FoREnSiC– An Automatic Debugging Environment for C Programs

Roderick Bloem* Rolf Drechsler† Görschwin Fey† Alexander Finder† Georg Hofferek*
Robert Könighofer* Jaan Raik‡ Urmas Repinski‡ André Sülflow†
*Graz University of Technology, Austria †University of Bremen, Germany
‡Tallinn University of Technology, Estonia

I. About FoREnSiC

FoREnSiC is an extensible environment making various debugging methods for C programs accessible in a unified way. FoREnSiC is short for "**Fo**rmal **R**epair **En**vironment for **Si**mple **C**", but it has already outgrown its name in two respects. First, it does not only perform error correction, but also error detection and localization. Second, it does not only apply formal methods, but also semi-formal and dynamic methods, implemented in different back-ends. The back-ends are accessible in a unified way allowing for trade-offs between scalability and reasoning power. Additionally, FoREnSiC also serves as a framework for implementing new program analysis, verification, and debugging techniques. FoREnSiC is available as open-source tool at [1].

FoREnSiC consists of three functional parts: the front-end, the model, and the back-ends. Figure 1 illustrates the architecture. A C program is the main input. The front-end is a GCC plug-in that parses this program and builds an internal model in form of a flow graph.

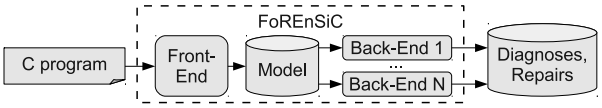

Fig. 1. The architecture of FoREnSiC.

FoREnSiC currently encompasses three back-ends described in the following application scenario.

II. Application Scenario

Assume we draft the C program in Fig. 2 to implement an algorithm to compute the *Greatest Common Divisor* (GCD) of two integer numbers. We use FoREnSiC's symbolic back-end relying on symbolic execution and SMT-solving [2] to compare the program with the Euclidean algorithm, which serves as a golden reference model. It uses model-based diagnosis and a repair based on templates to synthesize repairing expressions. The back-end detects an error and automated debugging commences. First, error localization reports the "`0`" in line 4 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. Replacing "`0`" with "`u + v`" fixes this bug. The second suggestion is "`u | v`", which is correct as well. Which fix should be taken is up to us. The symbolic back-end takes about 6 seconds to locate and fix

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).

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

Fig. 2. Draft for a C program

this bug. When analyzing the revised program in more detail, the back-end detects another error but it 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 fixes errors using simulation-based verification, error localization, and mutation-based repair. For the GCD example, the verification step fails if sufficient test cases are provided. Diagnosis starts by ranking statements according to their suspiciousness. For each fault candidate mutation-based repair is applied. The mutated designs are verified by simulation to check which mutation constitutes a repair. For our example, the back-end finds a fix after `149` mutations by replacing the assignment operator `+=` in line 17 by `-=`. 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. This nicely illustrates how well the different back-ends complement each other.

In a next step, assume we would like to implement this algorithm in hardware. We use the equivalence-checking back-end to check our program for equivalence with an HDL implementation using WoLFram [3]. The C program to calculate the GCD is compared to an HDL description with a data-width of 10 bit and equivalence was proven. The proof took 2 hours and 45 minutes. The hardware design was unrolled for up to 78 time cycles. Here, the C program is not required to contain timing information.

References

[1] 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.

[2] R. Könighofer and R. Bloem. Automated error localization and correction for imperative programs. In *International Conference on Formal Methods in Computer Aided Design*, 2011. To appear.

[3] A. Sülflow, U. Kühne, G. Fey, D. Große, and R. Drechsler. WoLFram – a word level framework for formal verification. In *International Workshop on Rapid System Prototyping*, pages 11–17, 2009.