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 multi-threaded 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) [2,6] 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.

More recently, BMC is used in software verification [5,10]: the behavior of a program is extracted from its source code and modeled using logic formulae. Clarke et al. [5] 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 needs adaptions to keep pace with trends in compiler construction.

Today, flexible compilers like the *Low Level Virtual Machine* (LLVM) [12] compiler afford program analysis and verification directly on the compiler's intermediate representation. For instance, researchers proposed prototype tools based on LLVM [3,15,17,4,14,11,16] for applications like symbolic execution, test generation, and BMC. The tools are conceptually similar but are often engineered from scratch. Each tool expresses its application as instances of the SAT problem with respect to some background theory. The instances are then solved using a corresponding *Decision Procedures* (DP), called *Satisfiability Modulo Theories* (SMT) solver.

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

[⋆] This work was supported by the German Research Foundation (DFG, grant no. FE 797/6-1) each FAuST tool is a software program. The output depends on its application. For instance, in fault-based test generation [19] the output is a test suite and in fault localization [20] the output is a set of potentially faulty program locations. The core engine of each tool is a highly customizeable 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 builds a SAT problem from the transformed program with respect to the application. (3) In the *logic layer* the SAT problem is simplified and solved using SAT and SMT solvers.

Figure 1 shows the typical flow of a tool in the FAuST framework. 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 [9] as a generic API interface to different SAT and SMT solvers.

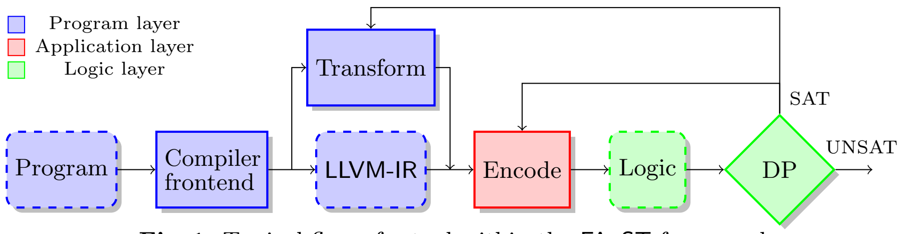

Fig. 1: Typical flow of a tool within the FAuST framework

FAuST is the first tool bench which integrates formal verification, automatic debugging, and test generation into a unified framework. Its 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 application currently integrated into FAuST. Section 4 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) [21] 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, $0 \leq i \leq n$, 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 in detail discussed in the *LLVM Language Reference Manual* [13]. 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* [22] 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). 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`. 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 potentially faulty statements. Basically, two strategies are supported: *Model-Based Diagnosis* (MBD) [18,7] and *Error Explanation* (EE) [8]. 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. The optimization problem is implemented as a binary search over logic variables utilizing incremental SAT. For 41 mutants of TCAS, we computed potentially faulty program locations using both strategies [20]: on average the computation takes 4.37 seconds with strategy MBD and 39.29 seconds with strategy EE.

3.3 Test Generation

FAuST provides a mutation-based test generator [19]: 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 one guard conditions to be true, respectively. From each counterexample a test case is extracted.

4 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. 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.
3. 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.
4. 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.
5. 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.
6. 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.
7. J. de Kleer and B. C. Williams. Diagnosing multiple faults. *Artificial Intelligence*, 32(1):97–130, 1987.
8. 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.
9. 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.
10. 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.
11. 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.
12. C. Lattner. LLVM: An infrastructure for multi-stage optimization. Master's thesis, University of Illinois at Urbana-Champaign, 2002.
13. C. Lattner and V. Adve. LLVM language reference manual, 2012. Last visit on 27th of March, 2012.
14. 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.
15. L. McMillan. Lazy annotation for program testing and verification. In *Conference on Computer Aided Verification*, pages 104–118, 2010.
16. 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.
17. 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.
18. R. Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32(1):57–95, 1987.
19. 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.
20. H. Riener and G. Fey. Model-based diagnosis versus error explanation. In *International Conference on Formal Methods and Models for Codesign*, 2012. In Review.
21. 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.
22. 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.

Part II: Oral Tool Presentation

Informal Plan

The presentation is structured similarly to the paper but elaborates on experiments for specific applications. The experimental results are partially published for fault-based test generation [19] and currently under review for fault localization [20].

Firstly, we start with a brief introduction which gives an overview of BMC and the LLVM compiler infrastructure.

Secondly, we discuss the traditional BMC approach for software programs as implemented, e.g., by CBMC (including loop unrolling) and contrast it with the more recent trend to formalize the Intermediate Representation (IR) of a compiler. In particular, we discuss the formalization of LLVM's IR into logic by example.

Thirdly, we present the architecture of our framework FAuST and its main features, that are, (1) a flexible structure 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. We comment on different applications currently integrated into the FAuST framework.

Next, we show a case study for two selected applications: fault-based test generation (partly published in [19]) and fault localization [20] (in review). For the presentation, we use the ANSI-C program TCAS from the *Software-Artifact Infrastructure Repository* (SIR).

Lastly, the presentation is concluded with a summary and outline.

Structure of the Presentation

1. Introduction
   1.1. Bounded Model Checking (BMC): An Overview
   1.2. LLVM and Recent Trends in Compiler Construction
2. Preliminaries
   2.1. The CBMC Approach
   2.2. ANSI-C → Logic
   2.3. Loop Unrolling
3. Formalizing LLVM-IR
   3.1. LLVM-IR → Logic
   3.2. Example
4. FAuST
   4.1. Architecture of FAuST
   4.2. Main Features
   4.3. Applications
5. Case Study and Experimental Results
   5.1. Benchmarks
   5.2. Fault-Based Test Generation
   5.3. Fault Localization
6. Summary and Outline