VIAP 1.1
(Competition Contribution)

Pritom Rajkhowa and Fangzhen Lin

Department of Computer Science
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 verification of safety properties of procedural programs with integer assignments and loops. It translates a given program to a set of first-order axioms with quantification over natural numbers. VIAP is integrated with newly developed recurrence solver functionality which enables it to verify several programs that were previously out of reach for VIAP. VIAP uses SMT solver *z3* as the backhand theorem prover.

1 Introduction

VIAP is a fully automatic program verifier designed for imperative programs with loop proving the correctness of these programs with respect to specifications given as program assertions and assumptions. VIAP first translates the program to a set of first-order axioms with natural number quantification using an algorithm proposed by Lin [1]. It is integrated with our novel recurrence solver mechanism which finds the closed-form solutions of inductive definitions in Lin's translation. The recurrence solver mechanism also provides support to find closed-form solutions of some classes of conditional and mutual recurrences. Due to the addition of the recurrence solver to our verification system, it can solve many more benchmarks that were previously out of its reach. The theorem proving part uses z3[2] directly when the translated axioms have no inductive definitions. Otherwise, it tries a simple inductive proof using z3[2] as the base theorem prover. The unique point of VIAP is that it can prove (partial) correctness of programs without needing intermediate loop invariants.

To illustrate how our system works, consider the simple program below needing intermediate loop invariants:

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

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

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

where 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 nth 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 recurrences in the above axioms. Both the recurrences are passed to the recurrence solver mechanism. 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 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 recurrence solver mechanism tries to solve the above simplified conditional recurrences, and returns the following closed-form solution:

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

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

$$
\begin{aligned}
&x_1 = N \wedge y_1 = ite(N < 50, N, 100 - N),\\
&N \geq 100),\\
&\forall n.n < N \rightarrow n < 100,
\end{aligned}
$$

With this set of axioms, SMT solvers like z3 can then be made to prove assertions like $y_1 == 0$. It takes a fraction of a second to return the results. Similarly, when an assertion like $y_1 == 1$ 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 counter example, 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 recurrences 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[3]).

Translation Our system supports the full C99 language (according to the standard ISO/IEC 9899). 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 set $\Pi_P^{\boldsymbol{X}}$ of axioms are generated inductively on the structure of P. The algorithm is described in detail in [1] and an implementation is [3]. The inductive cases are given in the table provided in the supplementary information depicted in [1]. We have extended our translation for the array which is provided in detail in[4].

Recurrence Solver (RS) The main objective of this module is to nd closed form solutions of recurrence relations generated from the translation of the loop body. Our recurrence solver takes a set of recurrences and other constraints, and returns a set of closed-form solutions it found for some of the recurrences and the remaining recurrences and constraints simplified using the computed closed-form solutions. It uses SymPy [5](Version 1.1.1) as the base solver. It also uses z3 extensively. It can nd the closed form solutions of a variety of recurrence equations including conditional and mutual recurrences [2].

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. Whenever an array element assignment occurs inside a loop, our system will generate an axiom like the following:

$$\begin{aligned} &\forall x_1, x_2 ... x_{k+1}, n. d\text{k}array_i(x_1, x_2 ... x_{k+1}, n+1) = \\ &\quad ite(x_1 = A \wedge x_2 = E_2 \wedge ... \wedge x_{k+1} = E_{h+1}, E, \\ &\qquad d\text{k}array_i(x_1, x_2 ... x_{k+1}, n)) \end{aligned} \tag{1}$$

where

– A is a k-dimensional array,
– $d\text{k}array_i$ is a temporary function introduced by the translator,
– x_1 is an array name variable introduced by the translator universally quantified over arrays of k dimensions,

[1] `https://goo.gl/2ZBGUr`
[2] https://github.com/VerifierIntegerAssignment/recSolver

– x_2,....,x_{k+1} are natural number variables representing array indices universally quantified over natural numbers,
– n is the loop counter variable universally quantified over natural numbers,
– $E, E_2, ..., E_{k+1}$ are expressions.

For an axiom like (1), our system performs two types of instantiations:

– **Instantiating arrays**: this substitutes each occurrence of variable x_1 in the axiom (1) by the array constant A, and generates the following axiom:

$$\forall x_1, x_2...x_{k+1}.dk array_i(A, x_2...x_{k+1}, n+1) = \\ ite(x_2 = E_2 \wedge ... \wedge x_{k+1} = E_{h+1}, E, dk array_i(A, x_2...x_{k+1}, n)) \quad (2)$$

– **Instantiating array indices**: This substitutes each occurrence of the variables x_i, $2 \leq i \leq k$ in the axiom (2) by E_i, and generates the following axiom:

$$\forall n.dk array_i(A, E_2...E_{k+1}, n+1) = E \quad (3)$$

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 a formula $\alpha \wedge \Pi_P^{\boldsymbol{X}} \models \beta$ in a background theory, T, is valid, we show that $\alpha \wedge \Pi_P^{\boldsymbol{X}} \wedge \neg\beta$ is T-unsatisable. In VIAP, we implemented two different strategies whose details can be found in [3].

3 Strength and Weaknesses

VIAP supports user assertions, including reachability of labels in the C-code. In SV-COMP 2018, these checks are only enabled for the ReachSafety category, specifically ReachSafety-Arrays, ReachSafety-Loops and ReachSafety-Recursive sub-categories. VIAP participated only in Arrays, Loops and Recursive subcategories of ReachSafety. However, VIAP provides little or no support for reasoning about dynamic linked data structures or programs with floating points. The SV-COMP18 results show that VIAP can be effectively verified on a number C programs from those categories. VIAP with array extensions came in second in the ReachSafety-Arrays sub-category behind *VeriAbs*.

Most formal verification tools for non-parallel imperative computer programs rely on having a suitable loop invariant to reason about the correctness of a given loop. VIAP does not require loop-invariants to verify the correctness of programs. It does not produce a witness for a successful proof that the program satisfies the stated assertions. This is because our system relies almost entirely on z3 to prove the assertions. It does not attempt to compute or discover loop invariants; instead, it translates programs to a set of inductive equations and first-order constraints about loops. Besides z3, it only makes use of the algebraic system *SymPy* for simplifying algebraic expressions and solving some simple recurrences. It also sometimes uses a simple mathematical induction on natural numbers. As far as we know, there is no simple solution to the problem of computing a "witness" when an automated theorem prover like z3 claims that a proof is found.

4 Tool Setup and Configuration

The version of VIAP (version 1.1) submitted to SV-COMP 2019 can be downloaded at:

`https://verifierintegerassignment.github.io/archive/2019/viap.zip`

VIAP 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 counter example 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.

References

1. F. Lin, "A formalization of programs in first-order logic with a discrete linear order," *Artificial Intelligence*, vol. 235, pp. 1 – 25, 2016.
2. L.de.Moura and N.Bjorner, "The z3 smt solver." http://research.microsoft.com/en-us/um/redmond/projects/z3/, 2012.
3. P. Rajkhowa and F. Lin, "Viap - automated system for verifying integer assignment programs with loops," in *19th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC 2016, Timisoara, Romania, September 21-24, 2017*, 2017.
4. P. Rajkhowa and F. Lin, "Extending viap to handle array programs," in *10th Working Conference on Verified Software: Theories, Tools, and Experiments, VSTTE 2018, Oxford, UK, July 18-19*, 2018.
5. SymPy Development Team, *SymPy: Python library for symbolic mathematics*, 2016.