Automatic Error Localization for Software using Deductive Verification ⋆

Robert Könighofer, Ronald Toegl, and Roderick Bloem

IAIK, Graz University of Technology, Austria.

Abstract. Even competent programmers make mistakes. Automatic verification can detect errors, but leaves the frustrating task of finding the erroneous line of code to the user. This paper presents an automatic approach for identifying potential error locations in software. It is based on a deductive verification engine, which detects errors in functions annotated with pre- and post-conditions. Using an automatic theorem prover, our approach finds expressions in the code that can be modified such that the program satisfies its specification. Scalability is achieved by analyzing each function in isolation. We have implemented our approach in the widely used Frama-C framework and present first experimental results. This is an extended version of [8], featuring an additional appendix.

1 Introduction

Formal verification attempts to detect mismatches between a program and its specification automatically. However, the time-consuming work of locating and fixing detected bugs is usually performed manually. At the same time, the diagnostic information provided by the tools is often limited. While model checkers commonly provide counterexamples, deductive software verification engines usually only give yes/no (or worse: only yes/maybe) answers. Analyzing a proof or witness given by the underlying theorem prover is usually not a viable option.

In this work, we strive to lessen this usability defect in the context of deductive software verification [2]. This approach assumes that source code is annotated with pre- and post-conditions. It computes a set of *proof obligations*, i.e., formulas that need to be proven to attest correctness. These formulas are then discharged by an automatic theorem prover. Scalability is achieved by analyzing functions in isolation. We extend this verification flow such that the tool does not only report the existence of an error, but also pinpoints its location.

Our solution assumes that some code expression is faulty. This fault model is fine-grained and quite general. If verification of a function fails, we iterate over each expression in this function and analyze if it can be modified such that the function satisfies its contract for all inputs. If so, we report this expression as potential error location. Expressions that cannot be modified such that the error goes away do not have to be analyzed by the developer when trying to fix the

⋆ This work was supported by the European Commission through project STANCE (31775) and the Austrian Science Fund (FWF) through project RiSE (S11406-N23).

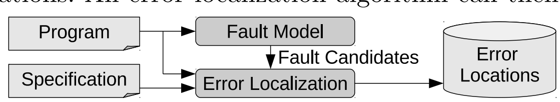
error. We have implemented a proof-of-concept in Frama-C [2], and provide first experimental results comparing our approach to FoREnSiC [1] and Bug-Assist [6].

Related work. Our fault model has been successfully applied before [7,1]: This approach also checks repairability of expressions, but only for fixed inputs. It uses assertions as specification, and SMT solvers as reasoning engines. [4] is similar but uses a model checker. In [6] a MAX-SAT engine is used. Our work resolves many drawbacks of these existing works: pre- and post-conditions are more powerful than assertions, we check repairability for *all* inputs, and we achieve scalability by analyzing functions in isolation. Model-based diagnosis [11] has already been applied in many settings (cf. [7]). Our approach is similar (we also check repairability), but focuses on single-fault diagnoses to avoid floods of diagnoses. Dynamic methods [5] rely on the quality of available test cases. In contrast, our method is purely formal. This is an extended version of [8], featuring an additional appendix with more detailed experimental results.

2 Automatic Error Localization

2.1 Fault Models

Intuitively, a fault model defines what can go wrong in a program, thereby inducing a set of candidate error locations. An error localization algorithm can then decide which of these candidates can actually be responsible for the detected problem. A good fault model needs to balance conflicting objectives: it should cover many errors, be fine-grained, allow for efficient error localization and not yield too many spurious error locations. Existing approaches include fault patterns [10] specifying common bugs, mutation-based fault models [3] assuming that the error is a small syntactic change, and faulty expressions [4,7] assuming that the control structure is correct but some code expression may be wrong. In this work we use faulty expressions because this fault model is fine-grained, more generic than mutation-based models, more automatic than fault patterns, and still allows for efficient error localization, as shown below.



2.2 Basic Idea for Error Localization

Our approach is inspired by [4,7]: An expression in the source code is a potential error location if it can be replaced such that the detected error is resolved.

Example 1. The program on the right is supposed to compute the maximum of `a` and `b`, but contains a bug in line 5. The post-condition `\result >= b` is incomplete but sufficient to detect the bug: it is violated if $b > a$. Our fault model (incorrect expressions) identifies 4 candidate error locations: Candidate C_1 is the expression "`a`" in line 3, C_2 is "`b > a`" in line 4, C_3 is the "`a`" in line 5,

```
1 /*@ensures \result >= b;@*/
2 int max(int a, int b) {
3   int r = a;
4   if(b > a)
5     r = a; //correct: r = b
6   return r; }
```

and C_4 is the "`r`" in line 6. Neither C_1 nor C_2 are error locations. C_1 cannot be changed to satisfy the post-condition because `r` is overwritten with the incorrect value "`a`" if $b > a$. If we change only C_2, `\result` will always be "`a`", which is incorrect if $b > a$. C_3 and C_4 are possible error locations, because these expressions can be replaced by "`b`" to make the program satisfy its specification. □

2.3 Realization with Deductive Verification

We now discuss how to answer such repairability questions automatically. From a high-level perspective, most formal verification tools compute a correctness condition $\mathsf{correct}(\bar{i})$ in some logic, where $\bar{i}$ is the vector of input variables of the program. Next, a solver checks if $\forall \bar{i} : \mathsf{correct}(\bar{i})$ holds. If not, an error has been detected. Deductive verification tools like the WP plug-in of Frama-C [2] follow this pattern by defining correct as implication: if the pre-condition of a function holds, then the function must satisfy its post-condition. Loops are handled with user-provided invariants, and a theorem prover checks $\forall \bar{i} : \mathsf{correct}(\bar{i})$. In practice, correct may be composed of parts that can be solved independently.

If a function is incorrect, we compute if a certain expression C is a potential error location as follows. First, we replace C by a placeholder c for a new expression. Next, we compute the correctness condition $\mathsf{correct}(\bar{i}, c)$, which depends now also on c. Finally, C is a potential error location if $\forall \bar{i} : \exists c : \mathsf{correct}(\bar{i}, c)$. This formula asks if expression C can, in principle, be replaced such that the function satisfies its contract. For every input $\bar{i}$, there must exist a value c to which the replacement of C evaluates such that the function behaves as specified. Note that this approach can, in principle, also compute a repair if the underlying theorem prover can produce a witness in form of a Skolem function for the c variable. However, this feature is not supported by our current implementation.

Example 2. We continue Example 1. We check if expression C_1 is a potential error location by replacing it with a placeholder c_1, as shown on the right. Next, we compute $\mathsf{correct}(a, b, c_1) = (b \leq a) \wedge (c_1 \geq b)$ using deductive verification. C_1 is not an error location because $\forall a, b : \exists c_1 : \mathsf{correct}(a, b, c_1)$ is false.

```
1 /*@ensures \result >= b;@*/
2 int max(int a, int b) {
3   int r = c1;
4   if(b > a)
5     r = a; //correct: r = b
6   return r; }
```

When replacing C_3 we get $\mathsf{correct}(a, b, c_3) = (b \leq a) \vee (c_3 \geq b)$. We have that $\forall a, b : \exists c_3 : (b \leq a) \vee (c_3 \geq b)$, so C_3 is a potential error location — as expected. □

2.4 Implementation in Frama-C

We implemented our error localization approach as a proof of concept in the WP plug-in of the widely used software verification framework Frama-C [2]. We discuss implementation challenges and reasons for imperfect diagnostic resolution.

Instrumentation. Frama-C normalizes the source code while parsing it into an Abstract Syntax Tree (AST). For instance, it decomposes complicated statements using auxiliary variables. Our instrumentation, replacing candidate expressions by a placeholder c, operates on this normalized AST. This makes it robust when handling complicated constructions. The disadvantage is that our approach may report error locations that are only present in the normalization. However, we do not consider this a severe usability issue, because the line number in the original code is available, and Frama-C presents the normalized source code and how it links to the original source code in its GUI.

Computation of correct($\bar{i}, c$). Internally, the WP plug-in of Frama-C performs simplifications that may rewrite or eliminate our newly introduced placeholder c, and thus, we cannot use WP a black-box to compute the correctness formula correct($\bar{i}, c$) after instrumentation. We solve this issue by extending Frama-C's memory model such that the placeholder c is not touched by simplifications.

Quantification. Once we have correct($\bar{i}, c$), we need to add the quantifier prefix $\forall \bar{i} : \exists c$. Unfortunately, correct may also contain auxiliary variables $\bar{t}$ that express values of variables at specific program points. Intuitively, c should not depend on variables that are assigned later in the program. This would violate the causality and lead to false-positives. Hence, we need to separate the variables of correct to construct the formula $\forall \bar{i} : \exists c : \forall \bar{t} :$ correct($\bar{i}, \bar{t}, c$). This is done by computing the input variables (parameters and globals) of the function under analysis and linking them to the corresponding variables in the formula.

Axiomatization. WP uses axiomatized functions and predicates in correct. For instance, for $a < b$ it writes $\mathrm{zlt}(a, b)$, where the predicate $\mathrm{zlt} : \mathbb{Z} \times \mathbb{Z} \to \mathbb{B}$ is axiomatized as $\forall x, y : (zlt(x, y) \to x < y) \wedge (\neg zlt(x, y) \to x \geq y)$. In our experiments we observed cases where the automatic theorem prover (AltErgo) could not decide formulas when using the axiomatization, but had no difficulty when the axiomatized predicates and functions are replaced by the corresponding native operators. Hence, we modified the interface to the theorem prover such that formulas do not contain axiomatized functions and predicates, where possible.

Diagnostic Resolution. Our implementation is neither guaranteed to be sound (it may produce spurious error locations) nor complete (it may miss potential error locations). The reasons are:

- The theorem prover may time-out or return "Unknown" if it could neither prove nor disprove the formula. We treat such verdicts as if the program was incorrect (a choice justified by experience), which results in incompleteness.
- Instead of one monolithic formula correct, WP may compute multiple formulas that are checked independently. In error localization, we also check each formula in isolation. This is weaker than checking the conjunction, i.e., can result in spurious error locations, but increases efficiency.
- Incomplete specifications can result in spurious error locations.
- The bug may not match our fault model. E.g., code may be missing or the control flow may be incorrect. This results in missed error locations.

3 First Experimental Results

Despite the potential imprecisions discussed in the last section, our implementation produces meaningful results. We evaluated our proof-of-concept implementation[1] on the widely used TCAS benchmark [12], which implements an aricraft traffic collision avoidance system in 180 lines of C code. It comes in 41 faulty versions that model realistic bugs. We annotated all functions with contracts.

3.1 Performance Evaluation

We compare the execution time and effectiveness of our approach with that of FoREnSiC [7,1] and Bug-Assist [6] on an ordinary laptop.[2] For our new approach, the error localization time (at most 129 [s], 37 [s] on average) is acceptable for all TCAS instances. For 37% of the cases, the execution time increases by only <40% when going from error detection to localization. FoREnSiC is slightly faster on average (17 [s]) but the median runtime is on par (16 vs. 18 [s]). With 7 [s] on average, Bug-Assist is even faster. Although only 66% of the benchmarks match our fault model, errors were successfully located in 90.2%. While FoREnSiC and Bug-Assist reported 15 error locations on average, our approach reported only 3.5. Thus, in our experiments, our tool provides much higher accuracy with only slightly longer runtime. The user has to examine only a few expressions in the code, which can speed-up debugging significantly.

3.2 Examples

This section investigates the reported error locations for a few TCAS versions.

Version 7. A constant is changed from 500 to 550 in an initialization function. Our tool reports exactly this constant `550` as the only possible error location. This takes 6 seconds, whereof 5.1 seconds are spent on error detection.

Version 9. This version contains the following function:

```
119 bool NonCrossBiasedDescend() {
120   bool r;
121   if (InhibitBiasedClimb() >= DwnSep) {
122     r = OwnBlTh() && VerSep >= MSEP && DwnSep >= ALIM();
123   } else {
124     r = !(OwnAbTh()) || (OwnAbTh() && UpSep >= ALIM());
125   }
126   return r; }
```

The correct program has a ">" instead of the ">=" in line 121. Our tool reports two potential error locations: `tmp_6 >= DwnSep` in line 121, and `tmp_1` in line 122. This output looks cryptic because the code has been normalized by Frama-C. `tmp_6` is an auxiliary variable that stands for `InhibitBiasedClimb()`. This is shown in the GUI. Hence, the first error location is just what we expect. `tmp_1` holds the value for `r` in line 122. This value can be changed to satisfy the specification for all inputs as well. Hence, it is also reported. `NonCrossBiasedDescend()` is not long, but contains complex logic. Analyzing this logic to locate a bug can be cumbersome. The diagnostic information provided by our approach helps.

Version 14 changes `MAXDIFF` (a preprocessor macro) from `600` to `600+50`. Our tool reports two possible error locations: `VerSep > 600+50` in line 167 and `OtherCap == 1` in line 168 of function `altSepTest`, which is shown below. The first one pinpoints exactly the problem. Note that `altSepTest()` is all but trivial.

[1] See `www.iaik.tugraz.at/content/research/design_verification/others/`.

[2] Table 1 in the Appendix gives more details to our performance results.

If verification fails, tracking down this bug can be a very time-consuming and frustrating task. By checking only the reported locations, we can significantly reduce the manual work to fix the bug. Thus, the reported error locations are usually both meaningful and helpful.

```
int altSepTest() {
  bool en, eq, intentNotKnown, needUpRA, needDwnRA;
  en = HConf && OwnTrAlt <= OLEV && VerSep > MAXDIFF;
  eq = OtherCap == TCAS_TA;
  intentNotKnown = TwoRepValid && OtherRAC == NO_INT;
  int altSep = UNRESOLVED;
  if (en && ((eq && intentNotKnown) || !eq)) {
    needUpRA = NonCrossBiasedClimb() && OwnBlTh();
    needDwnRA = NonCrossBiasedDescend() && OwnAbTh();
    if(needUpRA && needDwnRA) altSep = UNRESOLVED;
    else if (needUpRA)        altSep = UPWARD_RA;
    else if (needDwnRA)       altSep = DOWNWARD_RA;
    else                      altSep = UNRESOLVED;
  }
  return altSep; }
```

4 Conclusions

Tracking down a subtle program error in large source code is — like finding a needle in a haystack — a tedious task. We have extended a widely used deductive software verification engine so that it can report expressions that may be responsible for incorrectness. We evaluated our proof-of-concept implementation on a few examples and conclude that our approach is viable and gives fast and clear guidance to developers on the location of program defects.

Acknowledgment. We thank Loïc Correnson and the Frama-C team for their support with our proof-of-concept implementation.

References

1. R. Bloem, R. Drechsler, G. Fey, A. Finder, G. Hofferek, R. Könighofer, J. Raik, U. Repinski, and André Sülflow. FoREnSiC - An automatic debugging environment for C programs. In *HVC'12*. Springer, 2012.
2. P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-C - A software analysis perspective. In *SEFM'12*. Springer, 2012.
3. V. Debroy and W. E. Wong. Using mutation to automatically suggest fixes for faulty programs. In *ICST'10*. IEEE, 2010.
4. A. Griesmayer, S. Staber, and R. Bloem. Automated fault localization for C programs. *Electr. Notes Theor. Comput. Sci.*, 174(4):95–111, 2007.
5. J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *ASE'05*. ACM, 2005.
6. M. Jose and R. Majumdar. Cause clue clauses: error localization using maximum satisfiability. In *PLDI'11*, pages 437–446. ACM, 2011.
7. R. Könighofer and R. Bloem. Automated error localization and correction for imperative programs. In *FMCAD'11*. IEEE, 2011.
8. R. Könighofer, R. Toegl, and R. Bloem. Automatic error localization for software using deductive verification. In *HVC'14*. Springer, 2014. To appear.
9. J. R. Larus, T. Ball, M. Das, R. DeLine, M. Fähndrich, J. D. Pincus, S. K. Rajamani, and R. Venkatapathy. Righting software. *IEEE Softw.*, 21(3):92–100, 2004.
10. R. Reiter. A theory of diagnosis from first principles. *Art. Int.*, 32(1):57–95, 1987.
11. Siemens benchmark suite. `pleuma.cc.gatech.edu/aristotle/Tools/subjects`.

Appendix

Table 1 gives more details to our performance results on the TCAS benchmarks. Column 1 indicates if the error in this version of the benchmark matches our fault model. Even if this is not the case, our approach can often compute meaningful error locations. Column 2 lists the execution time for error detection. Column 3 gives the time for error localization including error detection. Column 4 gives the number of candidate expressions identified by our fault model. The number of potential error locations that have been reported by our implementation is listed in Column 5. The Columns 6 and 7 show the execution time for error localization (including error detection) and the number of reported (potential) error locations for the approach of [7], which is implemented in the tool FoREnSiC [1]. This approach can be run in two modes: the conservative mode may miss error locations, the non-conservative mode may find spurious locations. We used the non-conservative mode because otherwise no error locations are found for several benchmark versions. Furthermore, we let FoREnSiC compute single-fault diagnoses only. Otherwise, the number of diagnoses grows to several hundreds for certain benchmark versions. The last two columns show the same information for the approach of [7], which has been implemented in the tool Bug-Assist. All experiments were performed on a notebook with an Intel Core i5-3320M processor running at 2.6 GHz, 8GB of RAM, and a 64 bit Linux operating system. The memory consumption was insignificant in our experiments.

Table 1: Detailed performance results.

Column	1	2	3	4	5	6	7	8	9
TCAS Benchmark		Our new approach				FoREnSiC [7]		Bug-Assist [6]	
Version	Matches Fault Model [-]	Error Det. Time [s]	Error Loc. Time [s]	Nr. of Cand- idates [-]	Nr. of Loc. Rep. [-]	Error Loc. Time [s]	Nr. of Loc. Rep. [-]	Error Loc. Time [s]	Nr. of Loc. Rep. [-]
1	Yes	5.2	20	10	4	19	22	7.8	16
2	Yes	5.2	6.9	4	2	14	15	9.8	17
3	No	4.5	111	35	13	15	12	10	17
4	No	5.3	22	10	0	17	20	7.5	17
5	No	4.3	84	33	7	27	12	4.3	18
6	Yes	5.2	6.0	2	2	17	20	5.6	17
7	Yes	5.1	6.0	4	1	15	11	7.6	17
8	Yes	5.2	6.8	4	1	15	12	8.1	15
9	Yes	5.2	20	10	2	14	19	8.6	13
10	Yes	5.3	6.9	4	4	23	18	11	18
11	Yes	5.3	6.9	4	4	32	13	6.3	9
12	No	4.4	89	35	5	28	21	5.5	18
13	Yes	4.3	111	35	8	20	12	6.7	16
14	Yes	4.7	104	35	2	15	3	7.0	8
15	Yes	5.3	35	20	6	18	11	4.4	18
16	Yes	5.2	6.8	4	1	15	11	7.7	16
17	Yes	5.2	6.8	4	1	14	11	8.0	16
18	Yes	5.2	6.8	4	1	14	11	7.8	16
19	Yes	5.2	7.0	4	1	15	11	7.9	16
20	Yes	5.3	20	10	2	14	21	7.3	17
21	Yes	5.0	18.4	10	2	11	21	10	17
22	Yes	5.1	18.3	10	2	11	18	7.1	16
23	Yes	5.2	18.4	10	2	11	19	8.7	13
24	Yes	5.1	18.5	10	2	11	21	10	17
25	Yes	5.2	20	10	3	18	21	6.8	16
26	No	4.2	93	33	8	20	12	6.2	17
27	No	4.3	87	33	7	26	12	4.2	18
28	Yes	5.3	10	4	2	20	10	8.2	14
29	Yes	5.2	5.6	1	1	14	14	7.4	13
30	Yes	5.5	6.0	2	2	14	14	9.1	17
31	No	4.9	129	48	10	16	15	3.4	15
32	No	4.8	102	48	7	16	15	3.2	17
33	No	5.1	6.8	4	0	26	12	0.3	1
34	No	4.2	30	35	1	28	11	4.3	18
35	Yes	4.4	7.0	4	2	18	10	10	18
36	Yes	5.0	129	35	13	19	22	4.3	15
37	No	5.2	6.1	2	0	19	12	7.4	15
38	No	5.2	5.2	0	0	2	0	0.3	1
39	Yes	5.2	20	10	3	17	21	7.0	16
40	No	4.2	80	41	8	18	19	7.6	16
41	No	5.0	17	9	3	16	19	6.4	16
average	66 %	5.0	37	15	3.5	17	15	6.9	15
median		5.2	18	10	2	16	14	7.4	16