Test Case Generation from Mutants using Model Checking Techniques

Heinz Riener* Roderick Bloem† Görschwin Fey*

*Institute of Computer Science
University of Bremen, Germany
{hriener, fey}@informatik.uni-bremen.de

†Institute for Applied Information Processing and Communications
Graz University of Technology, Austria
rbloem@iaik.tugraz.at

***Abstract*—Mutation testing is a powerful testing technique: a program is seeded with artificial faults and tested. Undetected faults can be used to improve the test bench. The problem of automatically generating test cases from undetected faults is typically not addressed by existing mutation testing systems. We propose a symbolic procedure, namely SymBMC, for the generation of test cases from a given program using Bounded Model Checking (BMC) techniques. The SymBMC procedure determines a test bench, that detects all seeded faults affecting the semantics of the program, with respect to a given unrolling bound. We have built a prototype tool that uses a Satisfiability Modulo Theories (SMT) solver to generate test cases and we show initial results for ANSI-C benchmark programs.**

***Index Terms*—Mutation testing, Test case generation, Satisfiability modulo theories, Bounded model checking.**

I. Introduction

Mutation testing [1], [2] is a powerful testing technique based on the idea of making changes to a syntactic description of a computing task and deriving test cases from these changes. The changes mimic mistakes programmers or designers make during the description of the computing task.

We focus on program-based mutation testing: the source code of a program, written in a high-level programming language, is seeded with artificial faults and then systematically executed on each test case of a test bench. Undetected faults indicate holes in the test bench and can be used to improve it.

Mutation testing provides a fault-based *test criterion*, called *mutation adequacy*, i.e., a rule imposing *test requirements* on the test bench that "good" test cases should examine. Mutation adequacy is among the best test criteria. Its effectiveness has been shown by analytic comparison to other criteria [3], [4], [5] and is additionally supported by numerous empirical studies [6], [7], [8], [9].

The tasks of a *mutation testing system* or *tool* are manifold. The system supports the tester in the creation of faulty versions of a given program (called *mutants*), by duplicating the program and introducing syntactic changes into the duplication according to a fixed fault model, the management and the automated execution of test cases on the individual mutants, and in assessing the fault finding abilities of a test bench (e.g. by calculating the ratio of the number of detected faults to the number of seeded faults, called *mutation score*). Mutation testing systems, however, typically do not address the problem of detecting seeded faults by automated test case creation [10]. Thus, in contrast to push-button verification approaches (e.g. *model checking* [11]) mutation testing remains a labor-intensive, manual process. Research in the mid 1980s [12] and early 1990s [13], [14] in mutation testing already proposed approaches for the automated generation of test cases based on solving a *constraint satisfaction problem* by representing conditions under which mutants will be detected as algebraic constraints. These approaches are sound but incomplete, allowing the construction of "false negative" test cases that do not lead to the detection of seeded faults. Moreover, they target the programming language Fortran-77 omitting pointer aliasing and have never been shown to scale to programs larger than a few lines of code. The general problem of answering whether test data exists that eventually reaches a specific program location is undecidable [15].

The early test case generation approaches precede the development in *Satisfiability Modulo Theories* (SMT) [16] solvers, which implement efficient decision procedures for large instances of constraint satisfaction problems according to some specific background theories [17]. More recently, researchers focus on automated verification approaches dedicated to high-level programming languages like Java, C, and C++ [18], [19], [20], [21] improving applicability and scalability of software verification.

We propose a symbolic procedure, namely *SymBMC*, to generate test data, that achieves high mutation scores. The input of the SymBMC procedure is a program and a set of mutants of the program. The procedure examines the given program and its mutants on the same input data and attempts to find witnesses for their differences, that are, executions resulting in different externally observable outputs for the program and the mutants. If a failing execution is found, SymBMC saves the corresponding input data as *effective* test data that led to the detection of a seeded fault.

The SymBMC procedure uses a *Bounded Model Checking* (BMC) [22] approach: loops in the original and mutated programs are unrolled for a fixed number k of loop iterations. The unrolled original and mutated programs are then encoded into quantifier-free logic formulae f_k and f'_k, respectively, over the same input variables. SymBMC then generates a *propagation condition* g, i.e., a logic formula ensuring that a seeded fault affects the output of the mutant. Finally, the formula $f_k \wedge f'_k$ conjoined with the propagation condition g is

This work was supported in part by the German Research Foundation (DFG, grant no. FE 797/6-1) and the European Union (project DIAMOND, FP7-2009-IST-4-248613).

checked for satisfiability. A satisfying assignment if one exists serves as effective test input data that results in a different externally observable output on the original program and the mutant if executed.

We have evaluated our approach with a prototype implementation that uses the theory of bit-vectors of arbitrary size supported by standard SMT solvers to encode the logic formulae. The prototype implementation takes as input a source program in the programming language ANSI-C and transforms it into a RISC-like intermediate representation, *Low Level Virtual Machine* (LLVM) [23]. The prototype implementation then generates a *meta-mutant* [24], i.e., a program containing a set of faults and additional control logic to enable and disable the individual faults for testing purpose.

Our approach is sound and complete with respect to the unrolled model: the SymBMC procedure precisely detects all seeded faults that propagate in the unrolled model to an externally observable output and reports these fault to the user. In contrast to techniques entirely based on testing, we proof for all faults not reported to the user, that there is no execution in the unrolled model resulting in a different externally observable output. We have evaluated SymBMC on a small set of benchmark programs and report initial results.

The remainder of the paper is organized as follows. In Section II, we discuss related work in mutation testing and bounded model checking. In Section III, we introduce mutation testing, discuss a subset of the LLVM intermediate representation, and present the fault model used. Section IV gives our symbolic procedure for test input data generation. In Section V, we show initial empirical results. Section VI concludes the paper.

II. Related Work

Offutt et al. proposed *constraint-based test data generation* [13] or simply *Constraint-Based Testing* (CBT) [14] as an approach for the automated generation of inputs that distinguish a set of mutants from its original program when executed. Their approach uses three conditions (reachability, necessity, and sufficiency) per mutant. Each condition is described as a constraint over symbolic input variables that conjoined form a *Constraint Satisfaction Problem* (CSP). An assignment satisfying the CSP (if one exists) is an effective test case that distinguishes the mutant from the original program. Constraint solving was implemented as a *domain reduction procedure* that successively finds values for the variables, substitutes them into the CSP, and backtracks when the resulting CSP becomes inconsistent. Domain reduction suffers from shortcomings including handling arrays, loops, and nested expressions. Offutt et al. proposed a refinement of their procedure, called *dynamic domain reduction* [25], [26], that attempts to overcome previous shortcomings. Moreover, Offutt and Pan [27] used CBT as a decision procedure to decide functional non-equivalence of a program and one of its mutants.

CBT, however, was proposed as an approximation technique that does not exploit the sufficiency conditions, i.e., a test case generated by CBT may not propagate to an externally observable output. Thus, CBT serves as an over-approximation technique in case of test case generation and as an under-approximation technique in case of equivalence checking.

Our SymBMC procedure statically encodes the program and its mutants into one SMT formula. In case of a finite number of execution paths in the program, the SMT formula contains the complete path information corresponding to the encoding of all three conditions used in CBT for each mutant. Thus, we can precisely decide the test case generation problem and the equivalent mutant problem for programs with a finite number of paths. In case of input-dependent loops, the path space of a program is unbounded resulting in an infinite number of paths. We approximate the path space then by unrolling loops up to a given maximum bound, which under-approximates the solutions of the test case generation problem. We may miss seeded faults that affect an externally observable output in a program unrolled beyond the given maximum bound.

Ammann et al. [28] suggested specification-based mutation to obtain test cases with a model checker by turning a counterexample into a test case. Okun et al. [29] presented two approaches to obtain counterexamples that are guaranteed to propagate to an externally observable output. Although they target test case generation based on a specification, one of their approaches, namely *state machine duplication*, is similar to our work. Their approach duplicates a given state machine, introduces a fault into the duplicate, and adds a temporal logic formula given in *Computation Tree Logic* (CTL) [30] asserting equal externally observable outputs for the original state machine and its faulty duplicate. A model checker is then used to obtain a counterexample that violates the CTL formula, which can be automatically turned into a test case [31]. State machine duplication doubles the number of states for each considered mutation. We use a meta-mutant construction that encodes a set of mutants with respect to a fault model. The size of the meta-mutant grows linear with the size of the original program and linear with the number of mutants.

There are two potentially bad influences on the performance of test case generation using a model checker: on the one hand, equivalent mutants consume time for encoding and model checking but result in no test cases. On the other hand, many different mutants lead to identical or subsumed test cases. Fraser and Wotawa [32] addressed these two problems considering mutants of transition systems, where a mutant differs from the original transition system in exactly one transition. As an optimization of the state machine duplication approach, they proposed the *characteristic property* of a mutant, i.e., a temporal logic formula φ that guarantees non-equivalence of the original transition graph M and the mutant M' if $M \not\models \varphi$. The logic formula φ expresses that the mutated transition, firstly, corresponds to behavior already allowed in the original transition graph and, secondly, does not restrict the original transition graph.

Our SymBMC procedure alleviates the issues noted in [32]. The prototype implementation encodes all mutants with respect to a given fault model into one SMT formula and lets the SMT solver choose which individual fault to enable during test case generation. From a satisfying assignment, SymBMC extracts the detected fault and constrains the SMT formula such that the next satisfying assignment will lead to the detection of another fault. The equivalent mutants remain undetected. Eventually the SMT formula becomes unsatisfiable. The proof of the unsatisfiability of the formula corresponds to a proof of equivalence of all remaining mutants in the unrolled model and the unrolled original program.

Bounded Model Checking (BMC) [22] was originally proposed as a symbolic model checking approach complementary to model checking based on *Binary Decision Diagrams* (BDD) [33]. Given a model M of a finite-state system and a property in *Linear Time Logic* (LTL) [34], BMC searches for a counterexample of finite-length k in the model. Biere et al. [22] suggested encoding the model and the property into one Boolean formula, which is satisfiable if and only if there is a finite counterexample in the model that refutes the property. The Boolean formula was solved with a SAT solver. The parameter k is often called *unrolling bound* or *counterexample length*. Clarke et al. [19] proposed an efficient prototype implementation of BMC for a subset of the ANSI-C programming language, known as CBMC. Armando et al. [35] used an SMT rather than a SAT solver to encode BMC, which improves scalability in case of complex arithmetic or array manipulations and additionally produces more compact formulae. More recently, Sinz et al. [21] proposed a formalization of BMC for the LLVM intermediate representation, termed *Low-Level Bounded Model Checking*. They especially focused on the memory model and the translation of pointer manipulations into logic constraints. Moreover, Sinz et. al [21] argued that the syntax and semantics of LLVM are simpler and allow an easier formalization of the verification problem than high-level programming languages like C or C++. Our encoding of the LLVM intermediate representation into logic formulae is similar to the encoding used in [21]. We use BMC to generate test cases that are guaranteed to propagate to an externally observable output within a given unrolling bound k. Mutation and BMC are applied to the LLVM intermediate representation.

III. Preliminaries

In the following section, we describe the mutation testing process in detail (Section III-A), formalize the syntax and semantics of a subset of the LLVM intermediate language (Section III-B), and define the fault model used (Section III-C).

A. Mutation Testing Process

Program-based mutation testing forms a three step test process: given a program and a test bench. (1) The program is seeded with artificial faults according to a fixed fault model. Each seeded fault is kept in an individual copy of the original program source, called mutant. (2) Each test case from the test bench is then executed on the original program and on its mutants. We stick to the common mutation testing terminology: initially all mutants are *alive* and a mutant that results in a different externally observable output is called *killed*. Some syntactic changes may not alter the semantics of the original program. The corresponding mutants are called *equivalent*. (3) The mutation score quantifies the fault finding abilities of the test bench by calculating the ratio of mutants killed by the test bench to the number of seeded faults, i.e., a real value in the interval $[0, 1]$, which is interpreted as a percentage value. The test bench is then improved by adding additional (or replacing existing) test data. The second and third steps are repeated until the mutation score exceeds a predefined *fault sensitivity threshold*.

A fault sensitivity threshold of 100% is desirable. Equivalent mutants, however, cause a systematic underestimation of the mutation score. Equivalent mutant detection, as an optional step, targets the correction of the mutation score. The number of seeded faults is reduced by the number of detected equivalent mutants. An equivalent mutant adds no information to mutation testing and, thus, equivalent mutants are discarded from the mutation testing processes if detected.

Mutation testing systems implement the mutation testing process. Typically, mutation testing systems automate the first step, the second step, and the calculation of the mutation score from the third step. The task of equivalent mutant detection and the task of test case improvement are left to the test engineer. Unfortunately, as a direct consequence of Rice's theorem [36] automated equivalence checking of arbitrary programs is undecidable. Moreover, Budd and Angluin [37] discussed the close relation between test case generation and equivalence detection. They showed that in general neither an effective procedure for the generation of a mutation adequate test bench, nor an effective procedure for the detection of equivalent mutants exists. Although automated test case generation is a hard problem, we propose a symbolic procedure based on bounded model checking that uses an SMT solver. Our procedure is sound and complete in case of a finite number of program paths and otherwise approximates mutation adequacy by unrolling the program up to given maximum bound.

B. LLVM Intermediate Representation

Low Level Virtual Machine (LLVM) is a strongly typed, RISC-like assembly language which comes with two broad characteristics: (1) The language allows the usage of an unlimited number of registers and (2) LLVM programs can be translated into *Static Single Assignment* (SSA) form, where each register is assigned only once. These characteristics are desirable because they simplify the transformation of the operational semantics of LLVM into logic constraints.

In the following section, we describe the syntax and semantics of a subset of the LLVM instruction set. For the sake of simplicity, we omit memory and pointer instructions, inline function calls but do not handle recursions, and assume that each variable is of integral type. The omission of memory and pointer instructions and the handling of recursions are restrictions according to our prototype implementation. A formalization of a memory model for the LLVM language can be obtained from [21]. The other restrictions are only related to the descriptions.

A program in the resulting simplified LLVM language consists of global variables and, due to function inlining, of a single function. The function defines a graph with basic blocks as nodes and branches as edges, called *control flow* *graph*. A basic block is a sequence of instructions with a unique label, where the instructions are guaranteed to be executed in consecutive order. The last instruction of each basic block defines a set of successor basic blocks. We call the relation between a basic block and its successors *branches*. The first basic block of a function is the *initial basic block*, which corresponds to the root node of the control flow graph of the program. Moreover, we assume that two consecutive sequences of registers $i_1, i_2, \ldots, i_n$ and $o_1, o_2, \ldots, o_m$ are given, which serve as program inputs and outputs, respectively.

For LLVM programs, the *state* of a program is a valuation of the register set and the *program counter*, i.e., a special register that denotes the next instruction of the program to be executed. Initially the value of the program counter is the address of the first instruction in the initial basic block.

We assume that instructions are either load instructions, binary operator instructions, branching instructions, or phi instructions. The semantics of an LLVM program can then be defined in terms of the manipulations of the register set and the program counter of the individual instructions.

Suppose r_{dest} denotes a register, v, v_{op_1}, v_{op_2}, $\ldots v_{op_n}$ denote values that are either addresses of registers or constants, and l, l_{true}, l_{false} denote labels. We define the syntax and semantics of these instructions as follows:

- The load instruction is of the form

$$r_{dest} \texttt{ = } \textbf{\texttt{load}} \ v.$$

  The register r_{dest} is assigned the value v when the load instruction is executed.
- The branching instruction is either of conditional

$$\textbf{\texttt{br}} \ v\texttt{,} \ \textbf{\texttt{label}} \ l_{true}\texttt{,} \ \textbf{\texttt{label}} \ l_{false}$$

  or unconditional form

$$\textbf{\texttt{br label}} \ l.$$

  The conditional branching instruction sets the value of the program counter to the address of l_{true} when v equals 1 and otherwise to the address of l_{false}. The unconditional branching instruction sets the program counter to the address of label l when executed.
- The binary operator instruction is of the form

$$r_{dest} \texttt{ = binop } v_{op_1}\texttt{,} \ v_{op_2},$$

  where `binop` is a mnemonic denoting one of a fixed set of binary operations. The binary operations include arithmetic, relational, and bitwise operations, which are described in detail in the *LLVM Language Reference Manual* [38]. The binary operator instruction assigns the register r_{dest} the value of the function $f_{binop}(v_{op_1}, v_{op_2})$ when executed, where f_{binop} is a binary function representing the semantics of the binary operation denoted by `binop`. For the sake of simplicity, our definition of the binary operator instruction encompasses binary operator, bitwise binary operator, and comparison instructions from the LLVM language. We understand all of these instructions as binary operators that can be distinguished by the mnemonic `binop`, respectively.
- The phi instructions enable the transformation of LLVM programs into SSA form because the instructions merge data from different branches. A phi instruction is of the form

$$r_{dest} \texttt{ = phi } \texttt{[}v_{op_1}\texttt{,}l_1\texttt{]}\texttt{,}\texttt{[}v_{op_2}\texttt{,}l_2\texttt{]}\texttt{,}\ldots\texttt{,}\texttt{[}v_{op_n}\texttt{,}l_n\texttt{]}.$$

  We say that the current basic block is the basic block containing the phi instruction. The basic blocks with labels l_1, l_2, …, l_n are direct predecessors of the current basic block. The phi instruction assigns the register r_{dest} the value v_{op_i} if the basic block labeled with l_i, $1 \leq i \leq n$, was immediately executed before the current basic block.

In the remainder of the paper, we use the subset of the LLVM instruction set to discuss our fault model and the test case generation. The subset can naturally be extended to similar instructions from the full LLVM instruction set (e.g., `bitcast`, `switch`, etc.). We do not discuss instructions manipulating pointers.

C. Fault Model

A mutant is a duplication of the original program source containing one syntactic change. For instance, an addition operator is turned into a subtraction operator. The fault model defines a set of mutation operators, that are, rules describing possible syntactic changes to the source code of a program.

We use a fault model similar to the fault models proposed in [39] and [40]: we introduce syntactic changes into arithmetic, relational, and bitwise operators, and inject values into expressions used in load instructions. Our test case generation approach, however, is not tied on this fault model.

We formalize the fault model using *mutation operators* on the level of the LLVM intermediate representation. A mutation operator is a rule that describes how a particular syntactic pattern of LLVM programs is changed. When the mutation operator is applied to a source program, it parses the program source top-down. For each syntactic part of the program that matches the pattern of the mutation operator, the source program is duplicated and the matched part of the source code is replaced by syntactically different source code resulting in new mutants of the program. Thus, a mutation operator defines a mapping from a program to a set of mutants of the program. We use four mutation operators: `AOR`, `ROR`, `BOR`, and `IVI`. Technical details are left to Figure 1. The `ROR` mutation operator, for instance, replaces each occurrence of a relational operator by another relational operator, e.g., the mnemonic for lower than (**`lt`**) against the mnemonics for equality (**`eq`**), inequality (**`ne`**), greater than (**`gt`**), lower than (**`le`**), and greater equal (**`ge`**), resulting in five mutants of the source program.

Our mutation operators are similar to Offutt's set of *expression-selective mutation operators* [39], [41] for Fortran-77 and the mutation operators used in [40]. Some of the operators in [40] are only marginally described, which prevents us from making an in depth comparison. We took the `AOR` and `ROR` mutation operators from [41] and adapted them for the LLVM instruction set. Moreover, we supplemented the `BOR` mutation operator because in [41] no bitwise operators for

The fault model considers four mutation operators: the replacement of arithmetic, relational, and bitwise binary operators, and the injection of values into load instructions.

Suppose $\mathcal{P}$ is the set of programs, $\mathrm{Occ}(T, P)$ is the set of occurrences of the tokens $t \in T$ in the program P, and $P[t/t']$ denotes the replacement of token t by token t' in program P.

Replacement of Arithmetic Operators (AOR): The `AOR` mutation operator is a mapping

$$\begin{aligned} \mathsf{t}_{\mathsf{AOR}} :& \mathcal{P} \rightarrow 2^{\mathcal{P}}, \\ & P \mapsto \{P[t/t'] \mid t \in \mathrm{Occ}(\mathsf{AOp}, P), t' \in \mathsf{AOp} \setminus t\}, \end{aligned}$$

that mimics a mistake in an arithmetic binary operator, where AOp := {`add`, `sub`, `mul`, `div`, `mod`}.

Replacement of Relational Operators (ROR): The `ROR` mutation operator is a mapping

$$\begin{aligned} \mathsf{t}_{\mathsf{ROR}} :& \mathcal{P} \rightarrow 2^{\mathcal{P}}, \\ & P \mapsto \{P[t/t'] \mid t \in \mathrm{Occ}(\mathsf{ROp}, P), t' \in \mathsf{ROp} \setminus t\}, \end{aligned}$$

that mimics a mistake in a relational binary operator, where ROp := {`eq`, `ne`, `gt`, `ge`, `lt`, `le`}.

Replacement of Bitwise Operators (BOR): The `BOR` mutation operator is a mapping

$$\begin{aligned} \mathsf{t}_{\mathsf{BOR}} :& \mathcal{P} \rightarrow 2^{\mathcal{P}}, \\ & P \mapsto \{P[t/t'] \mid t \in \mathrm{Occ}(\mathsf{BOp}, P), t' \in \mathsf{BOp} \setminus t\}, \end{aligned}$$

that mimics a mistake in a bitwise binary operator, where BOp := {`and`, `or`, `xor`, `shl`, `lshr`, `ashr`}.

Integral Value Injection (IVI): The `IVI` mutation operator is a mapping

$$\begin{aligned} \mathsf{t}_{\mathsf{IVI}} :& \mathcal{P} \rightarrow 2^{\mathcal{P}}, \\ & P \mapsto \{P[t/t+1], P[t/t-1], P[t/0], t \in \mathrm{Occ}(\mathsf{Value} \setminus t)\}, \end{aligned}$$

that mimics off-by-one faults and the injection of zero values, where Value denotes the set of values. Constant values are simply replaced. In order to encode the off-by-one faults for values representing registers, the syntax of LLVM requires the insertion of additional add and sub binary operator instructions.

Fig. 1. The fault model considered in our work consisting of four mutation operators (`AOR`, `ROR`, `BOR`, and `IVI`).

Fortran-77 are described. The `IVI` mutation operator is similar to Offutt's `UOI` mutation operator and covers all mutations from the category "replace numerical constants" in [40].

IV. Test Case Generation

In the following section, we present the symbolic procedure SymBMC. We start with the discussion of a simplified version SimplifiedBMC. For a given program and one of its mutants, SimplifiedBMC attempts to generate an effective test case (if one exists) that distinguishes the behavior of the mutant and the original program, i.e., a test case that results in a different externally observable output when executed on the original program and its mutant. We give a high-level overview of SimplifiedBMC (Section IV-A), then we present the unrolling of the program (Section IV-B), and the encoding of the program into an SMT formula (Section IV-C) in detail. Next, we show the meta-mutant construction (Section IV-D) that statically encodes a set of mutants into one "meta" program. Finally, we present the symbolic procedure SymBMC (Section IV-E) that generalizes SimplifiedBMC and iteratively generates test cases for the set of mutants represented by a meta-mutant.

A. Simplified Symbolic Procedure

Procedure 1 gives SimplifiedBMC in pseudo code. First, for a given program P, one of its mutants P', and a maximum unrolling bound k, SimplifiedBMC generates a model of the original program P and the mutant P' by unrolling the programs with respect to the maximum unrolling bound k and consecutively translating them into SSA form. The models of the unrolled programs are then encoded into quantifier-free bit-vector formulae f_k and f'_k. We describe the unrolling, the SSA-transformation, and the encoding tasks by the pseudo code function `Encode` in lines 2 and 3. The function `Encode` encodes a program P with respect to the maximum unrolling bound k and returns the three parameters (i_j^P), (o_l^P), and f_k, where f_k is a logic formula corresponding to the SSA-transformed, k-times unrolled model of the program P, and (i_j^P), (o_l^P) are finite sequences of bit-vector variables used in f_k denoting the encoded input and output registers of P.

Second, SimplifiedBMC creates the propagation condition g in line 4 by asserting that there is at least one pair of different outputs $o_j^P \neq o_j^{P'}$, $1 \leq j \leq m$, under the assumption of equal inputs $i_l^P = i_l^{P'}$, $1 \leq l \leq n$, where m and n denote the length of the sequences, respectively. The models of the unrolled programs, represented by the logic formulae f_k and f'_k, are then conjoined with the propagation condition g in line 5 and solved by handing the resulting formula $f_k \wedge f'_k \wedge g$ to an SMT solver, denoted by the pseudo code function `Solve` in line 6. If the SMT solver finds a satisfying assignment, we extract input and output data from the satisfying assignment, denoted by the pseudo code function `ExtractTestCase` in line 7, and return the data as a test case. We save the test case in a database as an effective test case that kills the mutant P'. Otherwise, if the SMT solver concludes the unsatisfiability of the logic formula, the models of the unrolled original program and unrolled mutant are equivalent with respect to the maximum unrolling bound k (line 9).

B. Unrolling the Program

Suppose k is a maximum unrolling bound. The SymBMC procedure first unrolls the given program. The control flow graph of the program is step-wise traversed from the root node (corresponding to the initial basic block of the entry function) by following the outgoing edges in a breadth-first manner. If a node is revisited during traversing, SymBMC duplicates the corresponding basic block, directs the incoming edge to the duplicate, and continues traversing from the new node. The maximum unrolling bound k prevents unlimited unrolling in case of input-dependent loop conditions. Each basic block is duplicated at most k times. Currently, the symbolic procedure quits if the unrolling bound k is not sufficient to unroll all

Procedure 1: Test Case Generation for a Mutant

Input : a program P, a mutant of the program P', and a maximum unrolling bound k

Output: a test case that kills the mutant P' with respect to the maximum unrolling bound k if P and P' are non-equivalent and EQUIVALENT otherwise

```
1  begin
2      $((i_j^P), (o_l^P), f_k)$ := Encode ($P$, $k$);
3      $((i_j^{P'}), (o_l^{P'}), f'_k)$ := Encode ($P'$, $k$);
4      $g := \bigwedge_{j=1}^{n}(i_j^P = i_j^{P'}) \wedge \bigvee_{l=1}^{m}(o_l^P \neq o_l^{P'})$;
5      $s := f_k \wedge f'_k \wedge g$;
6      if Solve(s) = SATISFIABLE then
7          return ExtractTestCase (s);
8      else
9          return EQUIVALENT;
10     end
11 end
```

loops entirely, i.e., a loop condition is not satisfied after k-times unrolling the program. During traversing the control flow graph, SymBMC inlines function calls.

C. Encoding the Program

The unrolled programs are translated into SSA form to ease the logical encoding, which is a standard task provided by the LLVM compiler infrastructure. We encode programs into logic formulae over the theory of quantifier-free bit-vectors of arbitrary size, where a symbolic variable is either a *bit-vector variable* or a *binary decision variable*. A bit-vector variable consists of a finite sequence of bits, whereas a binary decision variable denotes a single bit. Bit-vector variables can be used to encode arbitrary information of finite length. SMT solvers supporting the theory of bit-vectors of arbitrary size provide several word-level operations (e.g. addition, subtraction, if-then-else, etc.) for manipulating bit-vector variables.

We introduce a bit-vector variable bv_x of corresponding size for each program variable x and additionally one binary decision variable bb_l for each basic block of the program, where l denotes the label of the basic block. The additional binary decision variables are used to encode the transfer of control flow of a program. The variable bb_l is 1 if and only if the basic block labeled with l has been executed.

Suppose $bv_{r_{dest}}, bv_v, bv_{v_{op_1}}, bv_{v_{op_2}}, bv_{c_1}, bv_{c_2}, \ldots, bv_{c_n}$ are bit-vector variables denoting the register r_{dest} and the values v, v_{op_1}, v_{op_2}, c_1, c_2, $\ldots$, c_n, respectively. The encoding of the individual instruction types is straightforward.

- The load instruction

  $$r_{dest} \;=\; \texttt{load}\; v$$

  in a basic block labeled with l is mapped to an implication

  $$bb_l \;\rightarrow\; (bv_{r_{dest}} \;=\; bv_v).$$

- The binary operator instruction

  $$r_{dest} \;=\; \texttt{binop}\; v_{op_1},\; v_{op_2}$$

  in a basic block labeled with l is encoded by mapping the mnemonic `binop` of the instruction to its SMT counterpart bv_{binop} resulting in an implication

  $$bb_l \;\rightarrow\; (bv_{r_{dest}} \;=\; bv_{binop}(bv_{v_{op_1}},\; bv_{v_{op_2}})).$$

- The branching instruction

  $$\texttt{br}\; v\; \texttt{label}\; l_{true},\; \texttt{label}\; l_{false}$$

  in a basic block labeled with l is encoded into a conjunction of implications

  $$((bb_l \wedge v) \;\rightarrow\; bb_{l_{true}}) \;\wedge\; ((bb_l \wedge \neg v) \;\rightarrow\; bb_{l_{false}}).$$

- The phi instruction

  $$r_{dest} \;=\; \texttt{phi}\; [v_{op_1}, l_1], [v_{op_2}, l_2], \ldots, [v_{op_n}, l_n]$$

  in a basic block labeled with l is encoded into a sequence of nested logic `ite` (if-then-else)-operations

  $$bb_l \;\rightarrow\; (\texttt{ite}(bv_{c_1}, v_{op_1},\; \texttt{ite}(bv_{c_2}, v_{op_2}, \ldots \; \texttt{ite}(bv_{c_{n-1}}, v_{op_{n-1}}, v_{op_n})\ldots))),$$

  where the value c_i, $1 \leq i \leq n$, denotes the logic condition under which the control flow transfers from the basic block labeled with l_i to the basic block labeled with l. The value c_i is calculated from the branching instructions of the basic block labeled with l_i.

Finally, we constrain the binary decision variable corresponding to the initial basic block of the program true, denoting that each execution of the program must enter the initial basic block of the program. The resulting logic formula is satisfiable if and only if there is an execution path from the initial basic block to the program's exit and the unrolling bound k is sufficient to unroll the program.

D. Meta-Mutant Construction

The SimplifiedBMC procedure attempts to generate a test case for a program and one of its mutants, whereas the SymBMC procedure generalizes the approach to generate test cases from a set of mutants. We create one "meta" program containing a set of mutants, called meta-mutant [24], rather than generating several independent mutants. The meta-mutant serves as an effective data structure to reason about a set of mutants.

Given a program P and a list of faults to be seeded into the program according to a fixed fault model, first, each fault gets a unique id, e.g. by consecutively numbering the faults. We start numbering at 1 and reserve the id 0 for the original program behavior. The faults are systematically seeded into the program. For each fault, the basic block that is seeded with the fault, is duplicated and then mutated. Thus, the meta-mutant contains the original and the new, mutated basic block.

We add a global variable `FAULT_ID` and additional control logic per fault to the program. The control logic enables one mutated basic block at a time if the value of `FAULT_ID` is equal to the id of the fault and the original basic block otherwise.

Figure 2 gives a short example program that conforms to the subset of LLVM described in Section III-B. The example

```
l1:  r1 = load i1
     r2 = lt i2, i1
     br r2 label l2, label l3
l2:  r1 = load i2
     br label l3
l3:  o1 = load r1
```

Fig. 2. An example program conforming to the subset of LLVM described in Section III-B, that calculates the minimum of two given variables `i1` and `i2` and saves the resulting output in the variable `o1`.

```
chk:  r1 = load FAULT_ID
      r2 = eq r1, 1
      br r2 label mut1, label l1
mut1: r3 = load i1
      r4 = le i1, i2
      br r4, label l2, label l3
l1:   r3 = load i1
      r5 = lt i1, i2
      br r5 label l2, label l3
l2:   r3 = load i2
      br label l3
l3:   o1 = load r3
```

Fig. 3. A meta-mutant of the example program given in Figure 2 with a single fault: the mnemonic of the binary operator instruction **`lt`** in basic block `l1` is replace with mnemonic **`le`**.

program calculates the minimum of two given program inputs and returns the result as program output. We denote the program inputs by the input registers `i1`, `i2`, and the program output by the output register `o1`. For the sake of simplicity, the example program is not in SSA form (the register `r1` is assigned twice).

Suppose a potential fault in the basic block `l1` with id 1, where the mnemonic of the binary operator instruction **`lt`** (lower than) is turned into the mnemonic **`le`** (lower equal). In order to construct the meta-mutant, the basic block `l1` is duplicated and mutated. The meta-mutant is shown in Figure 3. The basic block `mut1` corresponds to the faulty duplicate of basic block `l1` and the basic block `chk` serves as control logic, which, based on the value of the variable `FAULT_ID`, either enables the mutant in basic block `mut1` or the original program behavior in basic block `l1`.

Figure 3 contains only a single mutant, the meta-mutant construction can be extended to an arbitrary number of faults by adding more basic blocks each containing one fault (similar to the basic block `mut1`) and control logic (similar to the basic block `chk`). However, we use a more subtle construction using the switch instruction contained in the full LLVM instruction set. The size of the resulting meta-mutant is linear in the size of the original program.

E. Symbolic Procedure

Procedure 2 gives the symbolic procedure SymBMC, which is built similar as SimplifiedBMC. The inputs of SymBMC are a meta-mutant M of a program and a maximum unrolling bound k. The output of SymBMC is a set Ψ of test cases. Initially Ψ is empty (line 2).

The SymBMC procedure, first, unrolls the meta-mutant M twice, shown in line 3 and 4. The distinct bit-vector variables id_1 and id_2 represent the global variable `FAULT_ID` in the logic formulae f_k and f'_k that encode the unrolled programs, respectively. The first time the meta-mutant is unrolled, SymBMC constrains id_1 to the value 0 in line 3, and, thus, disables every fault in the first unrolled model. The first unrolled model behaves as the original program. The second time the meta-mutant is unrolled, id_2 remains unconstrained in line 4.

The logic formula s encodes the test case generation problem. The construction of s shown in line 5 and 6 is equal to the construction used in SimplifiedBMC. The resulting logic formula is then handed to an SMT solver. A satisfying assignment corresponds to an execution path through the program that kills a particular mutant. From the satisfying assignment, we extract the values of the symbolic variables denoting the program inputs and the value of the symbolic variable denoting the global program variable `FAULT_ID` to identify which mutant has been killed. The extraction task is represented by the pseudo code functions `ExtractTestCase` and `ExtractFaultID` in the lines 8 and 9, respectively. The symbolic variable id_2 is then constrained such that another satisfying assignment of the logic formula must enable a mutant distinguished from the mutants already killed. We solve and incrementally constrain id_2 until the logic formula becomes unsatisfiable and no remaining mutant can be killed. The unsatisfiability proof of the last call to the SMT solver proofs all remaining mutants functionally equivalent to the original program with respect to the unrolled model, i.e., none of the mutants result in a different externally observable output with respect to the maximum unrolling bound k.

Procedure 2: Test Case Generation for the Meta-Mutant

Input : a meta-mutant M and a maximum unrolling bound k

Output: a list of test cases Ψ that guaranteed kills all non-equivalent mutants of the meta-mutant M with respect to the maximum unrolling bound k

```
1  begin
2      $\Psi := \emptyset$;
3      $((i_j^P), (o_l^P), f_k) :=$ Encode$(M,k) \wedge (id_1 = 0)$;
4      $((i_j^{P'}), (o_l^{P'}), f'_k) :=$ Encode$(M,k)$;
5      $g := \bigwedge_{j=1}^{n}(i_j^P = i_j^{P'}) \wedge \bigvee_{l=1}^{m}(o_l^P \neq o_l^{P'})$;
6      $s := f_k \wedge f'_k \wedge g$;
7      while Solve(s) = SATISFIABLE do
8          $\Psi = \Psi \cup$ ExtractTestCase$(s)$;
9          $s = s \wedge (id_2 \neq$ ExtractFaultID$(s))$;
10     end
11     return $\Psi$;
12 end
```

TABLE I
RESULTS OF THE TEST CASE GENERATION WITH BOOLECTOR

Name	Instr. (Program)	Instr. (Meta-Mutant)	Faults	Test Cases	Time [s]
min	24	71	17	16	0.48
isl	20	80	19	18	0.14
fmin3	40	137	33	23	7.49
fmin5	58	203	49	37	34.38
fmin10	103	368	89	72	213.65
mid	52	194	46	43	6.82
tri	116	819	206	196	246.80

V. EXPERIMENTAL RESULTS

We present initial results for the SymBMC procedure on a case study of seven, small ANSI-C benchmarks programs. The programs are similar to benchmark programs considered by Offutt et al. [27] but are rewritten to ANSI-C. Moreover, we have not tested programs that use floating point types because the mapping from floating point types is not directly supported by SMT solvers. All experiments were conducted on a PC with an AMD Athlon™ 64 X2 Dual Core Processor 6000+, which has 2 cores with 3 GHz each and 4.12 GB RAM.

Table I reports the results of the test case generation on the seven benchmark programs. The columns from left to right name the individual benchmark program, give the number of LLVM instructions of the original program, the number of LLVM instructions of the unrolled meta-mutant, the number of seeded faults according to our fault model, the number of generated test cases using SymBMC, and the accumulated time required for incrementally solving the individual constraint satisfaction problems in seconds.

For solving the quantifier-free bit-vector formulae, we have used the SMT solver Boolector[1]. We do not report the time needed to encode the programs into bit-vector formulae in the table, which on average takes less than one second for the benchmark programs. The difference between the number of seeded faults and the number of test cases generated equals the number of equivalent mutants. For instance, 10 of 206 seeded faults in the meta-mutant of the benchmark program `tri` do not affect the program semantics.

Note, that not every LLVM instruction is mutated, e.g. conditional and unconditional branching instructions are not affected by our fault model. Moreover, the full LLVM instruction set contains allocation and bitcast instructions, which we do not mutate. Thus, the number of seeded faults is lower than the number of instructions for most of the benchmark programs. The `tri` benchmark program has several binary operator instructions, which significantly increases the number of seeded faults according to our fault model.

Figure 4 and Figure 5 show the generation of the individual test cases for the benchmark program `tri` in detail. Figure 4 gives the local time required for the generation of the individual test cases. On the t-axis, we show the time t in seconds separated into continuous intervals of one second and on the d_F-axis, we count the number of individual logic formulae

[1]Boolector 1.4, 64-bit [42]: http://fmv.jku.at/boolector/

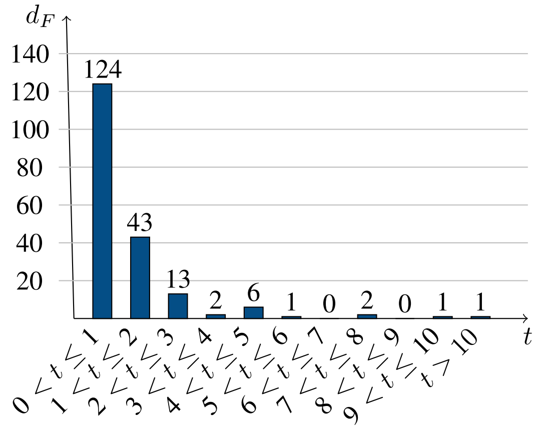

Fig. 4. Test case generation of the benchmark program `tri` over time. The t-axis separates the time into continues intervals of one second. The d_F-axis shows the number of the logic formulae solved by the SMT solver in the individual continues time intervals.

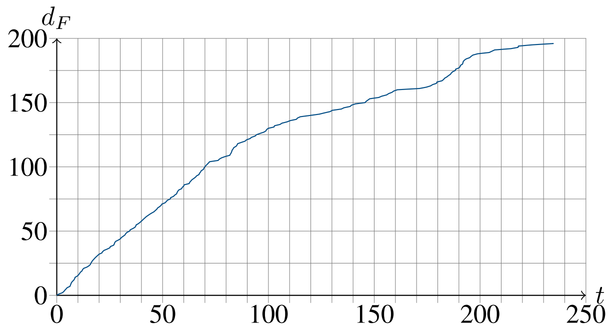

Fig. 5. Test case generation of the benchmark program `tri` over time. The t-axis shows the accumulated time needed by the SymBMC procedure in seconds. The d_F-axis gives the total number of faults detected by SymBMC.

solved by the SMT solver. For instance, for 124 seeded faults (60.19%) of the benchmark program `tri`, solving the individual logic formulae takes less or equal to one second per fault and, remarkably, for only five faults (2.46%) solving the individual logic formulae takes five or more seconds per fault.

Figure 5 shows the test case generation of the benchmark program `tri` over time. On the t-axis, we show the accumulated time needed by SymBMC and on the d_F-axis, we give the total number of faults detected by SymBMC.

Table I indicates that a completely symbolic approach may not scale to larger programs. Increasing the size of a program, drastically increases the time required by SymBMC. The precise test case generation from mutants in a particular, local context of a program still appears reasonable. Moreover, Figure 4 shows that a significant number of faults is detected very fast, which indicates that a heuristic approach focusing on these faults has the potential to speed up the test case generation.

VI. CONCLUSION

In this paper, we propose a new symbolic procedure, SymBMC, for the automated generation of test cases from a set of mutants using bounded model checking techniques. The procedure SymBMC obtains a model from a "meta" program, annotated with a set of artificial faults, and encodes the model as a logic formula using the theory of quantifier-free bit-vectors of arbitrary size. The SymBMC procedure derives a set of test cases by incrementally solving the logic formula. When a satisfying assignment detects a fault, the corresponding fault is excluded from the search space by a blocking clause.

We have presented initial empirical results using a prototype implementation with an SMT solver as backend. The results of the case study, despite still very small, are promising. Our symbolic procedure achieves on the example set considered a mutation score of 100% with respect to the maximum unrolling bound k. We conclude that a completely symbolic approach gives deeper insights into the problem of generating test cases from mutants and allows the development of better heuristics in the future.

VII. Acknowledgment

We would like to thank Finn Haedicke for supporting the development of the SMT solver backend used by SymBMC.

References

[1] R. G. Hamlet, "Testing programs with the aid of a compiler," *IEEE Transactions on Software Engineering*, vol. SE-3, no. 4, pp. 279–290, 1977.

[2] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *IEEE Computer*, vol. 11, no. 4, pp. 34–41, 1978.

[3] A. P. Mathur and W. E. Wong, "A theoretical comparison between mutation and data flow based test adequacy criteria," in *22nd Annual ACM Conference on Computer Science*, 1994, pp. 38–45.

[4] A. J. Offutt and J. M. Voas, "Subsumption of condition coverage techniques by mutation testing," Department of Information and Software Systems Engineering, George Mason University, Tech. Rep. ISSE-TR-96-100, 1996.

[5] P. Ammann and A. J. Offutt, *Introduction to Software Testing*. Cambridge Univeristy Press, Cambridge, 2008.

[6] P. G. Frankl and S. N. Weiss, "An experimental comparison of the effectiveness of branch testing and data flow testing," *IEEE Transactions on Software Engineering*, vol. 19, no. 8, pp. 774–787, 1993.

[7] A. P. Mathur and W. E. Wong, "An empirical comparison of data flow and mutation-based test adequacy criteria," *Software Testing, Verification, and Reliability*, vol. 4, no. 1, pp. 9–31, 1994.

[8] A. J. Offutt, J. Pan, K. Tewary, and T. Zhang, "An experimental evaluation of data flow and mutation testing," *Software — Practice & Experience*, vol. 26, no. 2, pp. 165–176, 1996.

[9] N. Li, U. Praphamontripong, and A. J. Offutt, "An experimental comparison of four unit test criteria: Mutation, edge-pair, all-uses and prime path coverage," in *IEEE International Conference on Software Testing, Verification and Validation Workshops*, 2009, pp. 220–229.

[10] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," CREST Centre, King's College London, Tech. Rep. TR-09-06, 2009.

[11] E. Clarke, O. Grumberg, and D. Peled, *Model Checking*. MIT Press, 1999.

[12] M. R. Girgis and M. R. Woodward, "An integrated system for program testing using weak mutation and data flow analysis," in *IEEE International Conference on Software Engineering*, no. 313-319, 1985.

[13] A. J. Offutt, "Automated test data generation," Ph.D. dissertation, Georgia Institute of Technology, Atlanta GA, 1988.

[14] R. A. DeMillo and A. J. Offutt, "Constraint-based automatic test data generation," *IEEE Transactions on Software Engineering*, vol. 17, no. 9, pp. 900–910, 1991.

[15] A. Goldberg, T. C. Wang, and D. Zimmerman, "Applications of feasible path analysis," in *International Symposium on Software Testing and Analysis*, 1994, pp. 80–94.

[16] C. Barrett, R. Sebastiani, S. Seshia, and C. Tinelli, "Satisfiability modulo theories," in *Handbook of Satisfiability*. IOS Press, Amsterdam, 2008, ch. 23, pp. 737–797.

[17] H. Gatzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli, "DPLL(T): Fast decision procedures," in *Computer Aided Verification*, vol. 3114, 2004, pp. 175–188.

[18] W. Visser, K. Havelund, G. Brat, and S. Park, "Model checking programs," in *IEEE International Conference on Automated Software Engineering*, 2000, pp. 3–11.

[19] 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*, 2004, pp. 168–176.

[20] 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*, 2008, pp. 209–224.

[21] C. Sinz, S. Falke, and F. Merz, "A precise memory model for low-level bounded model checking," in *Workshop on System Software Verification*, 2010, pp. 7–16.

[22] 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*, 1999, pp. 193–207.

[23] C. Lattner, "LLVM: An infrastructure for multi-stage optimization," Master's thesis, University of Illinois at Urbana-Champaign, 2002.

[24] R. H. Untch, A. J. Offutt, and M. J. Harrold, "Mutation analysis using mutant schemata," *ACM SIGSOFT Software Engineering Notes*, vol. 18, no. 3, pp. 139–148, 1993.

[25] A. J. Offutt, Z. Jin, and J. Pan, "The dynamic domain approach for test data generation: Design and algorithm," Department of Information and Software Systems Engineering, George Mason University, Tech. Rep. ISSE-TR-94-110, 1194.

[26] ——, "The dynamic domain reduction procedure for test data generation," *Software — Practice & Experience*, vol. 29, no. 2, pp. 167–193, 1999.

[27] A. J. Offutt and J. Pan, "Automatically detecting equivalent mutants and infeasible paths," *Software Testing, Verification, and Reliability*, vol. 7, no. 3, pp. 165–192, 1997.

[28] P. E. Ammann, P. E. Black, and W. Majurski, "Using model checking to generate tests from specifications," in *IEEE International Conference on Formal Engineering Methods*, 1998, pp. 46–54.

[29] V. Okun, P. E. Black, and Y. Yesha, "Testing with model checker: Insuring fault visibility," in *International Conference on System Science, Applied Mathematics & Computer Science, and Power Engineering Systems*, 2002, pp. 1351–1356.

[30] E. M. Clarke and E. A. Emerson, "Design and synthesis of synchronization skeletons using branching-time temporal logic," *Logics of Programs*, vol. 131, no. 10, pp. 52–71, 1982.

[31] P. E. Black, "Modeling and marshaling: Making tests from model checker counterexamples," in *Digital Avionics Systems Conference*, 2000, pp. 1–6.

[32] G. Fraser and F. Wotawa, "Mutant minimization for model-checker based test-case generation," in *Testing: Academic and Industrial Conference Practice and Research Techniques*, 2007, pp. 161–168.

[33] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation," *toc*, vol. C-35, no. 8, pp. 677–691, 1986.

[34] A. Pnueli, "The temporal logic of programs," in *Annual Symposium on Foundations of Computer Science*, 1977, pp. 46–57.

[35] A. Armando, J. Mantovani, and L. Platania, "Bounded model checking of software using SMT solvers instead of SAT solvers," *International Journal on Software Tools for Technology Transfer*, vol. 11, no. 1, pp. 69–83, 2009.

[36] H. G. Rice, "Classes of recursively enumerable sets and their decision problems," in *Transactions of the American Mathematical Society*, vol. 74, no. 2, 1953, pp. 358–366.

[37] T. A. Budd and D. Angluin, "Two notions of correctness and their relation to testing," *Acta Informatica*, vol. 18, no. 1, pp. 31–45, 1982.

[38] C. Lattner, "LLVM language reference manual," 2010, last visit on 22nd of December, 2010.

[39] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf, "An experimental determination of sufficient mutant operators," in *ACM Transactions on Software Engineering and Methodology*, 1996, pp. 99–118.

[40] B. J. M. Gruen, D. Schuler, and A. Zeller, "The impact of equivalent mutants," in *IEEE International Conference on Software Testing, Verification and Validation Workshops*, 2009, pp. 192–199.

[41] K. N. King and A. J. Offutt, “A Fortran language system for mutation-based software testing,” *Software — Practice & Experience*, vol. 21, no. 7, pp. 685–718, 1991.

[42] R. Brummayer and A. Biere, “Boolector: An efficient SMT solver for bit-vectors and arrays,” in *Tools and Algorithms for the Construction and Analysis of Systems*, 2009, pp. 174–177.