VIAP - Automated System for Verifying Integer Assignment Programs with Loops

Pritom Rajkhowa* and Fangzhen Lin†
Department of Computer Science
The Hong Kong University of Science and Technology
Clear Water Bay, Kowloon, Hong Kong
Emails: *prajkhowa@cse.ust.hk, †flin@cse.ust.hk

***Abstract*—This paper describes an automated system, VIAP, for proving the correctness of procedural programs with integer assignments and loops. VIAP does not require loop-invariants to verify the correctness of programs. It also includes functionalities resulting from the extension of existing work about program translation into FOL by solving recurrences generated during translation. This system is also able to prove partial correctness of programs, given precondition(s) and postcondition(s), without using loop invariants. We validate these claims by showing that VIAP can successfully prove the benchmark programs used by invariant-generating tools in literature for their validation, without the requirement of the generation of these loop-invariants. The system is fully automatic and points to a new way of proving properties of programs.**

***Index Terms*—Automatic Program Verification; Preconditions; Postconditions; First-Order Logic; Mathematical Induction; Recurrences; SMT; Arithmetic**

I. Introduction

Program verification has been a key problem in computer science since its beginning. Various approaches have been proposed for various types of programs. This paper considers a class of imperative programs composed of integer assignments, sequences, conditionals and loops. It describes the theory and implementation of a fully automated system we developed for proving the correctness of these programs with respect to specifications given by a pair of precondition and postcondition.

We call our system Verifier for Integer Assignment Programs (VIAP). Given a program with no annotations, and a proof obligation in the form of precondition(s) and postcondition(s), VIAP first translates the program to a set of first-order axioms with natural number quantifications. This part is independent of the proof obligation. It then attempts to prove postcondition(s) from translated axioms, and precondition(s) using SMT solver Z3 [1] as the basic theorem prover.

The translation part is based on Lin [2]. We improved it by using sympy [3] and Mathematica [4] to try to compute closed-form solutions of inductive definitions in the translation. The theorem proving part uses Z3 directly when the translated axioms have no inductive definitions. Otherwise, it tries a simple inductive proof using Z3 as the base theorem prover. To illustrate how our system works, consider the following program for computing the integer square root of a given non-negative integer X:

```
a=0;  su=1;  t=1;
while ( su <= X ) {
  a=a+1;  t=t+2;  su=su+t; }
return a;
```

With some simplification, the translation outlined in [2] would generate the following axioms:

$$
\begin{aligned}
&X_1 = X,\\
&a_1 = a_3(N), \quad a_3(0) = 0,\\
&a_3(n+1) = a_3(n) + 1,\\
&t_1 = t_3(N), \quad t_3(0) = 1,\\
&t_3(n+1) = t_3(n) + 2,\\
&su_1 = su_3(N), \quad su_3(0) = 1,\\
&su_3(n+1) = su_3(n) + t_3(n) + 2,\\
&\neg(su_3(N) \leq X),\\
&(n < N) \rightarrow (su_3(n) \leq X),
\end{aligned}
$$

where a denotes the input of the program variable `a`, a_1 is the output, and $a_3(n)$ is a temporary function denoting the value of `a` after the nth iteration of the while loop. Similar conventions apply to X, t, su and their subscripted versions. The constant N is introduced to denote the number of iterations of the while loop before it exits. The variable n ranges over natural numbers and is universally quantified.

By using sympy and if necessary, Mathematica, our translator computes closed-form solutions for $a_3()$, $t_3()$, and $su_3()$, eliminates them, and produces the following axioms:

$$
\begin{aligned}
&X_1 = X, \quad a_1 = N, \quad t_1 = 2N + 1,\\
&su_1 = N^2 + 2N + 1,\\
&\neg(N^2 + 2N + 1 \leq X),\\
&(n < N) \rightarrow (n^2 + 2n + 1 \leq X).
\end{aligned}
$$

With this set of axioms, Z3 can then be made to prove the following two postconditions

$$(a_1 + 1)^2 > X, \quad a_1^2 \leq X$$

given the precondition $X \geq 0$.

This is an example where all recurrences happen to have closed-form solutions that can be computed by both Mathematica and sympy. Sometimes there are recurrences that can

be solved by Mathematica but not by sympy and vice versa. More often, they cannot be solved by either of them. When this happens, the proof cannot be proved directly by Z3 as it needs to use mathematical induction which is not inbuilt in Z3. There has been work on extending Z3 with induction [5], and more recently extending CVC4 with induction [6]. For now our system implements a simple induction scheme.

The main contribution of the paper:

- VIAP implements and extends the translation proposed in [2] by solving recurrences generated by the translation.
- VIAP automatically proves the partial correctness of a program given precondition(s) and post-condition(s) by mathematical induction with Z3 as the base prover without using loop invariants.
- VIAP can prove the correctness of programs without generating loop invariants, unlike existing work in literature. There are many benchmark programs like [7] [8] [9] [10] [11] which are used to validate invariant generation tools in literature. We have shown that VIAP can also prove these benchmark programs without requiring to go through the process of loop-invariant generation. Additionally, we also included number of SV-COMP benchmark program in our experiment - the details are given in the experiment section later in the paper.

We will discuss related work later in a separate section. We would like to mention that for these benchmark programs, our system works without explicit loop invariants. Many of the programs that we tried with our system are benchmark programs for loop invariant discovery systems. Now our system can prove the (partial) correctness of these programs without needing these loop invariants.

The rest of this paper is organized as follows. We describe how our system translates programs to first-order logic in more details. This is followed by the description of how we use Z3 for program verification using the translated axioms, and define two proof strategies. We next provide the experimental results of running our system on 29 programs, many of them from work by others, and some classic arithmetic algorithms, such as factorial, power, and product are included to illustrate certain features of our system. We also ran Smack [12], Seahorn [13], Leon [10], 2LS [14] and UAutomizer [15] on the same programs We then describe possible uses of our system for proving properties about program termination, discuss more related work and finally conclude the paper.

II. Translation

Our system handles programs consisting of integer assignments, sequence, conditionals and while loops:

```
<P> ::= <X> = <E> |
        <P>; <P> |
        if <C> then <P> else <P> |
        while <C> {<P>}
```

where <X> is an integer variable, <E> an integer expression and <C> a boolean condition.

Our translation is based on the one described in [2]. It translates a program to a set of first-order axioms by introducing for each loop a natural number term that denotes the number of iterations the loop ran before it exits. For a while loop like `while C { B }`, it generates the following two formulas:

$$\neg C(N), \tag{1}$$

$$\forall n.n < N \rightarrow C(n) \tag{2}$$

where N is a new natural number constant denoting the number of iterations that the loop ran before it exits, and $C(n)$ a formula denoting the true value of the condition C at the end of the nth iteration of the loop.

We will not give the details of the translation here but illustrate it by examples and show how we improve on it. Consider the following algorithm for computing the integer division by Cohen [16] (the one below is from [2] which is adapted from [17]). It has two loops, one nested inside another:

```
// X and Y are two input integers;
// X>=0 Y > 0
Q=0;  R=X;
while (R >= Y) do {
  A=1;  B=Y;
  while (R >= 2*B) do {
    A = 2*A;
    B = 2*B;
  }
  R = R-B;
  Q = Q+A
} // return Q = X/Y;
```

Again for variable V, use V to denote its input value, V_1 its output value, and V_2, V_3, ... its temporary value during program execution, our translator outputs the following axioms:

$$Y_1 = Y, \tag{3}$$

$$X_1 = X, \tag{4}$$

$$A_1 = A_7(N_2), \tag{5}$$

$$q_1 = q_7(N_2), \tag{6}$$

$$r_1 = r_7(N_2), \tag{7}$$

$$B_1 = B_7(N_2), \tag{8}$$

$$r_7(n) < 2 \times 2^{N_1(n)} \times Y, \tag{9}$$

$$n_1 < N_1(n_2) \rightarrow r_7(n_2) \geq 2 \times 2^{n_1} \times Y, \tag{10}$$

$$A_7(n+1) = 2^{N_1(n)} \times 1, \tag{11}$$

$$B_7(n+1) = 2^{N_1(n)} \times Y, \tag{12}$$

$$q_7(n+1) = q_7(n) + 2^{N_1(n)} \times 1, \tag{13}$$

$$r_7(n+1) = r_7(n) - 2^{N_1(n)} \times Y, \tag{14}$$

$$A_7(0) = A, \tag{15}$$

$$B_7(0) = B, \tag{16}$$

$$q_7(0) = 0, \tag{17}$$

$$r_7(0) = X, \tag{18}$$

$$r_7(N_2) < Y, \tag{19}$$

$$n < N_2 \rightarrow r_7(n) \geq Y \tag{20}$$

where

- N_2 is a system generated natural number constant denoting the number of iterations that the outer loop runs before exiting;
- N_1 is a system generated natural number function denoting for each k, $N_1(k)$ is the number of iterations that the inner loop runs during the kth iteration of the outer loop;
- n, n_1 and n_2 are universally quantified natural number variables.
- r_7 and others are system generated temporary constants and functions denoting the values of the corresponding program variables during program execution. For this program, for each variable, say r, our translator generates six temporary ones $r_2, ..., r_7$ and eliminate all of them except the last one.

We use sympy to simplify algebraic equations and solve recurrences, and use Mathematica as backup for solving recurrences when sympy could not. For this program, the inner loop generates two recurrences, one for A and the other for B:

$$A_2(n+1) = 2 \times A_2(n), \quad A_2(0) = A,$$
$$B_2(n1+1) = 2 \times B_2(n), \quad B_2(0) = B.$$

sympy solves both, and are replaced by their closed form solutions:

$$A_2(n) = 2^n \times A, \quad B_2(n) = 2^n \times B.$$

This is the reason why we have (11) and (12).

III. VIAP ARCHITECTURE

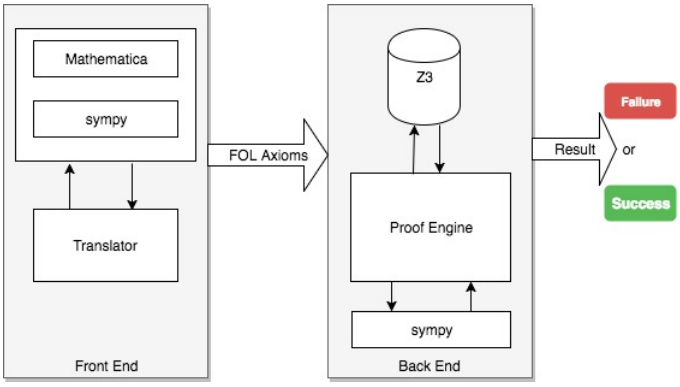

Fig. 1. Architecture Overview of VIAP

VIAP is implemented in python. The overall approach is illustrated in Figure 1. VIAP has been developed in a modular fashion and its architecture is layered in to two parts:

- **Front-End** : The system accepts a program (e.g. Java) as input and translates it to first order axioms. Mathematica and sympy solves the recurrences generated during the translation if closed form solutions are available.
- **Back-End:** : Takes the set of translated first order axioms and translates all the axioms to $Z3$ compatible equation by pre-processing them using sympy. Then the proof engine applies different strategies and tries to prove post-conditions in $Z3$. More details are discussed in Section IV)

IV. PROOF STRATEGIES

We use Z3 [1] to prove properties about a program with the axioms generated by our translator. Given that Z3 accepts first-order axioms, this seems like a straightforward task. The problem is of course that theorem proving in first-order logic is undecidable, and there are many limitations in what Z3 can prove. We describe below the strategies that we use to make Z3 work for us.

First, we use a natural number sort, but Z3 has only integer sort. So whenever we need to quantify over a natural number variable, we need to make it into a quantification on non-negative integers. For example, to encode in Z3 the following axiom that universally quantifies natural number variable n:

$$n < N_2 \rightarrow r_7(n) \geq Y$$

we use an integer type variable n and add the condition $n \geq 0$:

```
n=Int('n')
ForAll([n], Implies(n>=0, Implies(n<N2,
r7(n)>=Y)))
```

Another issue is that many of the benchmark programs that we tried have exponentiation. We found that Z3's builtin module NLSat (nonlinear real arithmetic) was too weak most of the time. Instead, we define our own exponentiation function $power(x, y)$ on integers:

```
power=Function('power',IntSort(),IntSort()
,IntSort())
```

and include the following axioms:

```
ForAll([n],Implies(n>=0, power(0,n)==0))
ForAll([n,m],Implies(power(m,n)==0,m==0))
ForAll([n],Implies(n>0, power(n,0)==1))
ForAll([n,m],Implies(power(n,m)==1,
Or(n==1,m==0)))
ForAll([n,m],Implies(And(n>0,m>=0),
power(n,m+1)==power(n,m)*n)))
```

Of course, these axioms are not complete. But so far they work better for us than Z3's inbuilt module.

Yet another issue that we had with Z3 was that it has difficulty with axioms of the form (2), which is crucial for our axiomatization of loops. Our strategy is to add the following consequence of (2) to the axiom set:

$$N = 0 \vee C(N-1). \tag{21}$$

There are some other preprocessings that we have done while translating the axioms to Z3. Chiefly, we make use of sympy's expand and simplify functions to convert arithmetic terms to a standard form. For example, a term like $(x+1)^2+x$ is converted to x^2+3x+1, and exponentiations are simplified by grouping those with the same base, for example, by converting $x^y \times x^z$ to x^{y+z}.

Finally, since Z3 does not have a build-in mechanism for induction, we implemented a simple induction scheme on top of it. In the end, we came up with two proof strategies: direct proof and proof by simple induction.

A. Strategy 1: direct proof

Input A - a set of first-order axioms; α - precondition (a sentence on the input values of program variables); β - postcondition (a sentence on the input and output values of program variables);

Output "proved", "failed to prove", or "refuted" (with a counter-example);

1. Declare all variables in A, α and β as integer variables;
2. Add binary integer function `power(x,y)` and its associate axioms;
3. Convert α and each axiom in A to Z3 format (after some pre-processing by sympy);
4. For each axiom in A of the form (2), add (21);
5. For each output variable `X1` in β, match it with an equation of the form `X1 = W` in A, and replace it in β by `W`. Let the resulting sentence be β'. Convert $\neg\beta'$ to Z3 format.
6. Query Z3 with the axioms thus generated and output "proved" if it returns "unsat", "refuted" (with a counter-example) if it returns the counter-example, and "failed to prove" otherwise.

B. Strategy 2: simple induction

First 4 steps of this strategy are the same as the first one:

Input A - a set of first-order axioms; α - precondition (a sentence on the input values of program variables); β - postcondition (a sentence on the input and output values of program variables);

Output "proved", "failed to prove", or "refuted" (with a counter-example);

1. Declare all variables in A, α and β as integer variables;
2. Add binary integer function `power(x,y)` and its associate axioms;
3. Convert α and each axiom in A to Z3 format (after some pre-processing by sympy);
4. For each axiom in A of the form (2), add (21);
5. Let Q be the set of Z3 axioms generated so far.
6. For each output variable `X1` in β, match it with an equation of the form `X1 = W` in A, and replace it in β by `W`. Let the resulting sentence be β'. Find in β' a system generated natural number constant N denoting the number of iterations in a loop. The following steps try to prove $\forall n.\beta'(N/n)$ by induction on n.
7. Replace N in β' by 0. Let β'' be the resulting sentence. Convert $\neg\beta''$ to Z3 axioms and add it to Q.
8. Query Z3 with the axioms generated. If it returns "unsat", then continue. Otherwise, return "failed to prove".
9. Choose a new symbol k not occurring in Q or β', and declare it to be a non-negative integer.
10. Replace N in β' by k to get a new sentence β_1. Replace N in β' by $k+1$ to get another sentence β_2.
11. For each term in β_2 of the form $f(k+1)$, match it with an equation in A of the form $\forall n.f(n+1) = W(n)$, and replace $f(k+1)$ in β_2 by $W(k)$. Let the resulting sentence be β_3. Convert $\neg(\beta_1 \rightarrow \beta_3)$ to Z3 format and add it to Q.
12. Query Z3 with the axioms thus generated. If it returns "unsat", then return "proved". Otherwise, return "failed to prove".

As an illustration, consider proving the following postcondition β (recall that q_1 and r_1 denote the output values of q and r, respectively):

$$X = q_1 Y + r_1 \tag{22}$$

under the precondition α:

$$X \geq 0 \wedge Y > 0$$

for the above Cohen's division program. First VIAP converts the postcondition into the following assertion using the equations about q_1 and p_1 in the set of axioms:

$$X = q_7(N_2)Y + r_7(N_2).$$

Then it tries to prove it directly using Z3, but fails. So it tries to prove the following more general one

$$\forall n.X = q_7(n)Y + r_7(n) \tag{23}$$

by induction on n, which succeeds.

V. Experiments

A. Verifiability

VIAP is implemented in python. The source code and the full experiments are available on

https://github.com/VerifierIntegerAssignment/VIAP

VIAP sometimes calls Mathematica(Wolfram Alpha) to solve a recurrence. When this happens, a working internet connection is needed. Table I is a summary of our experiments.We discuss them in more details below.

B. Benchmarks

We considered 29 programs, most of them are from or adapted from E. Rodriguez-Carbonell and Kapur's NLA test suite for their work [7] [8] [9]. In particular, the 10 power series programs `potSumm1` - `potSumm10` are adapted from the NLA test suite. These are programs for computing the power sum $\sum_{n=1}^{X} n^i$ for $i = 1, ..., 10$. As shown in Table I, for these programs, the postconditions are the closed-form formulas of the corresponding power sum. For example, for `potSumm2`, the postcondition is that the final value of `sum` will be `X*(X+1)*(2*X+1)/6`, which is a closed-form formula for $\sum_{n=1}^{X} n^2$. These programs demonstrate the capability of VIAP's recurrence solving module to find closed form solution of the polynomial recurrences of various orders. They also demonstrate the effectiveness of the system to prove the postconditions involving polynomial of various orders.

Two of the programs (`divDafny` and `addDafny`) are from Dafny presented at VSTTE 2008[11], and one (`abs`) from Leon [10].

The factorial program (`fact`) is used to demonstrate that our system can handle pre-defined functions, including finding closed-form solutions using the pre-defined functions in sympy and mathematica. The tower of Hanoi program (`hanoi`) and the exponentiation program (`power`) are used to demonstrate how our system can handle postcondition involving polynomials with arbitrary exponents.

C. Experimental results

Our experiments were done on a PC and 4GB of RAM running Windows 7. The system was run on Python 2.7.11 with sympy 1.0, Mathematica(Wolfram Alpha) 2.4, and Z3 4.4.1. Both sympy and Z3 are installed locally. However, Wolfram Alpha is a service from Wolfram Research and called remotely. Not all programs need Wolfram Alpha. For those that do need it, they can only be translated if there is a working internet connection to the server of Wolfram Alpha.

For Z3, we set its timeout parameter to 60 seconds. We included two sets of times in Table I: the t_1 column are the times for VIAP to translate the given program to axioms, and t_2 the times for it to return the proofs for the given verification tasks. Both are in CPU seconds. Notice that t_1 includes the times for calling sympy and/or Mathematica to solve recurrences and is independent of the verification tasks. The translation needs to be done only once. On the other hand, t_2 is specific to the given verification task in terms of a precondition and a postcondition. It includes time to process the two conditions as well as the Z3 query time.

We have also included some tasks that VIAP is not able to prove currently. For example, it was not able to prove the postcondition `r1<Y` (the remainder is smaller than the divisor) for the program divHard (an integer division program). Given the translated axioms for the program, VIAP converts the postcondition to `r7(_N2)<Y`. The direct proof strategy could not prove it as Z3 was not powerful enough for it. Our simple induction strategy would try to prove it by proving that for all natural number n, `r7(n)<Y` holds. But this cannot be proved as even the case, `r7(0)<Y` may not be true. We have considered a number of programs from SV-COMP benchmark in our experiments to present the capabilities of the tool to handle a broader range of programs. All the experiments' output are available in

https://github.com/VerifierIntegerAssignment/VIAP/tree/master/Experimental_Data/SV-COMP

, and thus are not included in Table I, but readers are requested to check these out in the GITHUB link provided.

D. A comparative study with other automatic verifiers

As an illustration of how VIAP compares with some of the automatic program verification systems, we ran Smack [12], Seahorn [13],Leon [10], 2LS [14], and UAutomizer [15] on the same set of programs. Here, the phrase "automated system" means a system that verifies a program with only preconditions and postconditions given as user input written in some programming language(s). There exist verifier tools like Vampire [18] and Valigator[19], where a user needs to convert program code to an intermediate format for verification instead of directly providing the source file. Thus, these are not considered to be automated verifiers. Thus, we are not comparing VIAP with Valigator and Vampire, and all the comparisons of VIAP are done against automated verifier systems.

We also have not considered InvGen [20] for comparative study this tool is not fully automatic program verifier, as it requires explicit templates for the generation of invariants.

The last four columns of Table I summarize the results of these systems. The full experimental results can be found in the following

https://github.com/VerifierIntegerAssignment/VIAP/tree/master/Experimental_Data.

We briefly describe these systems below.

$Leon$ [10] started as a verification system for the pure functional subset of Scala. Later it was extended to include imperative programs by translating them internally into pure functional programs. In our experiments, we use Leon 3.0, the latest version available on [1] at the time of writing. Leon also uses Z3.

$Smack$ [12] is a self contained software verifier and also a modular software verification tool chain. In our experiments, we use the latest version available on [2] at the time of writing. It got second the highest overall score to win silver medal in the SV-COMP'2017 competition

$Seahorn$ [13] is a fully automated analysis framework which compiles C program to LLVM intermediate representation (LLVM IR). Constrained Horn Clauses (CHC) are generated from optimized bit-code. The generated CHC are used for the analysis. The SeaHorn uses SMT-based model checking engines based on PDR engine Z3 as a back end. In our experiment, we use the latest version of the tool which is available on [3].

$2LS$ [14] is a static analysis and verification tool for C program which also performed well in the Loop subcategory of the Integers and Control Flow category in the SV-COMP'2016 competition. It is built upon the CPROVER infrastructure which implements incremental bounded model checking, invariant generation techniques and k-induction. In our experiment, we use 2LS-0.3 as submitted to SV-COMP'2016[4]. 2LS uses MiniSAT 2.2.1 [21].

$UAutomizer$ [15] is a model checker (that implements an approach based on automata) for C program which proves the correctness of the program by computing inductive invariants from interpolation. It got the highest overall score to get the gold medal in the SV-COMP'2017 competition.

On some tasks, Leon returned counter-examples. These are marked by "?" in Table I. For example, for the program `abs`:

[1] http://lara.epfl.ch/w/leon
[2] http://smackers.github.io/
[3] http://seahorn.github.io/
[4] http://www.cprover.org/svn/deltacheck/releases/2ls-0.3-sv-comp-2016/

TABLE I
EXPERIMENTAL RESULTS

Program	Desc	Postcondition	VIAP			T_1	T_2	T_3	T_4	T_5
			t_1 (s)	Strategy	t_2 (s)					
abs	absolute value [10]	`a1>=0`	< 1	✓(Direct)	< 1	?	✓	✓	×	×
sqrt	squre root [7] [8] [9]	`(a1+1)**2>X` `a1**2<=X`	1	✓(Direct) ✓(Direct)	1 1	× ×	× ×	× ×	× ×	× ×
fact	factorial	`F1==factorial(X)`	9	✓(Direct)	< 1	×	✓	×	×	×
product	multiply two numbers	`A1==a*b`	< 1	✓(Direct)	< 1	×	✓	×	×	×
power	exponential	`P1==a**b`	< 1	✓(Direct)	1	×	✓	×	×	×
square	square of a number	`s1==M**2`	< 1	✓(Direct)	2	×	✓	×	×	×
cubeCohen	cube [7] [8] [9]	`m1==X**3`	6	✓(Direct)	1	×	✓	×	×	×
hanoi	Tower of Hanoi	`h1==2**x-1`	< 1	✓(Direct)	1	×	×	×	×	×
subDafny	subtract two numbers	`r1==X-Y`	7	✓(Direct)	< 1	×	✓	×	×	×
addDafny	add two numbers [11]	`r1==X+Y`	7	✓(Direct)	< 1	×	×	×	×	×
potSumm1	geo series [7] [8] [9]	`sum1==X*(X+1)/2`	< 1	✓(Direct)	10	×	×	×	×	×
potSumm2	geo series [7] [8] [9]	`sum1==X*(X+1)*(2*X+1)/6`	13	✓(Direct)	15	×	×	×	×	×
potSumm3	geo series [7] [8] [9]	`sum1==X*(X+1)*X*(X+1)/4`	16	✓(Direct)	20	×	×	×	×	×
potSumm4	geo series [7] [8] [9]	`sum1==`α_4	18	✓(Direct)	16	×	×	×	×	×
potSumm5	geo series [7] [8] [9]	`sum1==`α_5	24	✓(Direct)	19	×	×	×	×	×
potSumm6	geo series	`sum1==`α_6	27	✓(Direct)	20	×	×	×	×	×
potSumm7	geo series	`sum1==`α_7	37	✓(Direct)	18	×	×	×	×	×
potSumm8	geo series	`sum1==`α_8	46	✓(Direct)	24	×	×	×	×	×
potSumm9	geo series	`sum1==`α_9	55	✓(Direct)	23	×	×	×	×	×
potSumm10	geo series	`sum1==`α_{10}	71	✓(Direct)	25	×	×	×	×	×
sumOfEven	geo series	`sum1==X*(X+1)`	< 1	✓(Direct)	1	×	×	×	×	×
sumOfOdd	geo series	`sum1==X*X`	< 1	✓(Direct)	1	×	×	×	×	×
geoReihe1	power series [7] [8] [9]	`m1==((Z**K-1)/(Z-1))*(Z-1)`	1	✓(Direct)	3	?	×	×	×	×
geoReihe2	power series [7] [8] [9]	`m1==(Z**K-1)/(Z-1)`	< 1	✓(Direct)	1	?	×	×	×	×
geoReihe3	power series [7] [8] [9]	`m1==((Z**K-1)/(Z-1))*a`	< 1	✓(Direct)	8	?	×	×	×	×

divDafny	int div [11]	`X==Y*q1+r1`
		`r1<Y`
		`r1>=0`
			< 1	✓(Direct)
				✓(Direct)
				✓(Direct)
					< 1
					< 1
					< 1
						×
						✓
						×
							✓
							×
							✓
								✓
								✓
								×
									×
									✓
									×
										×
										✓
										✓

divCohen	int div [7] [8] [9]	`X==Y*q1+r1`
		`r1<Y`
		`r1>=0`
			2	✓(Induction)
				✓(Direct)
				✓(Induction)
					115
					2
					3
						×
						✓
						?
							✓
							✓
							✓
								×
								×
								×
									×
									✓
									×
										×
										✓
										✓

divHard	int div [7] [8] [9]	`X==Y*q1+r1`
		`r1<Y`
		`r1>=0`
			< 1	✓(Induction)
				×
				✓(Induction)
					143
					-
					127
						?
						×
						×
							×
							✓
							×
								✓
								✓
								×
									×
									×
									×
										×
										×
										✓

divKaldewaij	int div [7] [8] [9]	`A==B*q1+r1`
		`r1<B`
		`r1>=0`
			< 1	×
				×
				✓(Induction)
					×
					-
					120
						-
						×
						×
							×
							✓
							×
								✓
								✓
								×
									×
									×
									×
										×
										×
										✓

Legends:
✓: proved; ×: failed to prove; ?: Leon gave counter examples (see the text for discussion);
T_1: Leon; T_1: Leon; T_2: smack; T_3: 2LS; T_4: UAutomizer
t_1 is the time (in miliseconds) for translating the program to axioms;
t_2 is the time (in miliseconds) for proving the postcondition using the axioms;
α_4 is `(6*X**5+15*X**4+10*X**3-X)/30`;
α_5 is `(2*X**6+6*X**5+5*X**4-X**2)/12`;
α_6 is `(6*X**7+21*X**6+21*X**5-7*X**3+X)/42`;
α_7 is `(3*X**8+12*X**7+14*X**6-7*X**4+2*X**2)/24`;
α_8 is `(10*X**9+45*X**8+60*X**7-42*X**5+20*X**3-3*X)/90`;
α_9 is `(2*X**10+10*X**9+15*X**8-14*X**6+10*X**4-3*X**2)/20`;
α_{10} is `(6*X**11+33*X**10+55*X**9-66*X**7+66*X**5-33*X**3+5*X)/66`.

```
object abs {
  def abs(X: Int): Int = {
    if(X <= 0) -X else X
  } ensuring(res => res >= 0)
}
```

Leon returned X = -2147483648 as a counter-example. The reason is that Leon assumes 32-bit representation of integers: $2^{31} = 2147483648$. Indeed, once we restrict the input X by the following condition:

```
require(-2147483648<X && X <2147483647)
```

Leon was able to prove the post-condition. Smack and Seahorn were able to prove the post-condition with any restriction. Although 2LS and UAutomizer did not return counter examples, they succeeded under the restricted input in precondition like Leon. The reason is that Leon, 2LS, Smack and UAutomizer assume either a 32-bit or 64-bit architecture memory model. In contrast, Dafny [22],seahorn [13] and our system VIAP assume an "ideal" integer model without any bounds. Thus in our model, for example, $X^2 \geq 0$ will be true for all integers, X. In fact, we inherit this model from Z3's built in theory for arithmetic. We also implicitly assume this by making use of sympy and Mathematica.

One could argue that it is not meaningful to compare systems that assume an infinite integer model with those that assume a finite integer model. However, there may be some interesting relationships between the two models. For instance, one way of using a system like ours for a finite integer model is to add a separate step of verifying that there is no integer overflow exception during the execution. Among the programs in our experiments, this is the case for `divDafny`, `divCohen`, `divHard` and `divKaldewaij`. Thus once they are verified under an ideal integer model, they work for all finite integer representations.

VI. Partial correctness and total correctness

Typically VIAP proves only partial correctness of a program with respect to a precondition and a post-condition. This is because by introducing a natural number term to denote the number of iterations that a loop ran before it exits, we assume that the loop terminates. Otherwise, a contradiction will arise.

Consider the following program:

```
while (X != 1) {
  if (X==0) { X=1; }
}
```

Clearly the program terminates only on $X = 1$ or $X = 0$. VIAP translates it into the following axioms:[5]

```
X1 = X2(_N1)
X2(_n1+1) = ite(X2(_n1)==0,1,X2(_n1))
X2(0) = X
X2(_N1)==1
_n1<_N1 -> X2(_n1)!=1
```

where `ite(c,e1,e2)` is the if-then-else conditional expression: if `c` then `e1` else `e2`. This set of axioms is contradictory with any value of X except 1 or 0. Thus given a precondition like $X = 2$, any postcondition can be proved. So in this case our system proves only partial correctness.

On the other hand, this example also shows that our translated axioms can be used to deduce on which input the program terminates. First, the set of axioms is consistent. This means that the program terminates on *some* inputs. Furthermore, it entails $X = 1 \vee X = 0$. This means that it can terminate only with these two values.

Both of these can be easily verified by Z3. The consistency is verified by Z3 as it returns a model for the following Z3 query which is a direct translation of the above axioms in Z3's python format:

```
from z3 import *
X=Int('X')
X1=Int('X1')
X2=Function('X2',IntSort(),IntSort())
_N1=Const('_N1',IntSort())
_n1=Int('_n1')
_s=Solver()
_s.set("timeout",60000)
_s.add(X1 == X2(_N1))
_s.add(ForAll([_n1],Implies(_n1>=0,
X2(_n1+1)==If(X2(_n1)==0,1,X2(_n1)))))
_s.add(X2(0) == X)
_s.add((X2(_N1)==1))
_s.add(ForAll([_n1], Implies(And((_n1<_N1)
,_n1>=0),
Not(X2(_n1)==1))))
_s.add(_N1>=0)
if sat==_s.check():
   print _s.model()
elif unsat==_s.check():
   print "unsat, proved"
else:
   print "time out"
```

Now if we add the following sentence to the above query:

```
_s.add(And(Not(X==1), Not(X==0)))
```

Z3 returns with "unsat", thus proves that the axioms entails $X = 1 \vee X = 0$.

While we have not systematically explored it, this does suggest that our translated axioms can be used for proving total correctness as well. More generally, they can be used to prove properties about termination. This will be one direction for our future work. There are many interesting questions to consider here. For instance, if the set of axioms for a program is consistent but becomes unsatisfiable when a precondition is added, does this entail that the program will not terminate for any input that satisfies the precondition? In other words, is the inconsistency only due to non-termination?

VII. Related work

There has been much work on program verification, and many related systems. Given the complexity of formal program verification, many tools provide interactive proof assistants, instead of a fully automatic prover like our system. These include ACL [23], Isabelle [24], Coq [25], Boyer-Moore provers [26] and PVS [27]. Most of these interactive proof assistant systems provide facilities to operate over purely functional programming languages. Many of them also support tactic-style proofs. We hope to integrate at least some of these tactics in our system in the future.

Most of the formal verification tools for non-parallel imperative computer programs relies on having a suitable loop invariant, a formula that remains unchanged throughout the execution of the loop body, to reason about the correctness of a given loop. There has been a large amount of work on automatic discovery of loop invariants. Our system can make use of loop invariants by simply adding them to the set of axioms: an invariant I for `while C { B }` corresponds to the axiom

$$\forall n. C(n) \wedge I(n) \rightarrow I(n+1).$$

However, we want to see how much we can do without using any explicit loop invariants. Furthermore, we believe that it may be more effective to use traditional mathematical induction with first-order axioms than specialized systems with loop invariants. We have shown with VIAP that this strategy works for some non-trivial programs that would require suitable loop invariants for other systems. So far we have only implemented a straightforward induction scheme using Z3 as the base theorem prover. There has been much work on automating induction (e.g. [28]). We hope to make use of this work and implement some more powerful induction strategies in a future version of VIAP.

[5]Notice that there are two equalities used in the axioms: "=" and "==". They have the same meaning in logic.

VIII. Concluding remarks and future work

We have described an implemented system called VIAP that can prove the partial correctness of a structured program with integer assignments with respect to a pair of a precondition and a post-condition. VIAP works by first translating the given program to a set of first-order axioms and then using the off-the-shelf SMT prover Z3 to prove that the post-condition holds under the given precondition. Integrated into both the translation part and the proving part is the use of the off-the-shelf math systems sympy and Mathematica to solve recurrences and simplify algebraic expressions. The following are some concluding remarks:

- The translation needs to be done only once for a program and the translated axioms can be used to prove multiple post-conditions. While not shown in this paper, properties about the program during its execution can be expressed and proved as well. All one needs is a label to mark the point of interest - see [2].
- For each loop in the program, the translated axioms include an explicit representation of the exact number of iterations that this loop executes before terminating. All properties about the loops are expressed in terms of this representation.
- Loop invariants are not required. Of course, they can be added to the axioms if available. Our experiments show that for many programs, VIAP can prove properties about them that would normally require proper loop invariants. In fact, most of these programs are benchmarks for testing systems that discover loop invariants.

For future work, we want to extend the system with more powerful heuristics for induction, for wide classes of programs, and for proving properties regarding termination.

Acknowledgment

We would like to thank Jianmin Ji, Jack Yao, Haodi Zhang and Prashant Saikia for useful discussions. We are grateful to the developers of Z3 and sympy for making their systems available for open use, to Wolfram Research for making Wolfram Alpha an online service available to the public. . All errors remain ours.

References

[1] L. de Moura and N. Bjorner, "The Z3 SMT Solver." http://research.microsoft.com/en-us/um/redmond/projects/z3/, 2012.

[2] F. Lin, "A formalization of programs in first-order logic with a discrete linear order," *Artificial Intelligence*, vol. 235, pp. 1 – 25, 2016.

[3] SymPy Development Team, *SymPy: Python library for symbolic mathematics*, 2016.

[4] Wolfram Research Inc., *Mathematica 8.0*, 2010.

[5] K. R. M. Leino, "Automating induction with an SMT solver," in *Proceedings of the 13th International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI'12, (Berlin, Heidelberg), pp. 315–331, Springer-Verlag, 2012.

[6] A. Reynolds and V. Kuncak, "Induction for SMT solvers," in *Verification, Model Checking, and Abstract Interpretation - 16th International Conference, VMCAI 2015, Mumbai, India, January 12-14, 2015. Proceedings* (D. D'Souza, A. Lal, and K. G. Larsen, eds.), vol. 8931 of *Lecture Notes in Computer Science*, pp. 80–98, Springer, 2015.

[7] T. Nguyen, D. Kapur, W. Weimer, and S. Forrest, "Using dynamic analysis to discover polynomial and array invariants," in *Proceedings of the 34th ICSE*, ICSE '12, (Piscataway, NJ, USA), pp. 683–693, IEEE Press, 2012.

[8] T. Nguyen, D. Kapur, W. Weimer, and S. Forrest, "DIG: A dynamic invariant generator for polynomial and array invariants," *ACM Trans. Softw. Eng. Methodol.*, vol. 23, no. 4, pp. 30:1–30:30, 2014.

[9] E. Rodríguez-Carbonell and D. Kapur, "Generating all polynomial invariants in simple loops," *J. Symb. Comput.*, vol. 42, no. 4, pp. 443–476, 2007.

[10] R. Blanc, V. Kuncak, E. Kneuss, and P. Suter, "An overview of the leon verification system: Verification by translation to recursive functions," in *Proceedings of the 4th Workshop on Scala*, SCALA '13, (New York, NY, USA), pp. 1:1–1:10, ACM, 2013.

[11] K. R. M. Leino and R. Monahan, "Dafny meets the verification benchmarks challenge," in *Proceedings of the Third International Conference on Verified Software: Theories, Tools, Experiments*, VSTTE'10, (Berlin, Heidelberg), pp. 112–126, Springer-Verlag, 2010.

[12] M. Carter, S. He, J. Whitaker, Z. Rakamarić, and M. Emmi, "Smack software verification toolchain," ICSE '16, (New York, NY, USA), pp. 589–592, ACM, 2016.

[13] A. Gurfinkel, T. Kahsai, A. Komuravelli, and J. A. Navas, *The SeaHorn Verification Framework*, pp. 343–361. Cham: Springer International Publishing, 2015.

[14] P. Schrammel and D. Kroening, "2ls for program analysis - (competition contribution)," in *TACAS 2016*, vol. 9636 of *Lecture Notes in Computer Science*, pp. 905–907, Springer, 2016.

[15] M. Heizmann, J. Christ, D. Dietsch, E. Ermis, J. Hoenicke, M. Lindenmann, A. Nutz, C. Schilling, and A. Podelski, *Ultimate Automizer with SMTInterpol*, pp. 641–643. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013.

[16] E. Cohen, *Programming in the 1990s - An Introduction to the Calculation of Programs*. Texts and Monographs in Computer Science, Springer, 1990.

[17] T. Nguyen, D. Kapur, W. Weimer, and S. Forrest, "Using dynamic analysis to discover polynomial and array invariants," in *2012 34th International Conference on Software Engineering (ICSE)*, pp. 683–693, IEEE, 2012.

[18] W. Ahrendt, L. Kovács, and S. Robillard, *Reasoning About Loops Using Vampire in KeY*, pp. 434–443. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015.

[19] T. A. Henzinger, T. Hottelier, and L. Kovács, *Valigator: A Verification Tool with Bound and Invariant Generation*, pp. 333–342. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008.

[20] A. Gupta and A. Rybalchenko, "Invgen: An efficient invariant generator," in *CAV*, pp. 634–640, Springer, 2009.

[21] N. Eén and N. Sörensson, "An extensible sat-solver," in *SAT 2003*, pp. 502–518, 2003.

[22] K. R. M. Leino, "Dafny: An automatic program verifier for functional correctness," in *Proceedings of the 16th LPAR*, LPAR'10, (Berlin, Heidelberg), pp. 348–370, Springer-Verlag, 2010.

[23] M. Kaufmann, J. S. Moore, and P. Manolios, *Computer-Aided Reasoning: An Approach*. Norwell, MA, USA: Kluwer Academic Publishers, 2000.

[24] T. Nipkow, M. Wenzel, and L. C. Paulson, *Isabelle/HOL: A Proof Assistant for Higher-order Logic*. Berlin, Heidelberg: Springer-Verlag, 2002.

[25] Y. Bertot and P. Castran, *Interactive Theorem Proving and Program Development: Coq'Art The Calculus of Inductive Constructions*. Springer Publishing Company, Incorporated, 1st ed., 2010.

[26] R. S. Boyer, M. Kaufmann, and J. S. Moore, "The boyer-moore theorem prover and its interactive enhancement," *Computers & Mathematics with Applications*, vol. 29, no. 2, pp. 27–62, 1995.

[27] S. Owre, J. Rushby, N. Shankar, *et al.*, "Pvs specification and verification system," *Look at URL http://pvs. csl. sri. com*, 2004.

[28] A. Bundy, "The automation of proof by mathematical induction," in *Handbook of Automated Reasoning* (J. A. Robinson and A. Voronkov, eds.), pp. 845–911, Elsevier and MIT Press, 2001.