Bug-Assist: Assisting Fault Localization in ANSI-C Programs*

Manu Jose[1] and Rupak Majumdar[1,2]

[1] University of California, Los Angeles,
Department of Computer Science, USA
[2] Max Planck Institute for Software Systems,
Kaiserslautern, Germany

Abstract. Several verification tools exist for checking safety properties of programs and reporting errors. However, a large part of the program development cycle is spend in analyzing the error trace to isolate locations in the code that are potential causes of the bug. Currently, this is usually performed manually, by stepping through the error trace in a debugger. We describe Bug-Assist, a tool that assists programmers localize error causes to a few lines of code. Bug-Assist takes as input an ANSI-C program annotated with assertions, performs bounded model checking to find potential assertion violations, and for each error trace returned by the model checker, returns a set of lines of code which can be changed to eliminate the error trace. Bug-Assist formulates error localization as a MAX-SAT problem and uses scalable MAX-SAT solvers. In experiments on a set of C benchmarks, Bug-Assist was able to reduce error traces to only a few lines of code. Bug-Assist is available as an Eclipse plug-in, enabling its easy deployment in the code development phase.

1 Introduction

Quality assurance is a major component of the software development cycle. Recent years have seen an explosion in static and dynamic analysis tools that can automatically look for classes of program errors. These tools take (unmodified) code as input, and produce error traces (and test cases) to demonstrate how assertions about correct behavior can be violated. However, after such tools report error traces, it is usually up to the programmer to debug and locate the faulty lines in the program, before modifying the program to eliminate the error trace. Thus, even with the support of automatic verification tools, a large part of the development cycle is spent in debugging, where the programmer looks at a long, failing, trace and tries to localize the problem to a few lines of source code that elucidate the cause of the problem.

We present Bug-Assist, a tool for fault localization for ANSI-C programs. The tool and the user manual can be downloaded at http://bugassist.mpi-sws.org. The input to our tool is a C program, instrumented with assertions which specify

* This research was funded in part by the NSF grant CCF-0546170.

the correct behavior of the program, and either a test case failing an assertion or an option to run a model checker looking for assertion violations. If a violation is found, the output is a minimal set of program statements such that there exists a way to replace these statements to correct the program execution.

Bug-Assist is available as a command-line tool and through a graphical user interface (GUI) as a plugin inside the popular integrated development environment (IDE) Eclipse. The GUI is similar to existing GUI based debugging tools inside Eclipse IDE, which are familiar to software engineers. So the programmer need not worry about the command line options or the internal symbolic analysis instead can look in to the highlighted potential buggy lines in the source code window.

Internally, Bug-Assist constructs a Boolean formula from a failing test case, and uses a MAX-SAT solver to identify minimal sets of instructions whose modification can eliminate the failing execution. The failing test case is obtained either directly from the testsuite for the program, or obtained using a bounded model checker (CBMC) integrated within Bug-Assist.

We evaluated Bug-Assist using programs from the Siemens set of benchmarks with injected faults [4]. In each case, Bug-Assist can efficiently and precisely determine the exact (to the human) lines of code that constitute the reason of the “bug”. The TCAS program in the testsuite is run with all the faulty versions in detail to illustrate the completeness of the tool. Each of these test cases consisted of 173 lines of code and the maximum time to localize the bug locations was 0.136 seconds with the average number of buggy lines came to 8% of the total lines of code [7]. The other 4 programs are used to show the scalability of the tool by using error trace reduction methods for real world programs. This shows the effectiveness of using the tool in parallel with the code development process.

While there has been a lot of research in fault localization [1,5,6], to the best of our knowledge, there is no publicly available tool for pin-pointing error locations. Delta-debugging tools, that minimize the input for a failing run, exist [9], but have a somewhat orthogonal emphasis: reducing the failure causing input. Bug-Assist fills in the gap between error traces and bug-fixing by pointing out exact locations of the code where corrections should be made instead of minimal code fragments demonstrating failure. We believe Bug-Assist can be effectively used in conjunction with a delta-debugging tool.

2 Tool Description

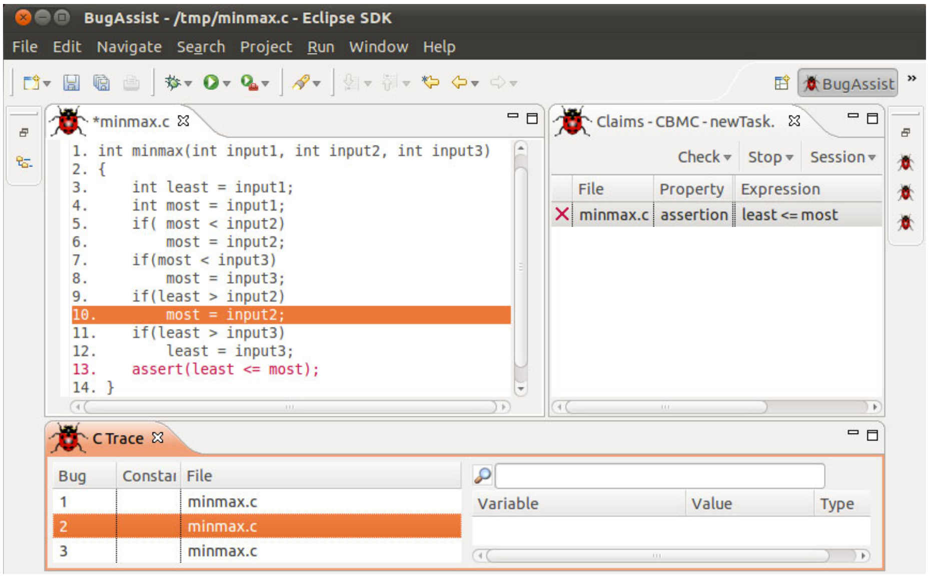
2.1 User Guide

The tool is available as an eclipse plugin and as a command line tool for linux platform. It takes as input an ANSI-C program with properties specified as *assert* statements. The tool can also generate implicit assertions for array bound checks, null pointer checks, etc. If the program violates the specifications, it outputs the potential lines of code which are the repair candidates for the program. For localizing error for a faulty program file.c, we can run Bug-Assist as follows:

./bugassist file.c –ba –maxsat <solver location>

where the option *–ba* outputs the potential repair locations. The *–maxsat* gives user the flexibility to choose the MAX-SAT solver by providing the solver executable location as Bug-Assist generates the MAX-SAT instances following the general MAX-SAT competition format. The detailed list of options can be obtained with the option *–help*.

2.2 The Integrated Debugging Environment


BugAssist - /tmp/minmax.c - Eclipse SDK
File Edit Navigate Search Project Run Window Help
BugAssist
*minmax.c
Claims - CBMC - newTask.
Check Stop Session
C Trace

```
1. int minmax(int input1, int input2, int input3)
2. {
3.     int least = input1;
4.     int most = input1;
5.     if( most < input2)
6.         most = input2;
7.     if(most < input3)
8.         most = input3;
9.     if(least > input2)
10.        most = input2;
11.    if(least > input3)
12.        least = input3;
13.    assert(least <= most);
14. }
```

	File	Property	Expression
✗	minmax.c	assertion	least <= most

Bug	Constai	File
1		minmax.c
2		minmax.c
3		minmax.c

Variable	Value	Type

Fig. 1. A screenshot of the tool localizing error for a simple program

To improve the usability of our tool, we have built an Eclipse plugin and a GUI to help the programmer find bug locations during the development process. During the code development phase, the programmers might only be interested in modules or files under the current development branch. Therefore Bug-Assist IDE creates a new project which gives the users the flexibility to pick the files and functions for which they want to check for property violations and debug the problems only within those files.

Figure 1 shows Bug-Assist IDE analyzing a simple program "minmax.c", which computes the minimum and maximum among its input arguments. There is an error in the program at line 10. During the first run of Bug-Assist, it lists out the assertions which are specified in the system. In the example program there is an explicit assertion claiming "least" should be less than or equal to "most" and is listed in the claims window. Additionally, one can specify implicit assertions for null pointer checks or array-bounds checks, by setting properties of the model checker.

When the model checker returns the violated assertions, the programmer can now double click on each of the violated assertions to find the potential lines of code that lead to the violation of that property. The three potential bug locations analyzed by the tool is given in the "C Trace" window in Figure 1. Selecting each of these bug locations highlights the corresponding location in the source code viewer, helping the programmer to effectively visualize the problem cause and to provide a fix. After fixing the error, (in this case it is to change the variable "most" to "least" at line 10) the user can run the model checker again to check if the fix is correct and does not break for other inputs.

2.3 Tool Architecture

Internally, our algorithm leverages symbolic analysis of software based on Boolean satisfiability, and reduces the problem to *maximum Boolean satisfiability*. Maximum Boolean satisfiability (MAX-SAT) is the problem of determining the maximum number of clauses of a given Boolean formula in CNF that can be satisfied by any given assignment.

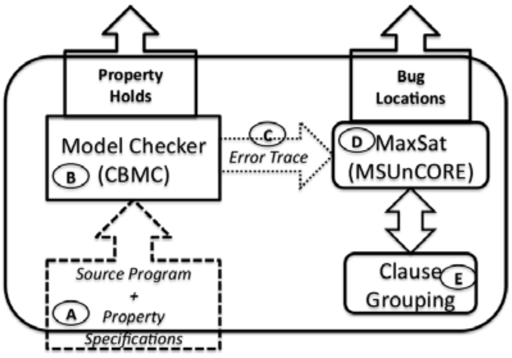


Fig. 2. The tool architecture

Figure 2 shows the tool architecture with the program flow. We describe the tool internals informally using the same program shown in Figure 1. The function `minmax` sets the "least" and "most" variables to the first arguments and then eventually sets those variable to the minimum and maximum value respectively from the input variables (shown in lines 5–12). After comparing "least" with "input2", instead of updating the "least" variable the program updates the "most" variable.

Bug-Assist performs the following two steps to localize the bug. First, from the program, it uses bounded model checking [2,3] (using the tool CBMC [3]) to construct a symbolic *trace formula*: a Boolean formula in conjunctive normal form such that the formula is satisfiable iff the program execution is feasible. For the example, CBMC (label B in Figure 2) gives a counter example with

$\texttt{input1} = 10$, $\texttt{input2} = 1$, and $\texttt{input3} = 5$, which makes $\texttt{most} = 1$ and violates the assertion at line 13. The trace formula TF encoding the program semantics is:

$$\begin{aligned}\mathsf{TF} \equiv\, & \texttt{least}_1 = \texttt{input1}_1 \wedge \texttt{most}_1 = \texttt{input1}_1 \wedge \texttt{guard}_1 = \texttt{most}_1 < \texttt{input2}_1 \wedge \\ & \texttt{most}_2 = (!\texttt{guard}_1 ? \texttt{most}_1 : \texttt{input2}_1) \wedge \texttt{guard}_2 = \texttt{most}_2 < \texttt{input3}_1 \wedge \\ & \texttt{most}_3 = (!\texttt{guard}_2 ? \texttt{most}_2 : \texttt{input3}_1) \wedge \texttt{guard}_3 = \texttt{least}_1 > \texttt{input2}_1 \wedge \\ & \texttt{most}_4 = (!\texttt{guard}_3 ? \texttt{most}_3 : \texttt{input2}_1) \wedge \texttt{guard}_4 = \texttt{least}_1 > \texttt{input3}_1 \wedge \\ & \texttt{least}_2 = (!\texttt{guard}_4 ? \texttt{least}_1 : \texttt{input3}_1)\end{aligned}$$

We then extend the trace formula by conjoining it with constraints so that the final states ensure the program post-condition:

$$\Phi \equiv \underbrace{\texttt{input1}_1 = 10 \wedge \texttt{input2}_1 = 1 \wedge \texttt{input3}_1 = 5}_{\text{test input}} \wedge \underbrace{\mathsf{TF}}_{\text{trace formula}} \wedge \underbrace{\texttt{least}_2 \leq \texttt{most}_4}_{\text{assertion}}$$

The extended trace formula essentially states that starting from the initial condition, executing the program trace leads to a state satisfying the post-condition. Notice that the extended trace formula for a failing execution must be unsatisfiable.

Second, it feeds the extended trace formula in conjunctive normal form (CNF) to a *partial maximum satisfiability solver* (label D in Figure 2). In partial MAX-SAT, the input clauses can be marked *hard* or *soft*, and the MAX-SAT instance finds the maximum number of soft clauses that can be satisfied by an assignment which satisfies every hard clause. In our algorithm, we mark the input constraints (that ensure that the input is a failing test) as well as the post-condition as hard. This is necessary: otherwise, the MAX-SAT algorithm can trivially return that changing an input or changing the post-condition can eliminate the failing execution.

In the case of Φ, we mark the constraints coming from the test input and the assertion as hard, and leave the clauses in the trace formula soft. We use an off-the-shelf MAX-SAT solver [8] to compute a maximal set of clauses of the extended trace formula that can be satisfied, and take the complement of this set as a candidate set of clauses that can be changed to make the entire formula satisfiable. Since each clause in the extended trace formula can be mapped back to a statement in the code, this identifies a candidate localization of the error in terms of program statements. Note that there may be several minimal sets of clauses that can be found in this way, and we enumerate each minimal set as candidate localizations for the user.

The clauses returned by MAX-SAT point to line 9 as a potential error location in the first iteration of our program. In the next iteration of MAX-SAT, we make the clauses arising out of line 9 hard and ask for any other correction locations. We repeat this process until MAX-SAT gives the problem to be unsatisfiable. All the three error locations reported by tool is shown in Figure 1.

In our implementation, we group clauses (label E) arising out of the same program statement together making the resulting MAX-SAT instance simple.

This ensures our algorithm localizes errors at the program statement level. Error localization can be performed at other levels of granularity as well. For example, to localize bugs at the function or module level, we can group clauses coming from the same function or module in the MAX-SAT instance.

A formal definition of the localization algorithm as well as experimental evaluations can be found in [7].

References

1. Ball, T., Naik, M., Rajamani, S.K.: From symptom to cause: localizing errors in counterexample traces. In: POPL 2003: Principles of Programming Languages, pp. 97–105. ACM, New York (2003)
2. Biere, A., Cimatti, A., Clarke, E.M., Fujita, M., Zhu, Y.: Symbolic model checking using SAT procedures instead of BDDs. In: DAC 1999: Design Automation Conference, pp. 317–320. ACM, New York (1999)
3. Clarke, E., Kröning, D., Lerda, F.: A tool for checking ANSI-C programs. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 168–176. Springer, Heidelberg (2004)
4. Do, H., Elbaum, S., Rothermel, G.: Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. Empirical Software Engineering 10(4), 405–435 (2005)
5. Groce, A., Chaki, S., Kroening, D., Strichman, O.: Error explanation with distance metrics. Int. J. Softw. Tools Technol. Transf. 8(3), 229–247 (2006)
6. Jones, J.A., Harrold, M.J.: Empirical evaluation of the tarantula automatic fault-localization technique. In: ASE 2005: Automated Software Engineering, pp. 273–282. ACM, New York (2005)
7. Jose, M., Majumdar, R.: Cause clue clauses: Error localization using maximum satisfiability. In: PLDI 2011: Programming Language Design and Implementation, ACM, New York (2011)
8. Marques-Silva, J., Planes, J.: Algorithms for maximum satisfiability using unsatisfiable cores. In: DATE 2008: Design, Automation and Test in Europe, pp. 408–413. ACM, New York (2008)
9. Zeller, A.: Isolating cause-effect chains from computer programs. In: Daemen, J., Rijmen, V. (eds.) FSE 2002. LNCS, vol. 2365, pp. 1–10. Springer, Heidelberg (2002)