What's Reusable In Program Analysis?

[Extended Abstract]

Grigory Fedyukovich
Computer Science and Engineering, University of Washington, Seattle, Washington, USA
grigory@cs.washington.edu

Software continuously evolves to meet rapidly changing human needs. Each evolved transformation of a program is expected to preserve important correctness and security properties. Aiming to assure program correctness after a change, formal verification techniques, such as Software Model Checking, have recently benefited from fully automated solutions based on symbolic reasoning and abstraction. However, the majority of the state-of-the-art model checkers are designed that each new software version has to be verified from scratch.

We present a survey on Formal Incremental Verification (FIV) techniques that aim at making software analysis more efficient by reusing invested efforts between verification runs. In order to show that FIV can be built on top of different verification techniques, we focus on three complementary approaches to automated formal verification.

First, we present a FIV technique for SAT-based Bounded Model Checking (BMC) developed to verify programs with nested functions with respect to the set of pre-defined assertions. We propose a function-summarization framework that allows extracting and reusing over-approxi-mations of function behaviors. We introduce an algorithm to revalidate the summaries of one program locally in order to prevent re-verification of another program from scratch.

Second, we present a technique for simulation relation synthesis for loop-free programs that do not necessarily contain assertions. We introduce an SMT-based abstraction-refinement algorithm that proceeds by guessing a relation and checking whether it is a simulation relation. We propose a novel algorithm for discovering simulations symbolically, by means of solving $\forall\exists$-formulas and extracting witnessing Skolem relations.

Third, we present a FIV technique for SMT-based Unbounded Model Checking developed to verify programs with (possibly nested) loops. We propose an algorithm that automatically derives simulations between programs with different loop structures. The automatically synthesized simulation relation is then used to migrate the safe inductive invariants across the evolution boundaries.

1. SAT-Based Bounded Model Checking via Function Summarization

BMC is one of the most successful formal techniques in academia and industry. In a nutshell, a classical BMC tool proceeds in 3 main steps. First, it unwinds all loops and recursive function calls up to a given number of iterations and a given recursion depth respectively. This phase results in the unwound program represented by the so called Static Single Assignment (SSA) form in which all variables are assigned at most once. The SSA form is then conjoined with the negation of the assertion. Second, the constructed SSA form is encoded to a so called BMC formula and sent to the appropriate SAT or SMT solver. Finally, the rest of the job is done by the solver: if the formula is proven unsatisfiable then the program is safe up to the given bound; otherwise each model of the formula witnesses a counter-example.

The biggest challenge in BMC related to FIV is searching for a *reusable* specification, i.e., an essence to migrate between verification runs. We propose to exploit the fact that the bounded safety of the program is indicated by the unsatisfiability of the BMC formula. In the context of SAT-based BMC, we show that the proof of unsatisfiability can be further used to discover over-approximating function summaries that gather all important information of the function's behavior to prove the bounded safety. Finally, we present a novel technique based on Craig interpolation [1] to achieve cheap and flexible function summarization.

The next challenge in BMC related to FIV is searching for an algorithm that effectively reuses the summaries synthesized after the verification of one program version to verify another program version. We propose to revalidate the existing summaries locally in order to prevent re-verification of the entire code from scratch. If the check fails for some function call, it needs to be propagated by the call tree traversal to the caller of the function. If the check fails for the root of the call tree (i.e., the "main" function of the program), the whole program should be verified from scratch. On the other side, if for each function call there exists an ancestor function with the valid old summary then the new program version is safe up to the predefined bound. Finally, for functions whose old summaries are not valid any longer, the new summaries are re-synthesized.

We refer the reader to the following papers to get more details of the technique, its implementation in a tool eVolCheck and evaluation: [10], [9], [5], and [6].

2. SMT-Based Simulation Discovery

Simulation [8] is one of the oldest concepts in program analysis, introduced as early as Hoare logic. A simulation relation represents a condition under which the complete set of behaviors of one program (later referred to as source and denoted by S) is included into the set of behaviors of another program (later referred to as target and denoted by T). Simulation discovery is a useful procedure for FIV since it does not require any assertions to be specified at the programs. Indeed, if S is simulated by T then all assertions that hold in T will also hold in S. Thus, a discovered simulation provides a more precise verification certificate (namely, a *relational* specification) than the function-summarization BMC approach from Sect. 1. When synthesized, this relational specification is an important ingredient for another class of FIV (to be discussed in Sect. 3), since it allows lifting the safe inductive invariants from T to S.

In realistic applications, there might be a sufficient semantic gap between S and T that essentially raises two challenges of finding an appropriate simulation relation between S and T: (1) the challenge of constructing a total simulation relation between two programs, and (2) whenever the target T does not simulate the source S, the challenge of finding an abstraction of the target T that simulates the source S.

We propose an SMT-based solution for these challenges. Our algorithm uses an abstraction-refinement reasoning which proceeds by guessing a relation and checking whether it is a simulation relation. We propose to reduce the problem of checking simulation to deciding validity of formulas of the form $\forall x \cdot S(x) \implies \exists y \cdot T(x, y)$. Intuitively, the formulas say "for each behavior of S there exists a corresponding behavior of T". We manipulate implicit abstractions of T by introducing existential quantifiers to the right-hand-side of the $\forall\exists$-formulas. We present a novel algorithm AE-VAL for deciding validity of $\forall\exists$-formulas that is based on efficient computations of the Model-Based Projections [7]. In addition, AE-VAL extracts a Skolem relation to witness the existential quantifiers. This Skolem relation is the key to refine the considered abstractions of T, and therefore requires to be minimized and factored. As a solution to this challenge, we propose to post-process the results of AE-VAL that results in a Skolem relation of the appropriate form.

We refer the reader to the paper [3] to get more details of the technique, its implementation in a tool SimAbs and evaluation.

3. SMT-Based Unbounded Model Checking via Abstract Simulation

The approaches to Unbounded Software Model Checking reduce the verification tasks to finding safe inductive invariants. The safe inductive invariants play an important role of proof certificates and over-approximate all safe behaviors of the program. Therefore, these techniques are served to provide sound analysis of the programs with unbounded (and possibly nested) loops. To compactly represent such complex programs, model checkers use a "large-block" encoding (LBE) that collapses the control-flow graph (CFG) into the Cut-Point Graph (CPG). In CPG, the nodes represent heads of the loops (called cutpoints), and the edges represent the longest loop-free program fragments. Whenever a program is proven safe, the CPG is labeled by predicates, such that for each CPG-edge the labeling of the corresponding in- and out- nodes constitutes a valid Hoare triple.

We refer to the challenge of constructing a FIV technique for Unbounded Model Checking as establishing a Property Directed Equivalence (PDE) between programs, i.e., to check whether the programs are happy with the same safe inductive invariant. Clearly, in contrast to BMC-based FIV (outlined in Sect. 1), PDE does not have a challenge for synthesizing a reusable specification, since the safe inductive invariants perfectly fit this goal. However, there still remains a challenge of migrating the existing invariant between two programs (the already verified one, and the modified another one). We propose a solution based on the concept of Abstract Simulation outlined in Sect. 2. We contribute an algorithm that performs an iterative abstract-refinement reasoning to automatically derive an abstraction of the already verified program that simulates the precise modified program.

One important feature of the algorithm is that it guides the abstraction generation by the safe inductive invariant. If a simulation for such a proof-based abstraction is found then the proof can be migrated directly. Another distinguishing feature of the algorithm is the ability to migrate the invariants through abstractions even if the abstractions do not preserve safety. It attempts to lift as much information from the invariant as possible, and then strengthens it using a Horn-based unbounded model checker.

We refer the reader to the papers [2] and [4] to get more details of the technique, its implementation in tools ASSI+PDE and evaluation.

4. REFERENCES

[1] W. Craig. Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. In *J. of Symbolic Logic*, pages 269–285, 1957.

[2] G. Fedyukovich, A. Gurfinkel, and N. Sharygina. Incremental verification of compiler optimizations. In *NFM*, volume 8430 of *LNCS*, pages 300–306. Springer, 2014.

[3] G. Fedyukovich, A. Gurfinkel, and N. Sharygina. Automated discovery of simulation between programs. In *LPAR*, volume 9450, pages 606–621. Springer, 2015.

[4] G. Fedyukovich, A. Gurfinkel, and N. Sharygina. Property directed equivalence via abstract simulation. In *CAV*, volume 9780, Part II, pages 433–453. Springer, 2016.

[5] G. Fedyukovich, O. Sery, and N. Sharygina. eVolCheck: Incremental Upgrade Checker for C. In *TACAS*, volume 7795 of *LNCS*, pages 292–307. Springer, 2013.

[6] G. Fedyukovich, O. Sery, and N. Sharygina. Flexible SAT-based Framework for Incremental Bounded Upgrade Checking. *STTT*, 17:1–18, 2015.

[7] A. Komuravelli, A. Gurfinkel, and S. Chaki. SMT-Based Model Checking for Recursive Programs. In *CAV*, volume 8559 of *LNCS*, pages 17–34, 2014.

[8] R. Milner. An algebraic definition of simulation between programs. In *IJCAI*, pages 481–489, 1971.

[9] O. Sery, G. Fedyukovich, and N. Sharygina. Incremental Upgrade Checking by Means of Interpolation-based Function Summaries. In *FMCAD*, pages 114–121. IEEE, 2012.

[10] O. Sery, G. Fedyukovich, and N. Sharygina. Interpolation-based Function Summaries in Bounded Model Checking. In *HVC*, volume 7261 of *LNCS*, pages 160–175. Springer, 2012.