VIAP 1.1
(Competition Contribution)

Pritom Rajkhowa(✉) and Fangzhen Lin

Department of Computer Science and Engineering,
The Hong Kong University of Science and Technology,
Clear Water Bay, Kowloon, Hong Kong
{prajkhowa,flin}@cse.ust.hk

Abstract. VIAP (Verifier for Integer Assignment Programs) is an automated system for verifying safety properties of procedural programs with integer assignments and loops. It is based on a translation from of a program to a set of first-order axioms with quantification over natural numbers, and currently makes use of SymPy as the algebraic simplifier and the SMT solver Z3 as the theorem prover. Our first version of the system competed at SV-COMP 2018. This paper describes VIAP 1.1, a new version that makes use of our newly developed recurrence solver. As a result, VIAP 1.1. is able to verify many programs that were out of reach for the older version VIAP 1.0.

Keywords: Automatic program verification · First-order logic · Mathematical induction · Recurrences · SMT · Arithmetic

1 Introduction

VIAP (Verifier for Integer Assignment Programs) is an automated system for verifying safety properties of procedural programs with integer assignments and loops. It translates a given program to a set of first-order axioms with natural number quantification using an algorithm proposed by Lin [1]. An earlier version of VIAP competed at SV-COMP 2018, and is described in [2,3]. A key feature of Lin's translation is that loops are translated to a set of recurrence relations. Then, VIAP simplifies those axioms by using a Python library for symbolic computation systems, *SymPy* [4], to compute the closed-form solutions of recurrence relations. *SymPy* is equipped with function *rsolve()* to compute closed-form solution of recurrence relation. The translation of the loop body generates recurrence relations which are either simple non-conditional, conditional or mutual in nature. But *rsolve()* can find the closed form solution only for certain class of simple non-conditional recurrence relations. This motivated us to design a recurrence solver (RS) that goes beyond what the *rsolve()* function can do in *SymPy*, and integrate it with our system. The new system, VIAP 1.1, is the one that will compete at this year's SV-COMP. VIAP 1.1 continues to use *SymPy* for simplifying algebraic expressions, and the SMT solver Z3 [5] as

D. Beyer et al. (Eds.): TACAS 2019, Part III, LNCS 11429, pp. 250–255, 2019.
https://doi.org/10.1007/978-3-030-17502-3_23

the underlying theorem prover without ever explicitly generating loop invariants. Because of the new recurrence solver, VIAP 1.1 can solve many more benchmarks that were previously out of the reach of VIAP 1.0.

To illustrate how our system works, consider the simple program below:

```
int x=0,y=0;
while (x<100) { if (x < 50){ y++; } else { y--; } x++; }
assert(y==0);
```

With some simple simplifications, the translation outlined in [1] would generate the following axioms:

$$x_1 = x_2(N), y_1 = y_2(N),$$
$$\forall n.x_2(n+1) = x_2(n) + 1, x_2(0) = 0,$$
$$\forall n.y_2(n+1) = ite(x_2(n) < 50, y_2(n) + 1, y_2(n) - 1), y_2(0) = 0,$$
$$\neg(x_2(N) < 100), \forall n.n < N \rightarrow x_2(n) < 100.$$

Here, x_1 and y_1 denote the output values of x and y, respectively, and $x_2(n)$ and $y_2(n)$ denote the values of x and y during the n-th iteration of the loop, respectively. The conditional expression $ite(c, e_1, e_2)$ has value e_1 if c holds and e_2 otherwise. Also N is a natural number constant, and the last two axioms say that it is exactly the number of iterations the loop executes before exiting.

There are two recurrence relations in the above axioms. Both the recurrence relations are passed to RS. It first solves $x_2(n)$ which yields the closed-form solution $x_2(n) = n$ which can then be used to simplify the recurrence relations for $y_2(n)$ into

$$y_2(0) = 0,\ y_2(n+1) = ite(n < 50, y_2(n) + 1, y_2(n) - 1).$$

Then RS tries to solve the above simplified conditional recurrence relations, and returns the following closed-form solution:

$$y_2(n) = ite(0 \leq n < 50, n, 50 - n).$$

After computing the closed-form solutions for $x_2()$ and $y_2()$ by RS, VIAP eliminates them, and produces the following axioms:

$$x_1 = N \wedge y_1 = ite(0 \leq N < 50, N, 100 - N), N \geq 100),$$
$$\forall n.n < N \rightarrow n < 100.$$

The translation of assertion results $y_1 == 0$. With this set of axioms, SMT solvers like Z3 can then be made to prove the assertion. Similarly, when an assertion like `assert(y==1)` is made to prove using above set of axioms, then Z3 will return following counterexample:

$$[y_1 = 0, N = 100, x_1 = 100].$$

Using this counterexample, VIAP constructs the violation witness.

2 VIAP Architecture

VIAP is implemented in Python 2. VIAP has been developed in a modular fashion, and its architecture is layered into two parts:

- **Front-End:** The system accepts a program written in C (C99 language) as input and translates it to first order axioms. The recurrence solver solves the recurrence relations generated during the translation if closed-form solutions are available.
- **Back-End:** The system takes the set of translated first-order axioms and translates all the axioms to equations compatible with Z3 (Version 4.5) by pre-processing them using *SymPy* (Version 1.1.1). Then the proof engine applies different strategies and tries to prove post-conditions in Z3 [2].

Translation. Given a program P, and a language $\boldsymbol{X}$, our system generates a set of first-order axioms denoted by $\Pi_P^{\boldsymbol{X}}$ that captures the changes of P on $\boldsymbol{X}$. Here, a language means a set of functions and predicate symbols. For $\Pi_P^{\boldsymbol{X}}$ to be correct, $\boldsymbol{X}$ needs to include all program variables in P as well as any functions and predicates that can be changed by P. The axioms in the set $\Pi_P^{\boldsymbol{X}}$ are generated inductively on the structure of P. The algorithm is described in detail in [1] and an implementation is explained in [2]. The inductive cases of translations are given in the table provided in the supplementary information[1]. We have extended our translation programs with arrays; the extension is described in detail in [3].

Recurrence Solver (RS). The main objective of this module is to find closed-form solutions of recurrence relations generated from the translation of the loop body. Our recurrence solver (RS)[2] takes a set of recurrence relation(s) and other constraints, returns a set of closed-form solutions it found for some of the recurrences and the remaining recurrences relations and constraints simplified using the computed closed-form solutions. It uses *SymPy* [4] (V 1.1.1) as the base solver. The RS classifies input recurrence relation(s) into three major categories (1. non-conditional 2. mutual and 3. conditional recurrences relation) and applies the following corresponding sub-solver and tries to find closed form solution(s).

- The Non-Conditional Recurrence Solver (*NCRS*): RS applies this sub solver to the non-conditional recurrence relation(s) of the form of either

$$X(n+1) = f(X(n), n),$$

where $f(x, y)$ is a polynomial function of x and y
or

$$X(n+1) = X(n) + f(n) + A_1F_1(n) + \cdots + A_kF_k(n),$$

where $f(n)$ is a polynomial function in n, A_i's are constants, and F_i's are function symbols.

[1] https://github.com/VerifierIntegerAssignment/VIAP_ARRAY/blob/master/Document/Inductive_Translation.pdf.

[2] https://github.com/VerifierIntegerAssignment/recSolver.

– The Mutual Recurrence Solver (MRS): RS applies this sub solver to a set $\boldsymbol{\sigma}$ of the mutual recurrence relations where each $\sigma \in \boldsymbol{\sigma}$ is the form of

$$X_i(n+1) = A * (X_1(n) + \ldots + X_h(n)) + C_i, \qquad \text{for } 1 \leq i \leq h,$$

where A and C_i are constants.
– The Conditional Recurrence Solver (CRS): RS applies this sub solver to conditional recurrence relation(s) of the form

$$X(n+1) = ite(\theta_1, f_1(X(n), n), ite(\theta_2, f_2(X(n), n) \ldots, f_{h+1}(X(n), n))),$$

where $\theta_1, \theta_2, \ldots, \theta_h$ are Boolean expressions, and $f_1(x, y), f_2(x, y), \ldots, f_{h+1}(x, y)$ are polynomial functions of x and y.

Instantiation: Instantiation is one of the most important phases of the pre-processing of axioms before the resulting set of formulae is passed on an SMT-solver according to some proof strategies. The objective is to help an SMT solver like Z3 to reason with quantifiers. There are two strategies (1) Instantiating arrays and (2) Instantiating array indices applied to an array element assignment that occurs inside a loop. More details are provide in the supplementary information[3].

Proof Strategies: As the semantics of P are precisely encoded as $\Pi_P^{\boldsymbol{X}}$, the goal is to prove that $\alpha \wedge \Pi_P^{\boldsymbol{X}} \models \beta$, where α is a set of assumption(s) and β is the set of assertion(s) to prove. We work in a refutation-based proof schema, i.e., in order to prove that a formula is valid in a background theory T, we show that $\alpha \wedge \Pi_P^{\boldsymbol{X}} \wedge \neg\beta$ is T-unsatisfiable. In VIAP, we implemented two different strategies whose details can be found in our previous work [2].

3 Strength and Weaknesses

VIAP supports user assertions, including reachability of labels in the C-code. In SV-COMP 2019, these checks are only enabled for ReachSafety-Arrays, ReachSafety-Loops and ReachSafety-Recursive sub-categories of ReachSafety category. VIAP translates a program to a set of axioms and then uses off-the-shelf systems like SymPy and Z3 to prove properties about the program. The advantage (strength) of this approach comes with a clean separation between the translation (semantics) and the use of the translation in proving the properties (computation). The translation part is stable. But as more efficient provers become available, the capabilities of the system improve. This is seen in our newer version of VIAP that we entered in this year's competition: by having a more powerful system for computing closed-form solutions of recurrences, the new system becomes more efficient and can prove many properties that our

[3] https://github.com/VerifierIntegerAssignment/VIAP_ARRAY/blob/master/Document/TranslatonInsRules.pdf.

previous system were not able to. However, VIAP provides little or no support for translation and reasoning about dynamic linked data structures or programs with floating points. We are working in the direction to strengthen our front- and backhand to handle all types of the program so that we can participate in all the sub-categories of ReachSafety in the future edition of SV-COMP. The SVCOMP'19 results show that VIAP can effectively verify a number C programs from those categories. VIAP came in first in the ReachSafety-Arrays and ReachSafety-Recursive sub-category. The major disadvantage of the method which translates loop body to the recurrence relation is that if they failed to find closed form solution, then they unable to find suitable invariant as a result they failed to complete the proof. When VIAP fails to come up with a closed-form solution, it falls back to simple induction using Z3. There is clearly a need of better way to do induction and we are working on it. In terms of closed-form solution, in general it is undecidable whether a recurrence has a closed-form solution or not.

4 Tool Setup and Configuration

The version of VIAP (version 1.1) submitted to SV-COMP 2019[4] is provided as a set of binaries and libraries for Linux x86-64 architecture. The options for running the tool are:

```
./viap_tool.py  --spec=SPEC INPUT
```

SPEC is the property file, and INPUT is a C file. The output of VIAP is "VIAP_OUTPUT_True" when the program is safe. When a counterexample is found, it outputs "VIAP_OUTPUT_False" and a file named *errorWitness.graphml* that contains the witness of error-path is generated in the VIAP root folder. If VIAP is unable find any result it outputs "UNKNOWN".

5 Software Project and Contributors

VIAP is an open-source project, mainly developed by Pritom Rajkhowa and Professor Fangzhen Lin of the Hong Kong University of Science and Technology. We are grateful to the developers of Z3 and *SymPy* for making their systems available for open use.

Acknowledgments. We are very thankful to the anonymous reviewers for their helpful comments on an earlier version of this paper.

[4] https://gitlab.com/sosy-lab/sv-comp/archives-2019/blob/master/2019/viap.zip.

References

1. Lin, F.: A formalization of programs in first-order logic with a discrete linear order. Artif. Intell. **235**, 1–25 (2016)
2. Rajkhowa, P., Lin, F.: VIAP - automated system for verifying integer assignment programs with loops. In: Jebelean, T., Negru, V., Petcu, D., Zaharie, D., Ida, T., Watt, S.M. (eds.) 19th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC 2017, Timisoara, Romania, 21–24 September 2017, pp. 137–144. IEEE Computer Society (2017)
3. Rajkhowa, P., Lin, F.: Extending VIAP to handle array programs. In: Piskac, R., Rümmer, P. (eds.) VSTTE 2018. LNCS, vol. 11294, pp. 38–49. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-03592-1_3
4. SymPy Development Team: SymPy: python library for symbolic mathematics (2016)
5. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24