Understanding Counterexamples with `explain`

Alex Groce, Daniel Kroening, and Flavio Lerda

Computer Science Department, Carnegie Mellon University
Pittsburgh, PA 15213

Abstract. The counterexamples produced by model checkers are often lengthy and difficult to understand. In practical verification, showing the existence of a (potential) bug is not enough: the error must be understood, determined to not be a result of faulty specification or assumptions, and, finally, located and corrected. The `explain` tool uses distance metrics on program executions to provide automated assistance in understanding and localizing errors in ANSI-C programs. `explain` is integrated with CBMC, a bounded model checker for the C language, and features a GUI front-end that presents error explanations to the user.

1 Introduction

In an ideal world, given a detailed error trace, a programmer would always be able to quickly identify and correct the faulty portion of the code or specification. Practical experience, however, indicates that this is not the case. Understanding a counterexample often requires as much effort as preparing a program for model checking. As software model checking has become more concerned with practical applicability, the need for automated assistance in understanding counterexamples has been recognized [2, 6]. The `explain` tool provides users with assistance in focusing on the relevant portions of source code and in understanding the causal dependencies involved in an error.

CBMC [7] is a tool for verifying ANSI-C programs. CBMC is a bounded model checker (BMC) [3]: it produces from a C program a Boolean formula satisfiable by executions of the program that violate its specification (counterexamples). The model checker supports pointer constructs, dynamic memory allocation, recursion, and the `float` and `double` data types. CBMC also features a graphical user interface designed to resemble an IDE (Integrated Development Environment) that allows users to interactively step through counterexample traces.

`explain` uses the same bounded model checking engine to further analyze counterexample traces produced by CBMC. In particular, `explain` uses *distance metrics* on program executions [5], in a manner inspired by the counterfactual theory of causality [8], to provide a number of automatic analyses:

- Given a counterexample execution, `explain` can automatically produce an execution that is as similar as possible to the failing run but *does not* violate the specification.

R. Alur and D.A. Peled (Eds.): CAV 2004, LNCS 3114, pp. 453–456, 2004.

- `explain` can also automatically produce a new counterexample that is as *different* as possible from the original counterexample.
- Finally, `explain` can determine causal dependencies between predicates in an execution.

`explain` is used through the same GUI as CBMC. The interface allows users to step through explanatory traces as they would in a debugger (with the ability to step forwards and backwards). Portions of the code that `explain` suggests may be faulty are highlighted for the user.

2 Using `explain`

Using `explain` is an interactive process. The tool assists the user in understanding counterexamples, but knowledge of the program (and the specification) is necessary to guide the tool. As an example, we will use `explain` to narrow in on an error in a small but non-trivial C program.

2.1 Debugging TCAS

TCAS (Traffic Alert and Collision Avoidance System) is an aircraft conflict detection and resolution system used by all US commercial aircraft. The Georgia Tech version of the Siemens suite [9] includes 41 buggy versions of ANSI-C code for the Resolution Advisory (RA) component of the TCAS system. A specification for this code (in the form of assertions) is available from another study [4].

The first step in using `explain` to understand an error is to produce a counterexample. We load `tcas.c` into the GUI and run the CBMC model checker. After a few seconds, the GUI reports that the assertion on line 257 has been violated.

The counterexample execution passes through 112 states. Single-stepping through the trace looking for a bug is not an appealing prospect, so we turn to `explain` for assistance in understanding what is wrong with our code. We run `explain` on the counterexample to find a successful execution that is as similar as possible to the failing run. `explain` uses the PBS [1] pseudo-Boolean solver to produce this trace, and lists the changes made to the original counterexample. The GUI highlights the lines that are involved in the changes[1].

Unfortunately, the explanation is less than useful. The failed assertion in the counterexample is an implication:

```
P3_BCond = ((Input_Up_Separation >= Layer_Positive_RA_Alt_Thresh)&&
           (Input_Down_Separation >= Layer_Positive_RA_Alt_Thresh)&&
           (Input_Own_Tracked_Alt < Input_Other_Tracked_Alt));
assert(!(P3_BCond && PrB)); // P3_BCond -> ! PrB
```

The successful execution most similar to the counterexample changes the value of `Input_Down_Separation` such that it is now `< Layer_Positive_RA_Alt_Thresh`, and *no other values*. We are really interested in finding out why, given that

[1] `explain` uses a causal slicing algorithm [5] to remove changes unrelated to the error.

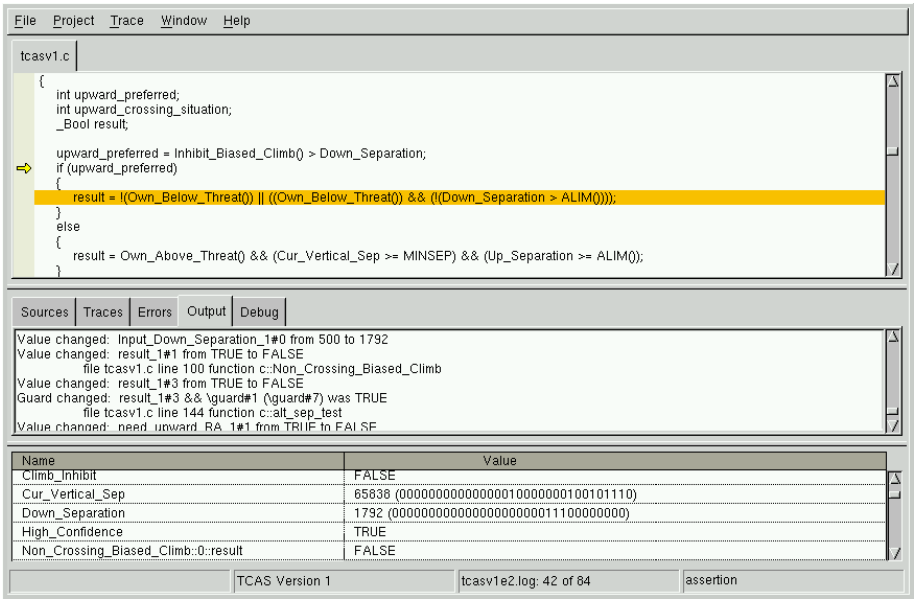

Fig. 1. Correctly locating the error in `tcas.c`.

`P3_BCond` holds, `PrB` also holds. In other words, we want to know the cause for the value of the consequent, not the antecedent, of the implication. Requesting the more precise explanation is a simple matter of adding the constraint `assume(P1_BCond)` and then rerunning `explain`.

The new explanation is more interesting (Figure 1). `Input_Down_Separation` is, again, altered. This time the value has increased, maintaining the original value of `P3_BCond`. Stepping through the source code we notice the first highlighted line in the run:

```
result = !(Own_Below_Threat()) || ((Own_Below_Threat()) &&
          (!(Down_Separation > ALIM())));
```

Most of the values in the expression are unchanged from the erroneous run. Only `Down_Separation` has changed, causing `result` to be `FALSE` instead of `TRUE`. In the original run, `Down_Separation` was 500, and now it is 1792. A quick examination of the other 4 highlighted lines shows that they simply propagate the value of `result` computed. If the comparison of `Down_Separation` and `ALIM()` had resulted in `TRUE` in the original run, the assertion would have held. Looking at the code, we notice that the `ALIM` function returns `Positive_RA_Alt_Thresh[Alt_Layer_Value]`. The variable watches window reveals that `ALIM()` has the value 500 in both runs.

Suspiciously, the value of `Down_Separation` in the counterexample was also 500, and the error did not appear when that value changed. It seems likely that the value of this expression is the source of the problem. In order to make the expression's value in the faulty run match the value in the successful run, we need to change the `>` into a `>=` comparison.

We modify the source code to reflect our hypothesis about the source of the error and rerun CBMC. This time, the model checker reports that verification is successful: the program satisfies its specification.

In other experiments, `explain` produced a 1 line (correct) localization of a 127 state locking-protocol counterexample for a 2991 line fragment of a real-time OS microkernel.

3 Conclusions and Future Work

`explain` is a tool that uses a model checker to assist users in debugging programs (or specifications). The tool is fully integrated with a model checker that precisely handles a rich variety of the features of the ANSI-C language, and provides a graphical front-end for user interactions. Case studies have demonstrated that `explain` can successfully localize errors in a number of programs.

In the future, we hope to improve both the graphical interface to `explain` and the underlying explanation algorithms, based on experience with more case studies and user feedback.

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. 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.
3. 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.
4. 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.
5. A. Groce. Error explanation with distance metrics. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 108–122, 2004.
6. A. Groce and W. Visser. What went wrong: Explaining counterexamples. In *SPIN Workshop on Model Checking of Software*, pages 121–135, 2003.
7. 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.
8. D. Lewis. Causation. *Journal of Philosophy*, 70:556–567, 1973.
9. G. Rothermel and M. J. Harrold. Empirical studies of a safe regression test selection technique. *Software Engineering*, 24(6):401–419, 1999.