Using a Bounded Model Checker for Test Generation: How to Kill Two Birds with One SMT-solver

Mikhail Belyaev, Kirill Gagarski, Maxim Petrov, and Vladimir Itsykson

Saint-Petersburg State Polytechnical University, Russia
{belyaev,gagarski,maxim.petrov}@kspt.icc.spbstu.ru vlad@icc.spbstu.ru

Abstract. Automated test generation has received a lot of attention in the last decades as it is one of the possible solutions to software testing's inherent problems: the need to write tests in the first place and providing test coverage in presence of human factor.
The de-facto most promising technique to generate test automatically is dynamic symbolic execution assisted by an automated constraint solver, e. g., an SMT-solver. This process is very similar to bounded model checking, which also has to deal with generating models from source code, asserting logic properties in it and process the returned model.
This paper describes a prototype unit test generator for C based on a working bounded model checker called Borealis and shows that these two techniques are very similar and can be easily implemented using the same basic components.
The prototype test generator has been evaluated on a number of examples and showed good results in term of test coverage and test excessiveness.

Keywords: Automatic Test Generation, Dynamic Symbolic Execution, Bounded Model Checking, Satisfiability Modulo Theories

1 Introduction

Modern world is a world of software. It is widely adopted in many areas of human life, including medical tools, space stations and nuclear power plants. In this world of software the cost of developer's error is very high.

The de-facto way of automating quality assurance in modern world is software testing, which has a number of problems. First, tests consume human resources as it takes time to both write the tests and support them. Another problem is human factor, as it is widely accepted [21] that programmers tend to make the same (sometimes incorrect) assumptions about data in tests as they do when writing code in the first place.

One of the possible solutions to these problems is automated test generation. It has been a rising topic of research in the last decades [22, 12, 9] and is one of the possible solutions to software testing's inherent problems. The most widespread approach to automated test generation at the moment is dynamic symbolic execution [26], based on selecting input data by using a logic engine (e. g., an SMT solver) to ensure the desired runtime behaviour of tests. This technique is very similar to bounded model checking (BMC), albeit with a different purpose, as BMC primarily focuses on finding software defects and violations of user-supplied contracts. The combination of BMC and test generation in one tool seems quite natural as the targets of test generation and software verification complement each other rather well to ensure software quality.

In this paper we present an ongoing work to implement a fully working unit test generation tool for C based on a bounded model checker called Borealis [1]. We show that the modifications needed to adapt a completely working bounded model checker to the task of generating unit tests is minimal compared to implementing both tasks as separate tools. We evaluate the approach taken on a number of examples.

The rest of the paper is structured as follows. Section 2 gives a brief introduction to topics of SMT, bounded model checking and test generation. Section 3 describes our approach in details. Section 4 includes a number of our prototype tool implementation details. Evaluation of our work is given in section 5.

2 Background

2.1 Bounded Model Checking

Model checking is a well-known approach to checking correctness and safety violations of code during compile time using exhaustive state space exploration. However, while model checking is particularly good for dealing with finite-space systems, it can become very inefficient due to state space explosion in larger programs. One way of dealing with this problem is bounded model checking [7]. This method is based on limiting the state space by analyzing program paths up to a given length, e.g. limiting loop iterations and recursive calls. The bounded model can be then converted to a logic formula (using theories needed to model program behaviour, namely, bit vectors, uninterpreted functions, and arrays) and solved by a logic engine, usually an SMT-solver that supports the needed theories.

Bounded model checking has been a very active area of research in last years. A number of tools have been introduced based on this technique (most widely known of which are CBMC [8], SMT-CBMC [2], LLBMC [19] and ESBMC [10]) to detect software defects, code contract violations and other possible problems of code.

The tool we use as the basis of our implementation is the Borealis project [1], which is based on LLVM compiler infrastructure and the Z3 SMT solver. It is fully capable of dealing with programs written in C to detect software defects as well as code contract specification and checking. It provides two ways to specify code contracts: a comment-based language of annotations similar to ACSL [3] and in-code intrinsic calls.

2.2 Dynamic Symbolic Execution for Test Generation

Dynamic Symbolic Execution (DSE) [13, 24, 25] is a state-of-the-art technique for generation of test data for white-box unit testing of functions. The technique is based on exploring possible path space for a known function by trying to deduce input data that leads to a different code path. Doing this efficiently is the key point of DSE.

DSE is an advanced technique that originally sparked from feedback-directed random testing [23] that is itself based on random testing [14] as a way to overcome random data well-known problems [5]. Test generation tools get the full knowledge about program code, thus becoming a white-box testing technique. This information can be used to limit the possible state of inputs the tool can generate thus limiting the complexity of test-generation algorithms.

There are different approaches to generating the needed test data based on function code. One of these approaches (and, up to the best of our knowledge, the most effective) is to turn the program into a logic formula and possible paths to logical constraints that can be solved using a constraint solver (e.g. an SMT solver). This approach is the basis of Pex [25], the de-facto leading test generating tool for `.Net`. This approach is frequently viewed as using model checking for test generation. Furthermore, the problems for using this technique to generate tests for C are the same (global mutable state, memory operations, loops, function calls) as those dealt with by bounded model checking.

We postulate that SMT-based bounded model checking and SMT-based dynamic symbolic execution are essentially identical mechanisms that can be clearly implemented using the same basic components. There are only two differences: the way constraints are formed and the way the resulting model is used. BMC for defect detection/contract violation detection uses the contract/defect as a constraint, while DSE introduces path constraints based on how the control flow is structured. When using the BMC for defect detection/contract violation, the model acquired from the SMT solver is usually used only to produce better debug/informational messages to the tool user, while DSE needs the model to generate source code for the tests.

Generation of test oracles, another subgoal of test generation, cannot be effectively done by automatic tools and should be based on specification. These specifications are essentially identical to checked postconditions in BMC and can be reused from them in a transparent manner.

3 Turning Bounded Model Checking to Test Generation

3.1 Employing Code Contracts for Test Generation

Automated test generation experience can be greatly improved by providing means to specify intended limits on inputs (thus not generating test data in ranges the function is not supposed to handle) and outputs (thus providing basis to generate test oracles). These two concepts correspond very well to code contracts for functions in the form of preconditions (required data) and postconditions (ensured data). These specifications are widely used in BMC to specify the intended behaviour for code as well as in test driven development [4] and design by contract [20] techniques.

Code annotations in form of preconditions, postconditions and assertions are very common among BMC tools. Most popular BMC benchmarks (e.g. NECLA [15] and SVCOMP [6]) are mostly focused on checking code contracts rather than finding software defects. These annotations can be exploited to acquire contract information during test generation.

3.2 Test Generation Goal

This paper focuses on the overall idea of implementing a unit test generation tool using a working BMC solution. More advanced techniques for generating data and asserting results are not covered due to not being implemented at the time.

In order to assess the test generation results, we use two basic parameters: test coverage and test excessiveness. The coverage measure is based on statement coverage rather than condition/path/branch coverage as being the simplest one to measure. The best results could be achieved using path coverage as measure, but that potentially leads to an explosion in number of required tests (2^{38}) potential test cases for a function with 38 independent `if` statements). The statement coverage is good enough in most cases and can be achieved using the least (among other coverage types) number of test cases. The excessiveness measure is based on the fraction of tests in the suite that are redundant, i.e. do not affect the coverage in any way and could be removed.

Let us consider a simple example of an annotated function in C:

```
// @ensures \result >= 0
// @ensures \result == \arg || \result == -\arg
int abs(int a) {
    if(a >= 0) return a;
    else return -a;
}
```

The `abs` function is an example of the simplest possible (diamond-shaped) control flow graph and can be fully covered by two testcases, e. g. $a = \{1, -1\}$. If a test generation tool generates three testcases, one of them is redundand and the overall excessiveness of the test suite will be equal to 33%. If a test generation tool generates several testcases all of which are positive (e.g. $a = \{1, 2, 10042\}$) it does only achieve 50% of statement coverage. The goal of this work is to achieve a 100% statement coverage for arbitrary C programs while keeping excessiveness as low as possible.

3.3 Predicate Abstraction and Test Data

The BMC implementation we use in this work (see section 4) is based on summarizing program state as sets of logical predicates. Each predicate is essentially a logical formula of one of two kinds: a *path predicate* or a *state predicate*. Path predicates are used to distinguish different paths of execution and directly correlate to conditional nodes in control flow graph of the function. State predicates are built from all other types of program constructs that do not affect the control flow. A *Predicate State* is either a simple set of predicates, a sequence of states or a choice of possible states divided by a path predicate condition. Using this model, the program can be summarized as a single compound state that can be leveled down to logical formulae and, at the same time, avoid unnesessary duplication of predicates.

Assuming the code is in static single assignment (SSA) form, this model can be directly acquired from source code instructions (this implementation uses LLVM IR as the source model for this transformation) by treating all the branching instructions and φ-nodes as path predicates and all the other instructions (arithmetics, logic operations, casts, etc.) as state predicates. Memory-independent instructions are treated as equality predicates (between the left and right parts), while memory-dependent ones are treated as operations on global memory arrays (see below).

Each formula inside a predicate is in first-order logic using bit-vector, uninterpreted function and array theories. Predicates are context-sensitive because of the need to model memory and global variables. The approach uses array theory to simulate memory (essentially, generating a new SMT array for each new memory state) and to turn a *State* into a single SMT expression, one needs to interpret the predicates on this set of arrays. Global variables are represented as special memory locations and as such they do not require any special treatment.

In order to provide statement coverage, we need to create a set of SMT formulae, each one corresponding to a point of execution after a conditional statement in the program. This boils down to constructing predicate states up from the function entry to each path predicate, thus forcing SMT solver to generate a model that corresponds to a single path covering this statement. Note, however, that each path constitutes of several CFG nodes and is likely to cover several path predicates thus leading to test excessiveness. If a predicate is impossible (that is, the solver returned an UNSAT result for the corresponding formula), it is considered dead code and cannot be covered by any test case.

4 Implementation Details

4.1 Prototype Overview

Our prototype tool is based on Borealis bounded model checker project [1] which uses Clang [17] for code lexing and parsing, LLVM [18] infrastructure for code analysis and Z3 [11] as the logic engine. The tool overview is shown on fig. 1.

Borealis operates the code via so-called LLVM passes — interdependant elementary operations on LLVM IR that are the basic building blocks of the LLVM framework. Test generation tool is implemented as two passes. The first pass invokes Borealis model checker procedures to gather input data for each basic block of LLVM IR. The second pass is responsible for dumping tests to C code and inserting test oracles based on Borealis contract data. We currently use `CUnit` [16] as the target test framework.

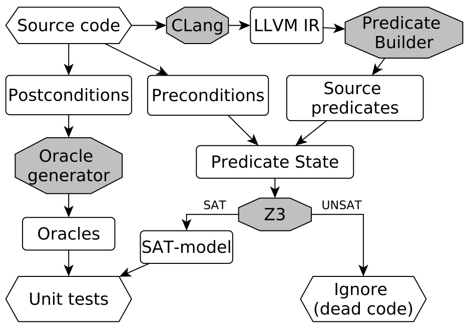

Fig. 1. Prototype overview

4.2 Extracting Input Data

LLVM intermediate code representation (LLVM IR) is structured in basic blocks — sequences of instructions that do not alter the control flow. Basic blocks are interconnected by branching instructions (always at the end of a basic block) and so-called φ-functions (always at the beginning of a basic block). Providing statement coverage effectively means asking Borealis for state at the beginning of each basic block.

In order to reduce excessiveness, additional measures are provided. Basic blocks that are fall-through (that is, that are always executed unconditionally) should be ignored unless the function consists of a single block. Function entry blocks can be ignored as well as they are executed every time function is called. Another simple measure is to ignore similar data generated for different blocks.

4.3 Generating Test Code

Each set of input data extracted by SMT-solver is one test case. For each test case we generate one tested function call. If function has contracts then we can use them for generating test oracles. Using contracts allows us to generate more complicated unit tests.

The function's pre-conditions describe constraints for the function input parameters. These are added to the predicate states automatically by state-building techniques and have no special handling for test generation.

Post-conditions are used to generate test oracles. They are specified in terms of input arguments and return value (`\result`). The return value of the function is stored into local variable. The post-conditions are simplified and converted into comparision statements which are inserted into unit tests as oracles.

5 Evaluation

An example of generated testcases for the `abs` function from section 3.2 follows. Pre-conditions are satisfied and post-conditions are checked using `CU_ASSERT`.

```
void testAbs_0(void) {
    int x = -1;
    int res = abs(x);
    CU_ASSERT((res > 0));
    CU_ASSERT(((res == x) || (res == -(x))));
}
void testAbs_1(void) {
    int x = 1;
    int res = abs(x);
    CU_ASSERT((res > 0));
    CU_ASSERT(((res == x) || (res == -(x))));
}
```

We have tested our prototype tool on the number of examples. The results are shown in table 1. Note that these examples are synthetic. As one can see, we have a statement coverage of 100% for all the cases. An excessiveness rate of 75% (the maximum value in the table) means that at average we have approximately 7 redundant testcases for 10 non-redundant ones, which is not a bad result.

Table 1. Test results

Tested function	Test cases	Redundant test cases	Statement coverage (modulo dead code)	Path coverage	Test excessiveness
Sum of naturals	7	0	100%	<1%	0%
Absolute value	2	0	100%	100%	0%
Square	1	0	100%	100%	0%
Is even?	1	0	100%	100%	0%
Synthetic function with multiple branches	7	2	100%	80%	29%
Example with redundant test cases	4	3	100%	75%	75%
Example with two ifs with same condition	3	2	100%	50%	67%
Example with dead code	2	1	100%	100%	50%

6 Conclusion

This work is focused on creating a working unit test generation tool prototype on top of an existing bounded model checker. We have evaluated the prototype on a set of simple test programs and have shown that it provides satisfiable coverage and excessiveness characteristics.

Current prototype does not support complex in-memory data structures largely used in C, intraprocedural effects that can happen during testing and provides very simple excessiveness reduction algorithms.

In the future we plan to overcome this limitations by introducing a memory structure reconstitution algorithm and do further research on the topic of test minimization.

References

[1] Marat Akhin, Mikhail Belyaev, and Vladimir Itsykson. Yet another defect detection: Combining bounded model checking and code contracts. PSSV'13, pages 1–11, 2013.

[2] Alessandro Armando, Jacopo Mantovani, and Lorenzo Platania. Bounded model checking of software using SMT solvers instead of SAT solvers. *Int. J. Softw. Tools Technol. Transf.*, 11(1):69–83, jan 2009.

[3] Patrick Baudin, Jean C. Filliâtre, Thierry Hubert, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. *ACSL: ANSI/ISO C Specification Language. Preliminary Design, version 1.4, 2008.*, preliminary edition, may 2008.

[4] Kent Beck. *Test-driven development: by example.* Addison-Wesley Professional, 2003.

[5] Boris Beizer. *Software testing techniques.* Dreamtech Press, 2003.

[6] Dirk Beyer. Competition on software verification. TACAS'12, pages 504–524. Springer, 2012.

[7] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. TACAS'99, pages 193–207, 1999.

[8] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. TACAS'04, pages 168–176, 2004.

[9] David M Cohen, Siddhartha R Dalal, Jesse Parelius, and Gardner C Patton. The combinatorial design approach to automatic test generation. *IEEE software*, 13(5):83–88, 1996.

[10] Lucas Cordeiro, Bernd Fischer, and Joao Marques-Silva. SMT-based bounded model checking for embedded ANSI-C software. ASE'09, pages 137–148, 2009.

[11] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. pages 337–340, 2008.

[12] RA DeMilli and A. Jefferson Offutt. Constraint-based automatic test data generation. *Software Engineering, IEEE Transactions on*, 17(9):900–910, 1991.

[13] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In *ACM Sigplan Notices*, volume 40, pages 213–223. ACM, 2005.

[14] Richard Hamlet. Random testing. *Encyclopedia of software Engineering*, 1994.

[15] Franjo Ivančić and Sriram Sankaranarayanan. NECLA static analysis benchmarks.

[16] Anil Kumar and J St Clair. Cunit-a unit testing framework for c. *Diposnıvel em: http://cunit. sourceforge. net/doc/index. html. Acesso em*, 25, 2005.

[17] Chris Lattner. LLVM and Clang: Next generation compiler technology. In *The BSD Conference*, 2008.

[18] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. CGO'04, pages 75–86, 2004.

[19] Florian Merz, Stephan Falke, and Carsten Sinz. LLBMC: Bounded model checking of C and C++ programs using a compiler IR. VSTTE'12, pages 146–161, 2012.

[20] Bertrand Meyer. Applying 'design by contract'. *Computer*, 25(10):40–51, 1992.

[21] Glenford J Myers, Corey Sandler, and Tom Badgett. *The art of software testing.* John Wiley & Sons, 2011.

[22] Clementine Nebut, Franck Fleurey, Yves Le Traon, and J-M Jezequel. Automatic test generation: A use case driven approach. *Software Engineering, IEEE Transactions on*, 32(3):140–155, 2006.

[23] Carlos Pacheco, Shuvendu K Lahiri, Michael D Ernst, and Thomas Ball. Feedback-directed random test generation. In *Software Engineering, 2007. ICSE 2007. 29th International Conference on*, pages 75–84. IEEE, 2007.

[24] Koushik Sen, Darko Marinov, and Gul Agha. *CUTE: a concolic unit testing engine for C*, volume 30. ACM, 2005.

[25] Nikolai Tillmann and Jonathan De Halleux. Pex—white box test generation for .Net. In *Tests and Proofs*, pages 134–153. Springer, 2008.

[26] Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Wolfram Schulte. Fitness-guided path exploration in dynamic symbolic execution. pages 359–368. IEEE, 2009.