W. Visser. What went wrong: Explaining counterexamples. In *SPIN Workshop on Model Checking of Software*, pages 121–135, 2003.

[18] H. Jin, K. Ravi, and F. Somenzi. Fate and free will in error traces. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 445–458, 2002.

[19] D. Kroening, E. Clarke, and F. Lerda. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 168–176, 2004.

[20] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communicatons systems. In *International Symposium on Microarchitecture*, pages 330–335, 1997.

[21] D. Lewis. Causation. *Journal of Philosophy*, 70:556–567, 1973.

[22] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, pages 530–535, 2001.

[23] K. Ravi and F. Somenzi. Minimal assignments for bounded model checking. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 31–45, 2004.

[24] M. Renieris and S. Reiss. Fault localization with nearest neighbor queries. In *Automated Software Engineering*, 2003.

[25] G. Rothermel and M. J. Harrold. Empirical studies of a safe regression test selection technique. *Software Engineering*, 24(6):401–419, 1999.

[26] L. Tan and R. Cleaveland. Evidence-based model checking. In *Computer-Aided Verification*, pages 455–470, 2002.

[27] A. Zeller. Isolating cause-effect chains from computer programs. In *Foundations of Software Engineering*, pages 1–10, 2002.

[28] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, 2002.