FoREnSiC– An Automatic Debugging Environment for C Programs*

Roderick Bloem[1], Rolf Drechsler[2], Görschwin Fey[2], Alexander Finder[2], Georg Hofferek[1], Robert Könighofer[1], Jaan Raik[3], Urmas Repinski[3], André Sülflow[2]

[1]Graz University of Technology, Austria [2]University of Bremen, Germany
[3]Tallinn University of Technology, Estonia

Abstract. We present FoREnSiC, an open source environment for automatic error detection, localization and correction in C programs. The framework implements different automated debugging methods in a unified way covering the whole design flow from ESL to RTL. Currently, a scalable simulation-based back-end, a back-end based on symbolic execution, and a formal back-end exploiting functional equivalences between a C program and a hardware design are available. FoREnSiC is designed as an extensible framework. Its infrastructure, including a powerful front-end and interfaces to logic problem solvers, can be reused for implementing new program analysis or debugging methods. In addition to the infrastructure, the back-ends, and a few experimental results, we present an illustrative application scenario that shows FoREnSiC in use.

1 Introduction

Debugging incorrect programs is labor-intensive, frustrating, and costly, yet unavoidable in modern software and hardware development. Errors have to be detected, located and corrected. Many methods and tools exist to automate error detection, e.g., automatic test case generation or model checking. Error localization and correction are mostly done manually. At the same time, these are the most challenging steps. Understanding the program and tracking down errors is time-consuming. Bug fixes often do not consider special cases, have side-effects, or create new bugs. The need for further automation and tool support is obvious.

Existing tools and methods automate error localization and correction in different settings. An extension of the Tarantula fault localizer [10] with mutation-based repair is presented in [8]. Model-based diagnosis [13] has been applied in various settings already. A counterexample-based repair method is presented in [5]. Sketch [15] uses similar techniques for program sketching. Also, several approaches have been proposed to check equivalence between system-level specifications and designs in *Hardware Description Language* (HDL), e.g. [6,16]. In contrast to our framework, these existing tools implement only one specific method in isolation. Refer to [11,12] for a more elaborate discussion of related work.

* This work was supported in part by the European Commission through project DIAMOND (FP7-2009-IST-4-248613), and by the Austrian Science Fund (FWF) through the national research network RiSE (S11406-N23).

We present FoREnSiC, an automatic debugging environment for C programs. FoREnSiC is short for "**Fo**rmal **R**epair **En**vironment for **Si**mple **C**", but it has already outgrown its name: it also detects and locates errors, and it applies also semi-formal and dynamic methods. FoREnSiC makes two main contributions.

First, it implements various debugging methods in different back-ends. They can be accessed in a unified way and provide different trade-offs between scalability and reasoning power. Currently, there are three back-ends. The *simulation-based back-end* [12] locates and repairs errors by executing the program on given test vectors, featuring good scalability. It is similar to [8], but uses techniques like program slicing to obtain better results. The *symbolic back-end* is based on symbolic execution and SMT-solving [11]. It uses model-based diagnosis [13] and a repair method similar to [15] and [5] (but with templates [7]) to synthesize repairing expressions. The *cut-based back-end* exploits functional equivalences between a C program and any HDL design for error localization and correction.

Second, FoREnSiC also serves as a framework for implementing new program analysis, verification, and debugging techniques. The infrastructure includes a GCC-based front-end to transform C programs into a graph-based representation. Moreover, FoREnSiC provides data structures to represent logic formulas and interfaces to logic solvers for solving them. FoREnSiC is available as open-source tool at `http://www.informatik.uni-bremen.de/agra/eng/forensic.php`.

This paper is organized as follows. Section 2 illustrates FoREnSiC on an example. Section 3 explains FoREnSiC's architecture, the internal model, the front-end, and the back-ends. Section 4 shows experimental results and Section 5 concludes.

2 Application Scenario

We demonstrate the features of FoREnSiC on the following scenario. Assume we want to implement an algorithm to compute the *Greatest Common Divisor* (GCD) of two integers. We start with the following draft program in C.

```
1  unsigned gcd(unsigned u,
2               unsigned v) {
3    unsigned sh = 0, res;
4    if(u == 0 || v == 0) {
5      res = 0;
6      return res;
7    }
8    while(((u | v) & 1) == 0) {
9      u >>= 1;
10     v >>= 1;
11     ++sh;
12   }
13   while((u & 1) == 0)
14     u >>= 1;
15   do {
16     while((v & 1) == 0)
17       v >>= 1;
18     if(u <= v) {
19       v += u;
20     } else {
21       unsigned diff = u - v;
22       u = v;
23       v = diff;
24     }
25     v >>= 1;
26   } while(v != 0);
27   res = u << sh;
28   return res;
29 }
```

Now we want to verify the correctness of our program. We use FoREnSiC's symbolic back-end to compare it to the Euclidean algorithm, which serves as a reference. The back-end detects an error and automated debugging commences. First, error localization reports the "0" in line 5 as potentially faulty. Next, the back-end synthesizes the following expressions to substitute "0" with: `u + v`

and `4294967295 & u | 4294967295 & v`. The reason is clear: our program computes `gcd(0,x) = gcd(x,0) = 0` for any x, but the result should be x instead. Replacing "`0`" with "`u + v`" fixes this bug. Since 4294967295 is `0xFFFFFFFF`, the second suggestion is actually "`u | v`", which is correct as well. We can now decide which fix we prefer. The symbolic back-end took only 6 seconds to locate and fix this bug. When analyzing the revised program in more detail, the back-end detects another error but is unable to locate or fix it within reasonable time. Therefore, we now switch to the simulation-based back-end.

The simulation-based back-end performs simulation-based verification, diagnosis, and mutation-based repair. For our example, verification fails if enough test cases are provided. Thus, diagnosis starts by ranking statements according to their suspiciousness. In case of a single fault assumption (see Section 3.1), the fault candidates are those statements which occur in all failing test cases. Next, mutation-based repair is applied to one fault candidate after the other. Each mutated design is verified by simulation to check whether the mutation constitutes a repair. For our example, the back-end finds a fix after `149` mutations. The mutation that fixes the fault is the replacement of the assignment operator `+=` in line 19 by `-=`. The two back-ends complement each other: while the simulation-based back-end had no difficulties debugging the second bug, it could not come up with the suggestions produced by the symbolic back-end for the first bug. The reason is that a mutation from `0` to `u | v` would be too far-fetched.

Assume that we now want to implement this algorithm in hardware. We use the cut-based back-end to verify equivalence between the HDL implementation and the C program serving as specification. The cut-based back-end implements a new verification, localization and repair method based on functional equivalences, so-called cutpoints, between two designs. For the equivalence check, the cut-based back-end determines corresponding output values in the two design descriptions automatically. Mismatches lead to counterexamples which are used to debug the HDL description or the C program respectively. The cut-based back-end localizes components in the design under debug to be replaced by corresponding ones from the reference to fix a bug. Alternatively, the counterexamples may be used as inputs for the simulation-based back-end for further debugging of the C program. The equivalence proof for the GCD example took 15 minutes, unrolling the hardware design for up to 78 time cycles.

3 Description of FoREnSiC

FoREnSiC consists of three functional parts: the front-end, the model, and the back-ends. Fig. 1 illustrates the architecture in a simplified form. A C program is the main input. The front-end parses it and builds an internal model in form of a flow graph representing the program in *Static Single Assignment* (SSA) form [1]. Each graph node represents a statement and is linked with its abstract syntax tree. All parts of the model have references to the original source code in terms of line and column numbers for communicating results to the user. The front-end is based on the GCC plug-in API, so that complete C/C++ is supported

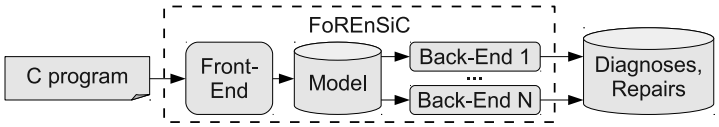

Fig. 1. The architecture of FoREnSiC.

and upward compatibility is ensured. FoREnSiC also includes classes to represent logic formulas and interfaces to SMT-solvers. This yields an infrastructure such that FoREnSiC can be easily extended with new back-ends and features.

FoREnSiC currently contains three back-ends operating on the model. They implement different advanced debugging methods. The back-ends require different supplementary inputs such as test vectors or a reference model. Not all back-ends fully support all language features. For instance, the symbolic back-end cannot accurately reason about pointer arithmetic. However, our main focus are programs modeling hardware designs, where these advanced features do not play such an important role. Details can be found in the manual [3].

3.1 Simulation-Based Debugging

The simulation-based back-end [12] simulates the design to detect and fix faults. The specification can be given as input vectors together with either expected output responses or a reference program in C. In the latter case, both programs are simulated on the same inputs while comparing the outputs.

The back-end performs three steps. First, simulation-based verification is performed. Second, if an error is detected, statistical methods are used to determine fault candidates. In case of a single fault assumption, nodes in the model of the design are associated with the number of failing simulation runs in which they occurred. The nodes with the maximum numbers are reported as fault candidates. In case of a multiple fault assumption, candidates are selected using the method presented in [8]. Dynamic slicing is applied to discard candidates that do not influence the simulation result. Finally, mutation-based repair is applied to fix the error by mutating (i.e., modifying) operators and numbers, and checking if this renders the program correct. The types of mutations can be configured.

3.2 Symbolic Debugging

The symbolic back-end implements the debugging method of [11]. It uses `assert` statements as specification. Assertions also allow flexible comparisons with reference implementations. Debugging is performed in three steps. First, symbolic or concolic execution is used to compute a formula defining when the program satisfies the specification. The symbolic execution engine has been implemented from scratch, the concolic engine is based on CREST [4]. Both provide many options for configuring the thoroughness of the analysis. In the second step, the diagnosis

engine computes potentially faulty components using model-based diagnosis [13]. Finally, the repair engine synthesizes new implementations of the faulty components using templates for expressions and iterative refinements which are guided by counterexamples [11]. Diagnosis and repair rely on SMT-solving. Currently, the solvers Yices and Z3 can be used with linear integer arithmetic and bit-vector arithmetic, either via their C-APIs or via SMT-LIBv2 [2].

3.3 Cut-based Debugging

This back-end formally verifies the equivalence between a C program, taken as reference specification, and an implementation in HDL. In case of a mismatch, a counterexample is returned, and input stimuli similar to the counterexample are generated for simulation of both designs. After simulation, a frontier of functional equivalences within both designs is computed, i.e. parts of implementation and specification which are found to be equivalent. Starting at the frontier, components within the implementation are replaced by components from the specification. Each replacement is checked whether it leads to further functional equivalences between implementation and specification. If this is the case, the repair is verified formally. In case of inequivalence, the repair engine suggests which parts of the implementation may be replaced by which parts of the specification in order to achieve functionally equivalent designs. Diagnosis and repair rely on SAT-solving using MiniSat2 [9].

4 Experimental Results

We briefly compare our back-ends on the `tcas` benchmarks of the Siemens suite [14]. `tcas` implements a collision avoidance system for aircrafts, has 12 integer inputs, around 180 lines of code, and comes in 41 faulty variants. The simulation-based back-end fixes 26 versions, including 5 that cannot be solved with the symbolic back-end and 6 that cannot be solved with the cut-based back-end. The symbolic back-end fixes 23 versions, including 2 that cannot be solved with the simulation-based method and 5 that cannot be solved with the cutpoint-based back-end. The cutpoint-based back-end solves 29 versions, including 9 that cannot be solved with the simulation-based back-end and 11 that cannot be solved with the symbolic back-end. This demonstrates that the back-end complement each other. Due to space constraints, we refer to [11] and [12] for more experimental data. The FoREnSiC archive contains additional data.

5 Summary and Conclusion

FoREnSiC is a novel environment for automating error detection, location, and correction in C programs. The back-ends can be accessed in a unified way, and complement each other in features and characteristics. This makes FoREnSiC a powerful tool for reducing manual debugging effort. FoREnSiC is also interesting

for developers as an open-source framework for new program analysis and debugging techniques. Using the existing infrastructure, the developer can focus on the new methods rather than parsing or interfacing solvers. By this, FoREnSiC can alleviate and stimulate further research and development in the challenging fields of automated hardware and software verification and debugging.

Future work includes improving the back-ends regarding performance, the subset of C that can be handled, and their fault models. Integrating back-ends more tightly, or combining their methods in a hybrid back-end is also planned.

References

1. B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *POPL*, pages 1–11. ACM, 1988.
2. C. Barrett, A. Stump, and C. Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2010.
3. R. Bloem, R. Drechsler, G. Fey, A. Finder, G. Hofferek, R. Könighofer, J. Raik, U. Repinski, and A. Sülflow. FoREnSiC - A Formal Repair Environment for Simple C. `http://www.informatik.uni-bremen.de/agra/eng/forensic.php`, 2011.
4. J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. In *ASE*, pages 443–446. IEEE, 2008.
5. K.-H. Chang, I. L. Markov, and V. Bertacco. Fixing design error with counterexamples and resynthesis. In *ASP-DAC*, pages 944–949. IEEE, 2007.
6. E. Clarke, D. Kroening, and K. Yorav. Behavioral consistency of C and Verilog programs using bounded model checking. In *DAC*, pages 368–371. ACM, 2003.
7. M. Colón, S. Sankaranarayanan, and H. Sipma. Linear invariant generation using non-linear constraint solving. In *CAV*, pages 420–432. Springer, 2003. LNCS 2725.
8. V. Debroy and W. E. Wong. Using mutation to automatically suggest fixes for faulty programs. In *ICST*, pages 65–74. IEEE, 2010.
9. Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In *Theory and Applications of Satisfiability Testing*, pages 502–518. Springer, 2003.
10. J. A. Jones and M. J. Harrold. Empirical evaluation of the Tarantula automatic fault-localization technique. In *ASE*, pages 273–282. ACM, 2005.
11. R. Könighofer and R. Bloem. Automated error localization and correction for imperative programs. In *FMCAD*, pages 91–100. FMCAD Inc, 2011.
12. J. Raik, U. Repinski, H. Hantson, M. Jenihhin, G. Di Guglielmo, G. Pravadelli, and F. Fummi. Combining dynamic slicing and mutation operators for ESL correction. In *ETS*, pages 1–6. IEEE, 2012.
13. R. Reiter. A theory of diagnosis from first principles. *Artif. Intell.*, 32(1):57–95, 1987.
14. Siemens Corporate Research. Siemens benchmark suite. `http://pleuma.cc.gatech.edu/aristotle/Tools/subjects`.
15. A. Solar-Lezama, L. Tancau, R. Bodik, V. Saraswat, and S. A. Seshia. Combinatorial sketching for finite programs. In *ASPLOS*, pages 404–415. ACM, 2006.
16. S. Vasudevan, J.A. Abraham, V. Viswanath, and J. Tu. Automatic decomposition for sequential equivalence checking of system level and RTL descriptions. In *MEMOCODE*, pages 71–80. IEEE, 2006.