NIAGARA: Unbounded SMT-based Upgrade Checking and Program Repair
[Extended Abstract]

Grigory Fedyukovich* Arie Gurfinkel† Natasha Sharygina*

*Formal Verification Lab of the Faculty of Informatics, Università della Svizzera italiana, Lugano, Switzerland
†SEI/CMU, USA

I. PROBLEM AND MOTIVATION

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 [2], [3], [16], [1], [11], [15].

However, there is a clear need for new techniques that would make the analysis more efficient by reusing invested efforts between verification runs. The recent progress in this field has yielded numerous approaches to upgrade (equivalence) checking [14], [13], [9], [17], [12], [8], [7], each of which, however, has essential limitations. One common limitation of the upgrade checkers is their inability to further employ results of performed verification to fix detected bugs.

II. BACKGROUND AND RELATED WORK

Our previous research was pursued in context of SAT-based Bounded Model Checking. Our upgrade checker EVOLCHECK [8] extracts over-approximating function summaries using Craig interpolation [4] from the old program version and then re-checks if the summaries still over-approximate function behaviour in the new program version. While behaving relatively faster than re-verification of the new version from scratch, EVOLCHECK relies on the user-provided bound for loops and recursive function calls.

Our current research extends recently established *proof-based* verification [11] producing and manipulating *unbounded verification certificates* that are essentially safe inductive invariants in First Order Logic. *Safe* means that the invariant defines an over-approximation of the sets of possible program states excluding those that violate the given assertion. *Inductive* means that the invariant covers all the initial program states, and any computation starting from a state represented by the invariant can only reach states still represented by the invariant.

In [7], we proposed an approach OPTVERIFY to migrate the safety proofs across evolution boundaries. It is based on rewriting invariants using a guessed variable mapping, followed by invariants weakening via incremental computing the Maximal Unsatisfiable Subset [11] and finally, strengthening using a proof-based model checker. Notably, performance of OPTVERIFY strictly depends on the sufficiency of the variable mapping. Simulation relation [6] could provide the most suitable mapping for OPTVERIFY.

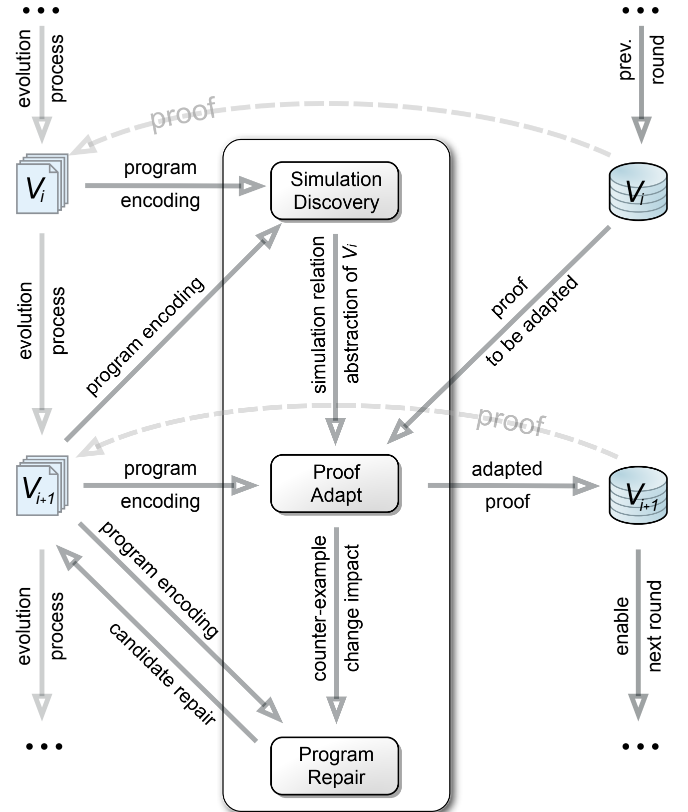

Fig. 1: A round of NIAGARA for V_i and V_{i+1}.

III. APPROACH AND UNIQUENESS

We present NIAGARA, an approach that combines efforts on unbounded proof-based upgrade checking and program repair. Its workflow outlined in Fig. 1 considers a transition within the evolutionary chain of a given program between versions V_i and V_{i+1}. NIAGARA obtains a simulation relation between versions, uses it to verify V_{i+1} by adapting the proof of V_i and (if disproven safe) attempts to incrementally fix the detected bug in V_{i+1}. Thus, NIAGARA iterates between the components Simulation Discovery, Proof Adapt, and Program Repair.

Simulation Discovery receives the symbolic encoding of V_i and V_{i+1}; and discovers an abstraction αV_i of V_i that simulates V_{i+1} via some total relation ρ. Initially, it guesses ρ to be a relation between V_{i+1} and (non-abstracted) V_i. All ingredients are encoded into a set of specialized $\forall\exists$-formulas, which are then checked for validity. The abstraction αV_i and the relation ρ are iteratively adjusted as a result of the weakening-strengthening reasoning driven by Skolem functions witnessing the existential quantifiers in the corresponding $\forall\exists$-formulas.

Proof Adapt takes V_i, V_{i+1} simulated by some αV_i via ρ and the safety proof $\widehat{\pi}_{V_i}$ of V_i w.r.t. assertion π; and either adapts $\widehat{\pi}_{V_i}$ to become a proof $\widehat{\pi}_{V_{i+1}}$ of V_{i+1}, or provides a counter-example. Our method of adapting $\widehat{\pi}_{V_i}$ to $\widehat{\pi}_{V_{i+1}}$ using ρ consists of: 1) transferring each invariant $\widehat{\pi}_{V_i}$ syntactically to a $\rho\widehat{\pi}_{V_{i+1}}$ (which will lose safety if the abstraction αV_i does not preserve the property π); 2) strengthening $\rho\widehat{\pi}_{V_{i+1}}$ (if necessary and possible) to become safe by a proof-based model checker. In both cases, if V_{i+1} is safe or not, NIAGARA generate a so called *change impact*, a function labeling the control-flow-graph (CFG) of V_{i+1} by boolean constants indicating whether the transformed inductive invariant in the correspondent program location did (or did not) require strengthening. In other words, it is an indication whether the code in a particular program location is affected by the change or not.

Program Repair receives V_{i+1}, a counter-example (i.e., a witness to violation of some assertion π) and the change impact between V_i and V_{i+1}; and outputs a *candidate repair*, a new program version V'_{i+1} which does not contain behaviors violating π. Our method aims to find an abstraction of V_{i+1} (where the code, not affected by the change, should be kept) such that it contains at least one behavior satisfying π. This abstraction is further refined using the witnessing Skolem function to finally deliver the candidate repair. The candidate repair is model-checked, and, if another counter-example is found, NIAGARA iterates until it discovers the repair fixing each counter-example, or a user intervention is made.

An important subroutine of Simulation Discovery and Program Repair is the novel decision procedure AE-VAL to solve quantified formulas and compute witnessing Skolem functions. It is assumed that an input formula to AE-VAL can be transformed into the form $S(\vec{x}) \implies \exists\vec{y}\,.\,T(\vec{x},\vec{y})$. AE-VAL iteratively discovers an *over-approximation* of $S(\vec{x})$ by a disjunction of quantifier-free Model-Based Projections (MBP [10]) of $\exists\vec{y}\,.\,T(\vec{x},\vec{y})$, s.t. $S(\vec{x}) \implies \bigvee_i T_i(\vec{x}) \implies \exists\vec{y}\,.\,T(\vec{x},\vec{y})$ (see example in Fig. 2). Skolem function is obtained as a natural by-product of the MBP algorithm.

IV. Results and Contributions

The approach of NIAGARA addresses the challenge of verifying the software safety by 1) enabling unbounded model checkers to scale to larger verification problems through reusing efforts between verification runs, and 2) providing automated solutions to fix buggy programs.

We implemented AE-VAL in Linear Rational Arithmetic on the top of the SMT solver Z3 [5]. We implemented Simulation Discovery and Proof Adapt (and extended them to support program versions with different CFGs) using AE-VAL and the model checker UFO [1] (find more details at [6]).

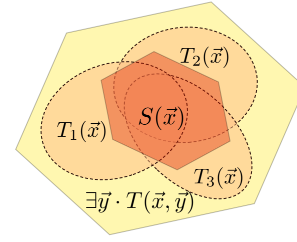

Fig. 2: 3 iterations of AE-VAL while deciding validity of $S(\vec{x}) \implies \exists\vec{y}\,.\,T(\vec{x},\vec{y})$ ($S, \exists T$ – hexagons, MBPs – ovals).

Our evaluation of NIAGARA on the LLVM optimizations and the benchmarks from Software Verification Competition confirmed that the program transformations can be checked efficiently. We are currently extending our development to Program Repair and expect to have promising results soon.

References

[1] A. Albarghouthi, A. Gurfinkel, and M. Chechik. UFO: A Framework for Abstraction- and Interpolation-Based Software Verification. In *CAV*, volume 7358 of *LNCS*, pages 672–678. Springer, 2012.

[2] T. Ball and S. Rajamani. Boolean Programs: A Model and Process for Software Analysis. Technical Report MSR-TR-2000-14, Microsoft Research, 2000.

[3] A. Biere, A. Cimatti, E. Clarke, O. Strichman, and Y. Zhu. Bounded Model Checking. *Advances in Computers*, 58:118–149, 2003.

[4] 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.

[5] L. M. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *TACAS*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

[6] G. Fedyukovich, A. Gurfinkel, and N. Sharygina. Automated discovery of simulation between programs. Technical Report USI:2014/05, Università della Svizzera italiana, 2014. http://www.inf.usi.ch/research_publication.htm?id=83.

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

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

[9] B. Godlin and O. Strichman. Regression verification. In *DAC*, pages 466–471. ACM, 2009.

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

[11] A. Komuravelli, A. Gurfinkel, S. Chaki, and E. M. Clarke. Automatic Abstraction in SMT-Based Unbounded Software Model Checking. In *CAV*, LNCS, pages 846–862. Springer, 2013.

[12] S. K. Lahiri, K. L. McMillan, R. Sharma, and C. Hawblitzel. Differential assertion checking. In *FSE*, pages 345–355. ACM, 2013.

[13] K. S. Namjoshi. Lifting Temporal Proofs through Abstractions. In *VMCAI*, volume 2575 of *LNCS*, pages 174–188. Springer, 2003.

[14] G. C. Necula. Translation validation for an optimizing compiler. In *PLDI*, pages 83–94. ACM, 2000.

[15] O. Sery, G. Fedyukovich, and N. Sharygina. FunFrog: Bounded model checking with interpolation-based function summarization. In *ATVA*, volume 7561 of *LNCS*, pages 203–207. Springer, 2012.

[16] Y. Xie and A. Aiken. Saturn: A SAT-Based Tool for Bug Detection. In *CAV*, volume 3576 of *LNCS*, pages 139–143. Springer, 2005.

[17] G. Yang, S. Khurshid, S. Person, and N. Rungta. Property differencing for incremental checking. In *ICSE*, pages 1059–1070. ACM, 2014.