Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 174 (2007) 95–111
www.elsevier.com/locate/entcs

Automated Fault Localization for C Programs[1]

Andreas Griesmayer, Stefan Staber, and Roderick Bloem

Graz University of Technology
{agriesma,sstaber,rbloem}@ist.tugraz.at

Abstract

If a program does not fulfill a given specification, a model checker delivers a counterexample, a run which demonstrates the wrong behavior. Even with a counterexample, locating the actual fault in the source code is often a difficult task for the verification engineer.
We present an automatic approach for fault localization in C programs. The method is based on model checking and reports only components that can be changed such that the difference between actual and intended behavior of the example is removed. To identify these components, we use the bounded model checker CBMC on an instrumented version of the program. We present experimental data that supports the applicability of our approach.

Keywords: fault localization, debugging, model checking, counterexample

1 Introduction

Debugging is one of the most time consuming parts of the software development cycle. Since the complexity of software systems is increasing, there is also an increasing demand in tools that aid the programmer in the debugging process.

Debugging consists of three steps: detecting that the program contains a fault, localizing the fault, and correcting the fault. Detecting faults has been an active area of research. There exists less work on fault localization and correction. In this work we focus on localization and present a novel method for determining the cause of faults in C programs.

We assume that we are given a program written in C and a specification. Suppose the program contains a fault, and we have a counterexample showing that the specification does not hold. We use the counterexample to create an extended version of the program. We fix the inputs of the program according to the values of the counterexample and introduce *abnormal predicates* for every component in

[1] This work was supported in part by the European Union under contract 507219 (PROSYD).

1571-0661 © 2007 Elsevier B.V. Open access under CC BY-NC-ND license.
doi:10.1016/j.entcs.2006.12.032

the program. If the abnormal predicate of a component is true, we assume that the component works abnormally. Therefore, we suspend the original behavior of the component and replace it by a new input. The localization problem is then one of finding which abnormal predicates need to be asserted and how the respective components have to be replaced in order to fulfill the original specification. In order to find abnormal predicates we negate the original specification to state that it is not possible to satisfy it by suspending components, and again use a model checker to verify the new system. If we find a failing run, we can derive from it the components that work abnormally and a replacement such that the program is fixed for the original counterexample.

1.1 Related Work

The presented work is closely related to the theory of model-based diagnosis (MBD) [21,16] which has its origins in diagnosing physical systems. Console et al. [4] show the applicability of MBD to fault localization in logic programs. Much work has been done in applying MBD to hardware designs on the source code level [9,25,20]. The theory has also been successfully applied to debugging programs written in a subset of Java [19,18]. In MBD, a model is derived from the source code. The model, together with a given failure trace, describes the actual, faulty behavior of the program. The correct output for the trace must be provided by an oracle (typically by the verification engineer). Model based diagnosis yields a set of components that explains the discrepancy between the model and the desired behavior but does not give a replacement for those components. In our approach we do not require a certain failure trace and an oracle that delivers the correct output values for that trace. Instead we assume that a specification is given and use a verification tool to obtain an arbitrary finite failure trace that proves that the specification is violated. We furthermore use the specification to automatically derive a correct behavior for that trace.

A related theory to model-based diagnosis was developed by Veneris, Smith, et al. See, for instance, [7]. They use a SAT solver to diagnose sequential circuits but do not consider specifications: the user must provide the proper behavior of the circuit for a given counterexample.

One application of diagnosis is design rectification. Huang and Cheng [13] develop a theory to locate sites at which a circuit can be modified to satisfy a new specification. In their case, the specification is given as another circuit.

In [15,23] a game based approach is presented for debugging. Using an LTL specification, the approach is able to locate a fault and provide a correction that is valid for all possible input values. Recently, this work was extended to pushdown systems to handle recursive Boolean programs. Details on the approach and its application to abstractions from C programs, are given in [10]. The problem of finding a correction that is valid for all inputs is computationally hard and therefore the approach is less efficient than the one presented in this paper, where we only provide a correction for the considered counterexamples.

Other existing work uses the difference between traces for fault localization.

Renieris and Reiss [22] assume the existence of a faulty trace and a number of successful traces. The difference between the traces points to suspicious parts of the program. Experimental data show that the quality of the fault localization strongly depends on the given set of traces. In the work of Groce [11] a specification is used to generate a successful trace that is as close as possible to a counterexample. The difference between failing and successful traces serves as a basis to explain the fault. Experimental data shows quite good results of their approach although for some examples manual assumptions have to be added in order to obtain insightful explanations. Comparison to our work can be found in Section 4. In [26], Zeller's delta debugging approach is used to obtain a cause-effect chain of a fault. The differences between states in heuristically chosen points of interest in a failing and a successful program run deliver an explanation for the fault. Ball et al. [1] use multiple calls to a model checker and compare the obtained counterexamples to a successful trace. Transitions that do not appear in a correct trace are reported as a possible cause for the fault.

All approaches that use the difference between traces need a successful run that is either generated or given by the user. The reported statements are fragments of the code present in the failing run but not in a successful runs. It is not guaranteed that the real fault location is reported or that the fragment of the code is minimal. It is possible that different failing runs lead to different fault candidates without a shared candidate. In contrary to these approaches, we do not compare runs on different inputs, but start with a counterexample and look for a minimal change in the program in order to generate a successful run with the same inputs. Thus, every reported location is a potential fault candidate and can be used to fix the fault for the used counterexample. It is guaranteed that the real fault is in the set of found candidates (assuming that the fault is in the considered fault model, c.f. 2.1). Therefore, we rerun the process on multiple counterexamples and use the intersection of the results to narrow down the number of fault candidates.

Other existing work does not report possible fault locations, but rather gives a deeper understanding of the nature of a failure. In [14] segments of the counterexample are identified that force the error to occur. Zeller and Hildebrandt [27] point out the significant difference between a failure inducing program input and a successful input. Groce and Visser [12] generate and analyze similar versions of a program run (successful and faulty runs) in order to obtain more information about the cause of the failure.

In the next section, we give the details of our approach in the general case. Section 3 gives the instantiation for the case of localizing faults in C programs as well as details on the used tools and methods. We show the applicability of the approach in Section 4 by examining two examples and explaining the gathered results. We conclude and give some outlook for future work in Section 5.

2 Fault Localization by Model Checking

In this section we describe, in general terms, how a model checker can be used for fault localization. Efficiency is not our prime concern in this section. In section 3 we give a specific instantiation of the approach to localize faults in C programs.

2.1 Preliminaries

We have a system S which consists of a set of *components* $C = \{c_0, \ldots, c_{n-1}\}$ and a set of signals $V = V_S \cup V_{ND}$. Signal $v_i \in V_S$ is defined by the component c_i as function $f_i(V)$, signals in V_{ND} are chosen nondeterministically and describe inputs or nondeterministic behavior of the system. (In the following, we refer to both inputs and nondeterminism in the system as "inputs".) The exact definition of components and signals depends on the application domain. For hardware, for example, one can define the components to be gates and signals to be wires. (Cf. [15,23].) We assume that the system is sequential and operates in discrete time. That is, a run of the system is a countable sequence of valuations of the signals, including inputs and nondeterministic choices.

Besides the system, we are given a specification, which is a safety property φ. If the system does not fulfill φ, a model checker gives us a *counterexample* π: a finite sequence of assignments to signals in $V_C \subseteq V_{ND}$ such that the system violates the specification.

We view the violation of the specification as a contradiction between the behavior specified by φ and that prescribed by S. Components of the system can be changed according to a *fault model*, which describes the allowed changes in behavior. E.g. a possible fault model for circuits: a replacement of a gate by a NAND or a NOR gate. We call a component that can be changed such that the error is avoided (while the rest of the system stays unchanged) a *single fault diagnosis*. The set of all such components is called the diagnosis set for single faults. We naturally extend this notation to tuples of components for dual or multiple faults.

2.2 Approach

Our approach consists of the following steps.

(i) We choose a parameter m that defines how many components may be changed in order to generate a system which obeys the specification. E.g., if $m = 1$ we look for single faults.

(ii) We construct a new system S'. System S' is obtained from S by making two modifications.

First, we build $V'_{ND} = V_{ND} \setminus V_C$ and $V'_S = V_S \cup V_C$ by adding components to assign the values given by the counterexample to V_C. Note that the method how to do that depends on the system and the used model checker. While for hardware we usually have values for all nondeterministic signals and time steps, a model checker for software only gives values for a single trace through the program and gives no informations on statements that were not reached.

Second, we add to V'_{ND} new signals $ab_0, \ldots, ab_{n-1}$ with Boolean domain and for every $v_i \in V_S$ a signal nd_i with the same domain as v_i. We further replace the function f_i of c_i by $f'_i = \neg ab_i \wedge f_i \vee ab_i \wedge nd_i$. Thus, if $ab_i = \mathsf{true}$, we suspect component c_i to be incorrectly chosen and *suspend* the original behavior of the component. If $ab_i = \mathit{false}$ the component works as prescribed by the system.

(iii) We construct a new specification, $\varphi' = \neg\varphi \wedge (\sum_i ab_i \leq m)$. The negated specification φ' states that φ is violated in the modified system when a maximum of m components are suspended, i.e. there is no possibility to change maximal m components such that φ is fulfilled. (We identify true with the number 1.)

(iv) We ask the model checker to produce a counterexample π' that shows that S' does not satisfy φ'.

Suppose that a counterexample π' for φ' and S' exists. The relevant part of the counterexample is the valuation of the ab_i and nd_i. The set of components $\{c_i \mid ab_i = \mathsf{true}\}$ is called a *diagnosis*. The existence of counterexamples shows that S does not contradict φ when a component in the diagnosis is suspended. If we replace the c_i in the diagnosis with a component that behaves like nd_i, the resulting system satisfies φ under the inputs prescribed by π'. Note that inputs from π' adhere to those from π if present. Thus, our claim is that the diagnosis found by our approach only contains components that can be used to fix the system for the given counterexample. Obviously, there is no guarantee that all we can fix the system regardless of the inputs. Finding a diagnosis that is valid for all counterexamples is much harder [15,23].

If no counterexample π' exists, then the system cannot be fixed by replacing only m components. Typically, we start with $m = 1$, increasing m whenever diagnosis fails. To obtain all components, we can run the model checker several times, each time excluding the elements of the diagnosis already found. This is usually wasteful, as it forces many recomputations that can often be avoided by a small modification of the model checker.

2.3 Discussion

The theory presented here is quite general. It is not restricted to finite domains, as long as the modifications mentioned in point ii can be expressed. Likewise, the restriction that a signal is defined by exactly one component is unnecessary. If it is removed, general constraint systems can be diagnosed [8].

The requirement that the system be a sequential, discrete-time system is quite relaxed: real time programs fit this requirement as a time step can be taken to be the execution of a single statement. Dense time (analog) systems are not amenable to our approach. Some possible application domains are sequential circuits, Boolean programs as used in [2] (cf. [17]) and C programs, which are the topic of this paper.

Listing 1: Loop Unwind

```
1   index = 0;
    if (!(index<5)) goto end;
    printf("%d",index);
    index++;
    if (!(index<5)) goto end;
6   printf("%d",index);
    index++;
    if (!(index<5)) goto end;
    printf("%d",index);
    index++;
11  if (!(index<5)) goto end;
    assert(false);
    end:
```

3 Fault Localization for C Programs

In order to apply the approach to C programs, we first give a short description on the used model checker CBMC and bounded model checking for software. Subsection 3.2 presents our approach to finding faults in C programs, which is a variation on the general approach described in the last section.

3.1 CMBC

To implement fault localization for C, we use the software model-checker CBMC [3]. CBMC takes ANSI-C source code as input and handles pointers, arrays, and the C data-types without requiring abstraction. Specification is done using assert statements. CBMC adheres to the C semantics very precisely. Uninitialized variables, for example, introduce nondeterminism in the model, as do constructs for which the C standard is ambiguous.

Given a C program P and a bound k, CBMC constructs an unwinding P_k of the program in which every statement is executed at most once. Loops are unwound by creating k copies of the loop along with the proper if statements for the case that the loop requires fewer than k iterations. Backward goto statements and recursive calls are dealt with similarly.

Listing 1 gives the result of unwinding the following loop to a depth of 3:

```
for(index = 0; index<5; index++)
    printf("%d",index);
```

The assert in Line 12 is an *unwind assertion*, which is added by CBMC to decide whether or not the unwinding is deep enough. If it is violated, we found a run which traverses the loop more than three times. We have to call CBMC again with a higher bound for the unwinding.

In the unwound program, CBMC supports assume statements as well as assert statements. CBMC turns the unwound program into a propositional logic formula that is satisfiable if and only if there is an execution of the program that ends in the violation of an assert statement and satisfies all assume statements that it encounters. Satisfiability is checked by a standard SAT solver.

3.2 Fault Localization

Suppose we are given a counterexample that shows that an assertion is violated. The counterexample includes the values of the inputs and the nondeterministic constructs. Such a counterexample can be obtained from CBMC . As before, we will treat nondeterministic constructs and input values alike. We regard to the values chosen by the counterexample as $i_0, \ldots, i_{l-1}$.

Our choice of components is the set of expressions in the program, i.e., the right-hand sides of assignments, the conditions in *if*, *while*, and *case* statements, etc. Let us assume we have n such components: e_0 through e_{n-1}. Obviously, the choice of *signals* for our model are the variables.

The process of building the instrumented program includes unwinding to the depth k. We now construct P'_k according to the rules defined in Section 2.2 as follows:

The number of components that may be changed at a time determine how to instrument the code. In the following, we give the approach for single faults. The extension to multiple faults is straightforward and described in Section 3.4.

We change P **to** P' by declaring a new variable *diag* at the beginning of P', whose value is chosen nondeterministically in the range of $\{0, \ldots, n-1\}$. We further replace each expression e_i by the expression *(diag == i ? nondet() : e_i)* where *nondet*() returns a nondeterministic value. The value of *diag* determines which expression is suspended while at the same time all others behave as in the original program P.

The instrumented program is unwound by CBMC resulting in P'_k. Finally, we use the assignments in counterexample π to set the respective variables in P'_k

We negate the specification by ensuring that we report exactly those runs which do not violate any assertion in P'_k. Because an unwound program does not contain any loops, we can do so by adding a statement *assert(false)* to the end of P'_k and replacing every assert statement *assert(c)* in P_k by an assume statement *assume(c)* in P'_k.

The new counterexample again is computed using CBMC. If no new counterexample is found we can look for dual faults or increase the number of unwinds. Keeping the unwind assertion unchanged in the previous step helps to adjust the correct number of unwinds.

The expressions in the diagnosis are exactly those that can be changed to satisfy the counterexample for a otherwise unchanged program. Note that we do not fix the choices of the nondeterministic constructs in P_k that are not visited by the counterexample — by changing the condition of a branching statement, we still can find paths that contain nondeterminism. We thus report all expressions for which there is a successful execution of P_k for some choice of the nondeterministic components respecting the choices made in the counterexample. (A more precise method would check that the execution is successful regardless of the behavior of the remaining nondeterministic components, but this introduces mixed quantifiers

Listing 2: A simple C program $P = P_k$

```
    void main() {
      int a,b,c,d;
3     if(a){  /* nondeterministic choice */
        a = 5;
        b = 6;
        c = a + b;
        d = a * b;
8       if(a % 2 == 0){
          int e;
          a = e; /* nondeterministic assignment */
        }
        assert(c == 12 && d == 36);
13    }
    }
```

and is hard to compute.)

The full diagnosis for a given counterexample is built iteratively: when an expression is found, we add an assume statement to the program code to avoid that the same component is reported a second time. CBMC is run repeatedly until no more expressions are found. A more efficient way would be to add blocking clauses to the SAT instance.

3.3 Example

In the following we illustrate our approach using a simple C program. Listing 2 shows a program P that adds and multiplies the variables a and b. As specification we use an assert statement in Line 12. In Line 4, a is assigned 5, which is incorrect and should be 6. Note that there are no loops in the code and thus $P_k = P$.

Assume that the counterexample sets a, b, c, d to the values *1*, *vb*, *vc*, and *vd*, respectively. (Values *vb*, *vc*, and *vd* are irrelevant, a can be set to any nonzero value.) Listing 3 shows P'_k. In Line 3 the new variable *diag* is introduced. The counterexample is fixed in Line 6, but no value is fixed for e in Line 14. The statement assert(false) is added to the end of the program and each expression is replaced to include the appropriate check for the diag variable. The numbers used for the expressions correspond to the line numbers in Listing 2.

A run of a model checker on P'_k provides us with a counterexample with *diag* equals 4 and nondet() returns the value 6. It also enters the *then*-block of the *if* statement in Line 13 and assigns an nondeterministic value to e, and in the following also to a. Those last two assignments do not affect the specification and therefore are arbitrary. We see that the assignment to a in Line 6 is the correct value for all possible counterexamples. While in general wi have to find a expression which is valid for all possible counterexamples, in this example we can use this information directly to fix the original program: We replace the value 5 by 6 in Line 4 of Listing 2. There are no other diagnoses.

3.4 Dual fault Diagnosis

The extension of the approach for two (or more) faults is quite simple. We add two variables $diag_1$ and $diag_2$ instead of one; both are assigned to a nondeterministic

Listing 3: Modified unwound C program P'_k

```
1   void main() {
      int a,b,c,d;
      int diag;
      diag = nondet();

6     a = 1;  b = vb; c = vc; d = vd;

      if(diag == 3 ? nondet() : a){
        a = (diag == 4 ? nondet() : 5);
        b = (diag == 5 ? nondet() : 6);
11      c = (diag == 6 ? nondet() : (a + b));
        d = (diag == 7 ? nondet() : (a * b));
        if(diag == 8 ? nondet() : (a % 2))}
          int e;
          a = (diag == 10) ? nondet() : e;
16      }
        assume(c == 12 && d == 36);
      }
      assert(false);
    }
```

value independently. An expression is in the diagnosis if one of the *diag* variables is set to its number. The rest of the approach is analogous to the single fault diagnosis described above.

To prevent every expression from being detected twice, we include the requirement that $diag_1 < diag_2$. Furthermore, we first perform single fault diagnosis and exclude all tuples $\{diag_1, diag_2\}$ such that $diag_1$ or $diag_2$ is in the single-fault diagnosis.

3.5 Discussion

By using expressions as components, we get a fine grained and quite natural model for localization of the fault. Examining expressions allows us to localize faults in assignments, function calls and return statements as well as in conditions for if-statements or loops. Because we do not only examine if an expression contributes to a fault, but also if it is possible to change it to avoid the fault, our results are more exact than comparable methods. The corrected version of the program in Listing 2 assigns 6 to variable a in Line 7. Groce [11], Zeller [26], Ball et al. [1], and the dynamic slice [24] for this example comprise all assignments from Line 4 to Line 7 while we are able to correctly pinpoint the fault to Line 4.

To find a diagnosis, we have to express a possible replacement for a expression. While it is relatively easy to do so for expressions or left hand sides of assignments, structural errors such as missing statements of forgotten braces for if-statements of loops are very hard to localize while more coarse methods might give a rough area of the program to be examined by the verification engineer.

Our approach returns all expressions that can be changed such that we can generate a correct run from a given counterexample. Note that in case of a faulty value which is propagated through the program, it is not possible to tell which of the involved expressions is to change if the program is weakly specified. Again have a look at the example in Listing 2. If we change the assert in Line 12 to $assert(d == 36)$, it is not possible to tell if the fault is located in Line 4 or Line 7.

Table 1
Results of the TCAS task of the Siemens Test Suite.

	#TC	#diag	Time	Score		#TC	#diag	Time	Score
v1	131	15 (17-19)	2953	0.906	v21	16	15 (17-17)	585	0.906
v2	69	5 (11-18)	836	0.975	v22	11	8 (8- 8)	223	0.950
v3	23	7 (13-19)	423	0.956	v23	41	9 (9- 9)	885	0.944
v4	20	15 (18-19)	576	0.906	v24	7	15 (16-17)	254	0.906
v5	10	7 (17-18)	159	0.956	v25	3	8 (9- 9)	68	0.950
v6	12	13 (16-17)	253	0.919	v26	11	8 (16-19)	311	0.950
v7	36	4 (9-18)	743	0.975	v27	10	7 (17-18)	153	0.956
v8	1	19 (19-19)	26	0.886	v28	75	2 (10-19)	642	0.988
v9	9	9 (10-11)	114	0.944	v29	18	3 (9-18)	224	0.981
v10	14	12 (17-19)	269	0.925	v30	57	4 (10-17)	939	0.975
v11	14	5 (10-14)	162	0.969	v31	14	14 (15-16)	449	0.913
v12	70	7 (16-19)	1664	0.956	v32	2	14 (16-16)	39	0.925
v13	4	9 (17-19)	149		v33	89	4 (8-19)	892	0.369
v14	50	4 (4- 4)	594		v34	77	7 (16-18)	1906	0.956
v15	10	7 (17-17)	283	0.956	v35	75	2 (10-19)	1069	0.988
v16	70	15 (17-18)	1263	0.906	v36	123	2 (2- 3)	877	
v17	35	4 (9-18)	1300	0.975	v37	93	5 (9-19)	822	0.969
v18	29	4 (3-17)	499	0.975	v39	3	8 (9- 9)	66	0.950
v19	19	4 (9-17)	691	0.975	v40	123	9 (10-15)	3017	0.944
v20	18	15 (18-19)	748	0.906	v41	20	15 (18-19)	956	0.906

Thus, we see that proper specification greatly improves the results of the approach. In order to produce a reliable diagnosis, we consider all assert statements of a program.

4 Examples

We show the applicability of the approach by considering two examples: The following section shows the results from the TCAS task of the Siemens test suite [6]. The Siemens suite contains five tasks which are widely used in literature. Each task consists of a C program and several versions of it with faults introduced. The position and kind of the faults is known. Besides the source-files, a number of test cases along with the information which of the code versions pass them is provided.

In the second example, we check the implementation of a data structure. Section 4.2 gives the details of a Red-Black Tree, its properties and how to check them by assertions.

All programs were checked automatically. If not stated otherwise, all instrumentation was done by scripts. If we had multiple test cases for one program, the full state space was examined only for the first test case. On subsequent test cases, the search was restricted to the diagnosis calculated from previous test cases as motivated in Section 3.2.

4.1 TCAS

The TCAS task of the Siemens test suite constitutes an aircraft collision avoidance system. It consists of about 250 lines of code. To check the effectiveness of fault detecting tools, the authors of the suite created 41 versions of the program. Each was created manually by adding one or more faults, usually a change in a single line. In the following we will refer to the versions by "v1" to "v41". The suite also contains a set of 1600 test cases and their results on each of the TCAS-versions.

Because no specification is given, we use failing test cases as counterexamples and the correct value for the test case as specification. Except for v38, for which no test case is given, the test suite provides between 3 and 130 failing test runs for each program version, 1500 in total. The program contains 34 expressions in assignments and conditions, which were identified as potential cause of an error and instrumented as described above.

Table 1 gives the number of test cases for each version of TCAS. The third column presents the size of the final diagnosis as well as the variation of the size of the diagnoses if they are calculated for each test case separately. The time in the fourth column gives the overall calculation time for each TCAS version. Note that in most cases a subset of the test cases would result in the same final diagnosis. Thus, a proper choice of test cases to consider would significantly decrease the calculation time. As such a choice would require analysis of the test cases, and we do not concentrate on performance, all available test cases were considered for computation.

The last column gives the result from the scoring function proposed by [22]. It is based on program dependency graphs (PDG) and gives the distance between the elements reported in the diagnosis and the faulty statements as number in the range $0-1$. Higher numbers are better. For three of the of the versions it is not possible to calculate this score because the fault is located in a *#define* statement and thus is not part of the PDG. All of the scores are quite good except for v33. We get the low score in v33 because in that example, the fault is a wrong index in initializing array elements in four lines that were not reported by our single fault approach. Despite the low rank, the result contains the only statement accessing the array and thus seems quite useful.

In [11], Groce gives the scores for 5 versions of TCAS, for which his approach is applicable, and compares them with results from Renieris and Reiss [22]. Groce compares several methods with and without delta slicing and additional assumptions, we summarize the results in Figure 1. The vertical axis gives the result of the scoring function of Renieris [22], which is reciprocally proportional to the amount of code the user has to review when he starts at the given results. If a method reaches 100%, the fault is located most accurate. Note that while all approaches give good results for some of the examples, only the approach presented in this paper scores best or second for all of them.

We will now have a closer look at v2, its source code is given in Listing 4. Functions that are not important for the example and initialization of the variables are omitted. Constants are written in capital letters, global variables representing input values from the test case start with a capital letter, and local variables start with lower case letters. The failure was introduced in function *Inhibit_Biased_Climb*, Line 4 by confusing two constants, the original version is shown as annotation in the line above.

We demonstrate the fault with an error trace with 72 states reported by CBMC: It enters the body of the if-statement in Line 40. The call to *NCB_Climb* in Line 42 is irrelevant as the test case is such that *OBT()* returns false and thus the assignment

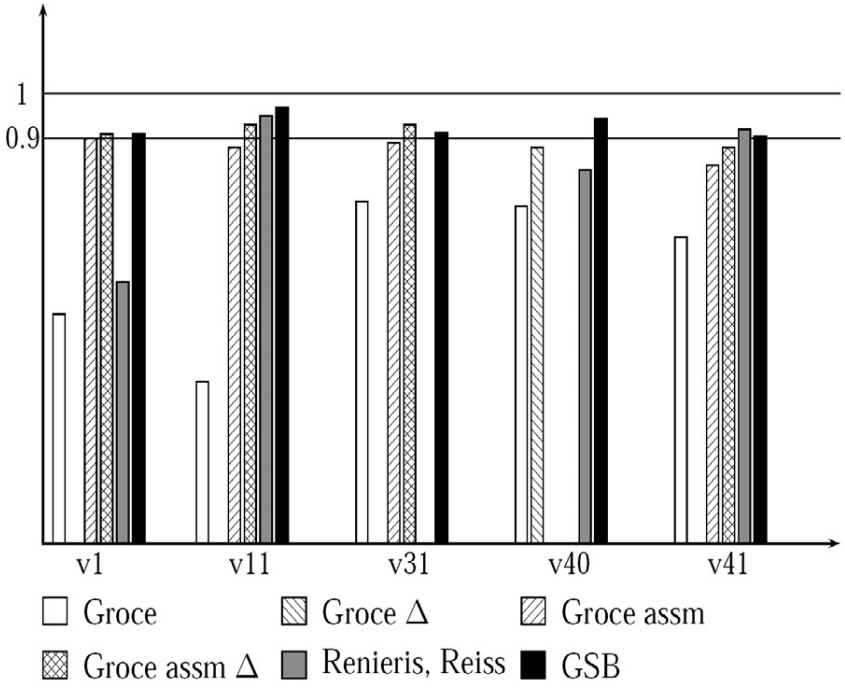

Fig. 1. Comparison of results from the TCAS examples with Groce and Renieris and Reiss

to *need_upward_RA* is false. In Line 43, *NCB_Descend* is called which subsequently calls the function containing the fault, *Inhibit_Biases_Climb*. The fault has the effect that *up_pref* is assigned *true* instead of *false* in Line 23 which results in a wrong assignment of *false* to *result* in Line 25 which is returned in Line 29. Thus, *need_downward_RA* in Line 43 is assigned *false* instead of *true*. This leads to the condition *false* in Line 48 and thus to setting the wrong return value in Line 51.

For this version of TCAS, we examined 69 test cases, each giving us a diagnoses with 10 to 18 elements. The final diagnosis found by our approach is the condition and left expression in Line 4 and Lines 8, 43, and 53 shown underlined. We examine if we can remove the error for the counterexample in each of the lines given as diagnosis:

- Line 53 is obvious, if we return *DOWNWARD_RA* in all cases, the assertion will not be violated anymore. In fact, a return statement of an erroneous function, or its call statement, can always be replaced by a call to a function which implements the correct behavior.
- If we set *need_downward_RA* to *true* in Line 43, we enter the correct else branch and return the correct value (*need_upward_RA* is *false* in the counterexample at hand).
- A repair in Line 8 is more complicated. A look at the counterexample showed that to remove the error, function *OBT* has to return *false* when called in Line 42, and *true* when called in Line 25. Because *OBT* compares two input variables, it should always return the same values within a run. A repair at this position

Listing 4: TCAS v2

```
1  int Inhibit_Biased_Climb ()
2  {
3    //return (Climb_Inh ? Up_Sep + NOZCROSS : Up_Sep);
4    return (Climb_Inh ? Up_Sep + MINSEP : Up_Sep);
5  }
6
7  bool OBT()
8  { return (Own_Tracked_Alt < Other_Tracked_Alt); }
9
10 bool NCB_Climb()
11 { /*declarations omitted*/
12   up_pref = Inhibit_Biased_Climb() > Down_Sep;
13   if (up_pref)
14     result = !(OBT()) || ((OBT()) && (!(Down_Sep >= ALIM())));
15   else
16     result = OAT() && (CV_Sep >= MINSEP)
17                    && (Up_Sep >= ALIM());
18   return result;
19 }
20
21 bool NCB_Descend()
22 { /*declarations omitted*/
23   up_pref = Inhibit_Biased_Climb() > Down_Sep;
24   if (up_pref)
25     result = OBT() && (CV_Sep >= MINSEP)
26                    && (Down_Sep >= ALIM());
27   else
28     result = !(OAT()) || ((OAT()) && (Up_Sep >= ALIM()));
29   return result;
30 }
31
32 int alt_sep_test ()
33 { /*declarations omitted*/
34   enabled = High_Conf && (OTA_Rate <= OLEV)
35                       && (CV_Sep > MAXALTDIFF);
36   tcas_eq = Other_Capability == TCAS_TA;
37   intent_not_known = Two_of_Three_Reports_Valid
38                        && Other_RAC == NO_INTENT;
39   alt_sep = UNRESOLVED;
40   if (enabled && ((tcas_eq && intent_not_known) || !tcas_eq))
41   {
42     need_upward_RA = NCB_Climb() && OBT();
43     need_downward_RA = NCB_Descend() && OAT();
44     if (need_upward_RA && need_downward_RA)
45       alt_sep = UNRESOLVED;
46     else if (need_upward_RA)
47       alt_sep = UPWARD_RA;
48     else if (need_downward_RA)
49       alt_sep = DOWNWARD_RA;
50     else
51       alt_sep = UNRESOLVED;
52   }
53   return alt_sep;
54 }
55
56 main(int argc, char*argv)
57 { initialize ();
58   /* initialization of global variables omitted*/
59   assert( alt_sep_test ()==DOWNWARD_RA);
60   }
```

seems unsuitable.

- The remaining elements of the diagnosis are the condition of the conditional statement in Line 4 and the expression where we introduced the fault in the same line. The condition is part of the diagnosis because the fault was introduced by wrongly using the larger constant *MINSET* instead of the smaller constant *NOZCROSS*. Thus, in the test cases leading to a wrong behavior, the return

Table 2
Results of the TCAS task for dual fault diagnosis.

	#TC	#diag	Time		#TC	#diag	Time
v6	12	4 (4- 5)	118	v25	3	10 (10-12)	82
v8	1	1 (1- 1)	6	v31	14	4 (4- 5)	177
v9	9	5 (5- 7)	129	v32	2	5 (5- 6)	20
v16	70	4 (4- 5)	889	v36	126	6 (6- 8)	1975
v22	11	6 (6- 8)	184	v39	3	10 (10-12)	74
v23	42	5 (5- 6)	588	v40	126	6 (6-15)	2722
v24	7	1 (1- 1)	44				

value of *Inhibit_Biased_Climb* was too high. By changing the condition to always select the second expression, which does not have this constant, we circumvent this behavior.

Although there is some work left for the verification engineer to decide which is the best position to repair the program, we see that the approach gives good results that only point to positions where we have the possibility to remove the failure.

In addition to the diagnosis, the approach gives us the value of *nondet()* that was chosen to avoid the wrong behavior for each of the identified expressions. This information is valuable to understand how to repair the program. In the example above, the condition of the conditional statement in Line 4 always selects the second, smaller, expression for return. To avoid the fault, we can also we replace the left expression by a negative value, indicating that the returned value should be smaller.

Other versions of TCAS give even better replacements for the faulty expression. E.g. in version 7, the fault was introduced by initializing a constant by 501 instead of 500. Examination of the suggested value for this line resulted in the correct assignment of 500 in order to remove the fault for all test cases.

4.1.1 Dual Faults

In addition to single fault diagnosis, we looked for diagnoses where exactly two expressions are replaced. The results are given in Table 2. Direct comparison with the results from Table 1 shows that the time needed for calculations of an example is quite comparable to the single fault case. Although a single run of CMBC takes about three times the time of single fault diagnosis, the overall runtime remains constant because the smaller size of the diagnosis rapidly leads to a smaller set of expressions to check (less calls of CBMC).

4.2 Red-Black Tree

A Red-Black tree is a self-balancing data-structure whose operations for insert, remove and search are performed in $O(n \log n)$ time [5]. It is a self-balancing binary search tree where each node has an color attribute. In addition to the requirements for binary search trees, following additional requirements have to be satisfied.

(i) A node is either *RED* or *BLACK*.

(ii) The root is *BLACK*.

(iii) All leaves are *BLACK*.

(iv) Both children of every *RED* node are *BLACK*.

(v) All paths from any given node to its leaf nodes contain the same number of *BLACK* nodes.

Insert-, remove- and search-operations are implemented as with usual binary search trees. After such an operation, the tree is examined to check if it is consistent with the rules above. If not, constant time rotation and repainting operations are performed to restore the consistency. This ensures that the tree remains balanced and guarantees the time bounds.

We took the implementation to check from the Wikipedia entry on Red-Black trees [2], and added the specification in form of assert statements as follows:

rule (ii): single assertion after a insert operation.

rule (iv) and binary search tree order: after insert operations, a routine is called which traverses the tree. If a red node is found, it checks that both children are black. The same routine also performs a comparison of the values to ensure the correct order of the binary search tree.

pointer access: Some of the functions implicitly assume that they are called with valid pointers. Assertions were added to ensure that pointers are not *NULL* when their fields are accessed even in the presence of faults in the code.

We don't have to check rule (i) as every node is initialized with color *RED* and only reassigned to *RED* or *BLACK*. Rule (iii) is always true because *NULL* pointers are treated as *BLACK* leafs. Rule (v) is not checked in this example.

Note that all of the assert statements are present in the source code at the same time, allowing only diagnoses which satisfy all of them. The example was tested by examining some test cases where nodes are inserted in an empty tree in the following order (the description gives the assumed operations on a full functioning data structure):

0,5 This simple case initializes the tree with a node with value 0 and adds a new node with value 5 as its right child. No repainting or rotation of the node is necessary.

5,1 Analogous to the first test case, but the second node is inserted as left child of the root node.

5,1,2 The last operation creates an unbalanced tree. To restore the properties of Red-Black trees, two rotations and a reassignment of the colors are necessary.

5,9,4,6 The last insertion creates a balanced tree, but one *RED* node has a *RED* child, which contradicts Rule iv. To restore the properties, a reassignment of the colors in two passes is necessary.

The source code implementing Red-Black trees consists of approximately 250 lines of code. Because the data-structure uses pointers to connect the nodes, diagnosis on even this small examples showed to be quite hard. Allowing CBMC

[2] http://en.wikipedia.org/wiki/Red-black_tree

to insert arbitrary values for assignments to the pointers of the data-structure can cause a complete reordering of the tree, inducing a large state space to be explored. Thus, in this example, performing diagnosis requires more effort then checking a program with a small amount of nondeterminism or none at all.

We introduced two faults to the sources:

(i) To satisfy rule (ii), the first node inserted to a tree is colored *BLACK*. This assignment was changed to *RED*.

(ii) Every node first is inserted to the tree like in a usual binary search tree. Therefore the tree is traversed to find the proper position for the new node. In the traversion routine, a comparison was changed from <= to == leading to a wrong order of the tree.

The first fault resulted in only one element giving the correct line of the fault for every test case. The second fault does not cause an incorrect tree for the first test cases. The second and third test case, however, lead to a diagnosis of three expressions containing the correct line. The state space for the fourth test case exceeded memory limits for the full diagnosis, but it was possible to check one by one if a expression from a previous test cases was also a in the diagnosis for the fourth test case. That way, we were able to rule out one expression and resulted in a final diagnosis with two elements.

5 Conclusions and Future Work

We presented a new approach to localize faults in C programs by constructing a modified system that allows a given number of expressions to be changed arbitrarily. The new system contains the inverted specification from the original program. If we can find an error trace for the new system, we found expressions to repair the original program.

We have demonstrated the applicability of the approach on two examples. The run time of the approach can still be improved. A significant portion of the time is taken by parsing, unwinding and generating the internal representation of the program. This representation does not change during calculation of the diagnosis for a single program. Overall run time would be significantly reduced by using blocking clauses and rerun the SAT solver to find the full diagnosis, instead of rerunning the complete CBMC process.

References

[1] T. Ball, M. Naik, and S. K. Rajamani. From symptom to cause: Localizing errors in counterexample traces. In *30th Symposium on Principles of Programming Languages (POPL 2003)*, pages 97–105, January 2003.

[2] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In M.B. Dwyer, editor, *8th International SPIN Workshop*, pages 103–122, Toronto, May 2001. Springer-Verlag. LNCS 2057.

[3] E. M. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'04)*, pages 168–176, 2004.

[4] L. Console, G. Friedrich, and D. Theseider Dupré. Model-based diagnosis meets error diagnosis in logic programs. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI'93)*, pages 1494–1499. Morgan-Kaufmann, 1993.

[5] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *An Introduction to Algorithms*. McGraw-Hill, New York, 1990.

[6] H. Do, S. G. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering: An International Journal*, 10:405–435, 2005.

[7] M. Fahim Ali, A. Veneris, S. Safarpur, R. Drechsler, A. Smith, and M. Abadir. Debugging sequential circuits using Boolean satisfiability. In *International Conference on Computer Aided Design*, pages 204–209, 2004.

[8] A. Felfernig, G. Friedrich, D. Jannach, and M. Stumptner. Consistency-based diagnosis of configuration knowledge bases. *Artificial Intelligence*, 152:213–234, 2004.

[9] G. Friedrich, M. Stumptner, and F. Wotawa. Model-based diagnosis of hardware designs. In *European Conference on Artificial Intelligence*, pages 491–495, 1996.

[10] A. Griesmayer, R. Bloem, and B. Cook. Repair of boolean programs with an application to C. In *18th Conference on Computer Aided Verification (CAV'06)*, pages 358–371, 2006. LNCS 4144.

[11] A. Groce. Error explanation with distance metrics. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'04)*, pages 108–122, Barcelona, Spain, March-April 2004. LNCS 2988.

[12] A. Groce and W. Visser. What went wrong: Explaining counterexamples. In *Model Checking of Software: 10th International SPIN Workshop*, pages 121–135. Springer-Verlag, May 2003. LNCS 2648.

[13] S.-Y. Huang and K.-T. Cheng. *Formal Equivalence Checking and Design Debugging*. Kluwer Academic Publishers, 1998.

[14] H. Jin, K. Ravi, and F. Somenzi. Fate and free will in error traces. *Software Tools for Technology Transfer*, 6(2):102–116, August 2004.

[15] B. Jobstmann, A. Griesmayer, and R. Bloem. Program repair as a game. In K. Etessami and S. K. Rajamani, editors, *17th Conference on Computer Aided Verification (CAV'05)*, pages 226–238. Springer-Verlag, 2005. LNCS 3576.

[16] J. de Kleer and B. C. Williams. Diagnosing multiple faults. *Artificial Intelligence*, 32:97–130, 1987.

[17] D. Köb. *Extended Modeling for Automated Fault Localization in Object-Oriented Software*. PhD thesis, Graz University of Technology, 2005.

[18] D. Köb and F. Wotawa. Fundamentals of debugging using a resolution calculus. In *Fundamental Approaches to Software Engineering (FASE'06)*, pages 278–292. Springer-Verlag, March 2006. LNCS 3922.

[19] W. Mayer and M. Stumptner. Extending diagnosis to debug programs with exceptions. In *18th IEEE International Conference on Automated Software Engineering, 2003*, pages 240–244, 2003.

[20] B. Peischl and F. Wotawa. Automated source-level error localization in hardware designs. *IEEE Design and Test of Computers*, 23(1):8–19, 2006.

[21] R. Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32:57–95, 1987.

[22] M. Renieris and S. P. Reiss. Fault localization with nearest neighbor queries. In *International Conference on Automated Software Engineering*, pages 30–39, Montreal, Canada, October 2003.

[23] S. Staber, B. Jobstmann, and R. Bloem. Finding and fixing faults. In D. Borrione and W. Paul, editors, *13th Conference on Correct Hardware Design and Verification Methods (CHARME '05)*, pages 35–49. Springer-Verlag, 2005. LNCS 3725.

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

[25] F. Wotawa. Debugging hardware designs using a value-based model. *Applied Intelligence*, 16:71–92, 2002.

[26] A. Zeller. Isolating cause-effect chains from computer programs. In *10th International Symposium on the Foundations of Software Engineering (FSE-10)*, pages 1–10, November 2002.

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