FAuST: A Framework for Formal Verification, Automated Debugging, and Software Test Generation*

Heinz Riener[1] and Görschwin Fey[1,2]

[1] Institute of Computer Science, University Bremen, Germany,
{hriener,fey}@informatik.uni-bremen.de,
http://www.informatik.uni-bremen.de/agra/

[2] Institute of Space Systems, German Aerospace Center, Germany,
goerschwin.fey@dlr.de,
http://www.dlr.de/irs/

Abstract. We present FAuST, an extensible framework for Formal verification, Automated debugging, and Software Test generation. Our framework uses a highly customizeable *Bounded Model Checking* (BMC) algorithm for formal reasoning about software programs and provides different applications, e.g., property checking, functional equivalence checking, test case generation, and fault localization. FAuST supports dynamic execution and parallel symbolic reasoning using the LLVM compiler infrastructure and an abstraction layer for decision procedures.

Keywords: Formal verification, Debugging, SAT

1 Introduction

Bounded Model Checking (BMC) [3,7] is a technique to check whether finite-state systems conform to their specifications. BMC searches for counterexamples of bounded length and successively increases the bound until either a counterexample is found or the system's correctness can be guaranteed. The BMC problem is represented symbolically as multiple instances of the *Satisfiability* (SAT) problem. In practice BMC serves as a refutation technique because BMC problems often exhaust a resource limit before the system is proven correct. The instances are then solved using a corresponding *Decision Procedures* (DP), called *Satisfiability Modulo Theories* (SMT) solver.

More recently, BMC is used in software verification [6,12]: the behavior of a program is extracted from its source code and modeled using logic formulae. Today, flexible compilers like the *Low Level Virtual Machine* (LLVM) [14] compiler allow for program analysis and verification directly on the compiler's intermediate representation.

We present FAuST, an extensible framework for Formal verification, Automated debugging, and Software Test generation. FAuST offers a tool bench for different verification and debugging applications exploiting their similarities. The input of each FAuST tool is a software program. The output depends on its application. For instance, in fault-based test generation [22] the output is a test suite and in fault localization [23] the output is a set of potentially faulty program locations. The core engine of each tool is a highly customizable BMC algorithm.

The conceptual architecture of FAuST is built in three layers: (1) in the *program layer* FAuST deals with analyzing and transforming the input program. (2) In the *application layer* FAuST chooses a suitable background theory and

* This work was supported by the German Research Foundation (DFG, grant no. FE 797/6-1)

builds a SAT problem from the transformed program depends on the application. (3) In the *logic layer* the SAT problem is simplified and solved using SAT and SMT solvers.

Figure 1 shows the flow of the BMC tool in the FAuST framework for property checking. Dashed boxes denote objects and solid boxes denote transformations on those objects. In the program layer we leverage the LLVM compiler to lower the input program to LLVM's intermediate representation, LLVM-IR. In the application layer we instantiate an encoder with respect to the application, i.e., a customized BMC algorithm which generates a SAT instance from the transformed program. In the logic layer we use metaSMT [10] as a generic API interface to different SAT and SMT solvers. Other FAuST tools operate similarly.

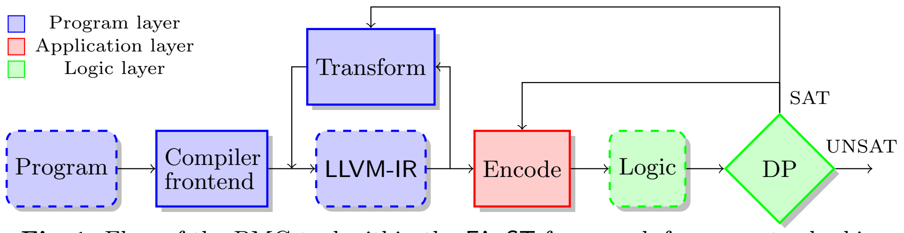

Fig. 1: Flow of the BMC tool within the FAuST framework for property checking

FAuST is the first tool bench which integrates formal verification, automatic debugging, and test generation into a unified framework. The main features are: (1) state-of-the-art compiler technology built on the LLVM compiler infrastructure, (2) dynamic execution using Just-In-Time (JIT) compilation, (3) an abstraction layer for decision procedures leveraging metaSMT, and (4) parallel solving using multiple SAT and SMT solvers simultaneously.

The remainder of the paper is structured as follows: In Section 2 we describe the BMC-based approach to formalize LLVM-IR into logic. In Section 3 we discuss the applications currently integrated into FAuST. In Section 4 we present related work. In Section 5 concludes the paper.

2 Formalizing LLVM-IR into Logic using BMC

We use a BMC approach to formalize LLVM-IR into logic: given an imperative, non-concurrent program P and an unrolling bound k, we unroll loops and recursive functions in the program with respect to k and transform the unrolled program into *Static Single Assignment* (SSA) [24] form. The transformations for loop unrolling and to establish SSA form are provided by the LLVM compiler infrastructure.

The resulting program consists of global program variables and a set of functions with one entry function. A function f defines a *Control Flow Graph* (CFG) $\mathsf{CFG}(f) := (V_f, E_f)$ with nodes V_f and edges E_f. The nodes $v \in V_f$ correspond to basic blocks and the edges $e \in E_f$ correspond to possible control flow transfers between basic blocks. Each basic block is a sequence of instructions over program variables and constant values and has a unique label. We write $\mathsf{Pred}(v)$ and $\mathsf{Inst}(v)$ to denote the set of predecessors and the set of instructions of the basic block v.

Suppose P is a program consisting of functions f_i, $0 \le i \le n$, with the entry function f_0 we encode the program into a logic formula,

$$p := \bigwedge_{i=0}^{n} \bigwedge_{b \in V_{f_i}} \left[\bigvee_{b' \in \mathsf{Pred}(b)} e_{b',b} \leftrightarrow \bigwedge_{s \in \mathsf{Inst}(b)} \mathsf{Encode}(s) \right] \wedge e_{f_0},$$

i.e., an instance of the SAT problem. We introduce a logic variable with corresponding data type for each program variable and a constant symbol for each constant value in P. The program is encoded by formalizing the semantics of each function, each basic block, and each instruction. The LLVM-IR instruction set is discussed in detail in the *LLVM Language Reference Manual* [15]. Encoding the individual instruction types is straightforward, i.e,. either the logic of choice provides a corresponding word-level operation or we use an approach similar to *Tseitin's encoding* [25] to lower the operation to a semantically equivalent logic formula using Boolean connectives. We write $\mathsf{Encode}(s)$ to denote the logic formula obtained from encoding instruction s.

In order to encode the control flow of a program, we introduce one Boolean variable for each edge in a $\mathsf{CFG}(f_i)$, $0 \leq i \leq n$, and additional Boolean variables for each function call and return from a function to the callers site. The value of a Boolean variable corresponds to a control flow transfer in the program, i.e., the value is true if the control flow transfers when the program is executed and false otherwise. We write $e_{b',b}$ to denote the Boolean variable which corresponds to the control flow transfer from basic block b' to basic block b and we write e_{f_i} to denote the Boolean variable which corresponds to the entry of function f_i.

Each satisfying assignment of the resulting logic formula p corresponds to a possible assignment to the program variables in P and determines an execution of the program. Figure 2 shows a fragment of an LLVM program and the logic formula in SMT-LIB version 2 [1] format. The program stores the minimum of two given program variables a and b in program variable c.

```
;<label>:1
  %2 = icmp slt i32 %a, %b
  br i1 %2, label %3, label %4

;<label>:3
  br label %5

;<label>:4
  br label %5

;<label>:5
  %c = phi i32 [%a, %3], [%b, %4]
```

```
(set-logic QF_BV)
(declare-fun |%a| () (_ BitVec 32))
(declare-fun |%b| () (_ BitVec 32))
(declare-fun |%c| () (_ BitVec 32))
(declare-fun |%2| () Bool)
(declare-fun |-->%1| () Bool)
(declare-fun |%1-->%3| () Bool)
(declare-fun |%1-->%4| () Bool)
(declare-fun |%3-->%5| () Bool)
(declare-fun |%4-->%5| () Bool)
(assert (=> |-->%1|
            (= |%2| (bvslt |%a| |%b|))))
(assert (=> |-->%1|
  (and (=> |%2| |%1-->%3|)
       (=> (not |%2|) |%1-->%4|))))
(assert (= |%1-->%3| |%3-->%5|))
(assert (= |%1-->%4| |%4-->%5|))
(assert (=> (or |%3-->%5| |%4-->%5|)
  (= |%c| (ite |%3-->%5| |%a| |%b|))))
```

Fig. 2: A fragment of an LLVM program (on the left) and the corresponding logic formula in SMT-LIB version 2 format (on the right).

3 Applications

In this section we outline the applications currently implemented as FAuST tools and list their runtimes for the ANSI-C program TCAS from the *Software-Artifact Infrastructure Repository* (SIR) using specific SMT solvers. However, FAuST supports a large set of different SAT and SMT solvers via API calls and can pass formulae to any interactive SMT solver supporting SMT-LIB version 2 format. We mainly use FAuST to deal with C and C++ programs. However, FAuSTcan be used for other programming language if an LLVM compiler front-end is available which transforms programs into LLVM-IR. In order to use any tool from FAuST, a user has to mark the program's input variables with special function calls `__FAuST_input`. The program variables are then treated as open variables with non-deterministic values when encoded. Moreover, the user has to pass the name of the entry function to be checked to a tool.

3.1 Formal Verification

FAuST provides a standard BMC tool for formal verification which supports *property checking* and *functional equivalence checking*. In the former case the user has to provide local assertions in the program's source code. In the latter case a reference implementation serves as the formal specification. Then, the user has to mark corresponding pairs of program variables in the two implementations to be compared with a special function call `__FAuST_output`. Counterexamples can either be viewed on LLVM-IR or mapped back to the source code passed to the LLVM compiler front-end utilizing LLVM metadata. Optionally, FAuST allows for validation of counterexamples on the real program using LLVM's JIT compiler and execution engine, i.e., a test driver with the values of the counterexample is automatically synthesized, compiled, and executed. Functional equivalence checking of TCAS takes 0.18 seconds using Z3 as SMT solver which is comparable to state-of-the-art BMC tools.

3.2 Automatic Debugging

FAuST provides an extension of the BMC tool for automatic debugging. Given a program that does not conform to its formal specification, the tool computes statements which are potentially faulty. Basically, two strategies are supported: *Model-Based Diagnosis* (MBD) [21,8] and *Error Explanation* (EE) [9]. The MBD strategy computes program variables which when replaced with open variables in the SAT instance correct the program. The EE strategy selects a counterexample and compares the values assigned to the program variables to the values assigned in the most similar execution trace which does not refute the formal specification. Different values indicate potentially faulty statements. In contrast to the Explain [9] tool, FAuST does not use a *Pseudo Boolean* (PB) solver but solves the optimization problem as a binary search over logic variables utilizing incremental SAT. For 41 mutants of TCAS, we computed potentially faulty program locations using both strategies [23]: on average the computation takes 4.37 seconds with strategy MBD and 39.29 seconds with strategy EE using Boolector as SMT solver.

3.3 Test Generation

FAuST provides a mutation-based test generator [22]: a given LLVM-IR program is seeded with artificial faults. The fault seeding is implemented as an LLVM compiler pass. The resulting program, called meta-mutant, contains all faults each guarded with a condition. FAuST instantiates the BMC tool to generate a counterexample for each fault by successively asserting a single guard condition to be true, respectively. From each counterexample a test case is extracted.

Other recent test generators are FShell [11], KLEE [4], and KLOVER [16]. KLEE and KLOVER use a symbolic execution procedure. FShell is a front-end to CBMC and provides a query engine for formulating testing goals. All three tools focus on test case generation subject to traditional coverage criteria. In contrast, our test generator is fault-based, i.e., it imposes constraints that a fault has to be reached, the program state has to be infected, and the infected program state has to propagated to an observable program output. The strength of mutation-based testing criteria was investigated by Offutt and Voes [19]. They outlined that mutation-based criteria subsume several other coverage criteria including *Modified Condition/Decision Coverage* (MC/DC) when a certain set of standard mutations is used.

4 Related Work

Today, BMC is a well established technique for searching bugs in hardware and software. Clarke et al. [6] introduced the *C Bounded Model Checker* (CBMC) which implements BMC considering finite-state systems given as ANSI-C programs. However, CBMC uses its own ANSI-C language parser and relies on a custom-made intermediate representation, called *GOTO programs*. Our BMC core engine is similar to CBMC but uses LLVM-IR as intermediate representation. A program in LLVM-IR is similar to a GOTO program which makes tools based on LLVM-IR neither less efficient nor more abstract than CBMC. However, LLVM-IR is the compiler's intermediate representation which is finally translated into the target code which makes it more suitable for verification and debugging. For instance, it provides the additional capability to detect bugs after certain optimizing transformations are applied to the source code. Also, the LLVM compiler provides a rich tool support for LLVM-IR including a compiler, linker, optimizer, disassembler, and debugger.

Researchers proposed prototype tools based on LLVM [4,17,20,5,16,13,18] for applications like symbolic execution, test generation, and BMC. The most recent BMC tool is LLBMC [18] which focuses entirely on detecting bugs in C/C++ programs either checking for assertions provided by the user or built-in checks, e.g., for overflow detection or memory consistency. However, FAuST is a framework for different applications additionally allowing for test case generation and automatic debugging.

CPAChecker [2] is a configuration software verification platform and follows the idea of having a unified framework for different, formal applications. Programs written in the C and C++ programming language are parsed and transformed into Control Flow Automata (CFA) utilizing Eclipse's CDT plugin. However, the existing procedures implemented for CPAChecker target software verification similar to CBMC.

5 Conclusions

We have presented FAuST, an extensible framework for Formal verification, Automated debugging, and Software Test generation. The framework offers a tool bench for different verification and debugging applications. FAuST utilizes the LLVM compiler infrastructure for analyzing and transforming programs and metaSMT as a generic API interface to different SAT and SMT solvers.

References

1. C. Barrett, A. Stump, and C. Tinelli. The SMT-LIB standard version 2.0, 2010.
2. D. Beyer and M. E. Keremoglu. CPAchecker: A tool for configurable software verification. In *Conference on Computer Aided Verification*, pages 184–190, 2011.
3. A. Biere, A. Cimatti, E. M. 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. C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Symposium on Operating Systems Design and Implementation*, pages 209–224, 2008.
5. V. Chipounov, V. Kuznetsov, and G. Candea. S2E: A platform for in-vivo multi-path analysis of software systems. In *Conference on Architectural Support for Programming Languages and Operating Systems*, pages 265–278, 2011.
6. E. Clarke, D. Kroening, 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.
7. E. M. Clarke, A. Biere, R. Raimi, and Y. Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7–34, 2001.
8. J. de Kleer and B. C. Williams. Diagnosing multiple faults. *Artificial Intelligence*, 32(1):97–130, 1987.
9. A. Groce, S. Chaki, D. Kröning, and O. Strichman. Error explanation with distance metrics. *International Journal on Software Tools for Technology Transfer*, 8(3):229–247, 2006.
10. F. Haedicke, S. Frehse, G. Fey, D. Große, and R. Drechsler. metaSMT: Focus on your application not on solver integration. In *International Workshop on Design and Implementation of Formal Tools and Systems*, pages 22–29, 2011.
11. A. Holzer, C. Schallhart, M. Tautschnig, and H. Veith. FShell: Systematic test case generation for dynamic analysis and measurement. In *Conference on Computer Aided Verification*, pages 209–213, 2008.
12. D. Kröning. Software verification. In A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors, *Handbook of Satisfiability*, pages 505–532. IOS Press, 2009.
13. M. Vujošević-Janičić V. Kuncak. Development and evaluation of LAV: An SMT-based error finding platform. In *International Conference on Verified Software: Theories, Tools and Experiments*, pages 98–113, 2012.
14. C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization*, pages 75–88, 2004.
15. C. Lattner and V. Adve. LLVM language reference manual, 2012. Last visit on 27th of March, 2012.
16. G. Li, I. Ghosh, and S. Rajan. KLOVER: A symbolic execution and automatic test generation tool for C++ programs. In *Conference on Computer Aided Verification*, pages 609–615, 2011.
17. L. McMillan. Lazy annotation for program testing and verification. In *Conference on Computer Aided Verification*, pages 104–118, 2010.
18. F. Merz, S. Falke, and C. Sinz. LLBMC: Bounded model checking of C and C++ programs using a compiler IR. In *International Conference on Verified Software: Theories, Tools and Experiments*, pages 146–161, 2012.
19. J. Offutt and J. M. Voas. Subsumption of condition coverage techniques by mutation testing. Technical Report ISSE-TR-96-01, George Mason University, 1996.
20. D. A. Ramos and D. R. Engler. Practical, low-effort equivalence verification of real code. In *Conference on Computer Aided Verification*, pages 669–685, 2011.
21. R. Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32(1):57–95, 1987.
22. H. Riener, R. Bloem, and G. Fey. Test case generation from mutants using model checking techniques. In *IEEE International Conference on Software Testing, Verification, and Validation Workshops*, pages 388 – 397, 2011.
23. H. Riener and G. Fey. Model-based diagnosis versus error explanation. In *International Conference on Formal Methods and Models for Codesign*, 2012. To Appear.
24. B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *Symposium on Princples of Programming Languages*, pages 12–27, 1988.
25. G. S. Tseitin. On the complexity of derivation in propotional calculus. In *Automation and Reasoning: Classical Papers in Computational Logic 1967-1970*, 1983. Originally published in 1970.