VIAP 1.1

Pritom Rajkhowa and Fangzhen Lin

Presenter: Pritom Rajkhowa

Hong Kong University of Science and Technology

April 5, 2019
Aim and Motivation

- 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.

- Here we 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. An earlier version of VIAP competed at SV-COMP 2018, and is described in [RL17, RL18].
Work Flow of VIAP

Input
Program

Translation

Recurrence
Solver

Z3

Output

Input
Program

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

Output
of
Translator

\begin{verbatim}
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 \rightarrow x_2(n) < 100.
\end{verbatim}

Output
of
Recurrence
Solver

\begin{verbatim}
x_1 = N,
y_1 = ite(0 \leq N < 50, N, 100 - N),
N \geq 100,
\forall n < N \rightarrow n < 100.
\end{verbatim}

Input
to Z3

\begin{verbatim}
s.add(x_1 == N)
s.add(y_1 ==
  If(And(0 \leq N, N < 50)
    , N, 100-N))
s.add(N \geq 100)
s.add(Forall([n],Implies(And(n \geq 0, n < N), n < 100))
\end{verbatim}
VIAP Architecture

Front End
- Control Flow Simplification
- Translator
- Recurrence Solver

Back End
- Proof Engine
- SMT solver (z3)
- Result
- Violation witness
- Correctness witness
- Unknown

*.c
- Pritom Rajkhowa and Fangzhen Lin
- Presenter: Pritom Rajkhowa (UST)
- April 5, 2019
Recurrence Solver (RS)

Rec-Classifier

Non-Conditional Recurrence Solver

Mutual Recurrence Solver

Conditional Recurrence Solver

Closed form solution

\[ x(n + 1) = x(n) + 1, \quad x(0) = 0 \]
\[ y(n + 1) = x(n) + y(n), \quad y(0) = 0 \]
\[ z(n + 1) = \text{ite}(n < 50, z(n) + 1, z(n)), \quad z(0) = 0 \]
\[ a(n + 1) = a(n) + b(n), \quad a(0) = 0 \]
\[ b(n + 1) = a(n) + b(n), \quad y(0) = 0 \]
\[ z(n + 1) = \text{ite}(n < 50, z(n) + 1, z(n)), \quad z(0) = 0 \]
\[ z(n) = \text{ite}(0 \leq n < 50, n, 50) \]

\[ a(n) = \text{ite}(0 \leq n \leq 1, 0, 2^n) \]
\[ b(n) = \text{ite}(0 \leq n \leq 1, 0, 2^n) \]

\[ x(n) = n, \quad y(n) = n \ast (n + 1)/2 \]
Divergent from other Approaches

- Existing work based on recurrences analysis either focuses on computing accurate information about syntactically restricted loops, or focuses on over-approximate analysis of general loops. In contrast, our recurrences precisely summarize the semantics of general loops. In other words, we aim to analyze the accurate semantics of general loops.

- A major disadvantage of existing methods is that if they fail to find a closed form solution, then they are unable to find suitable invariants as a result, they fail to complete the proof. Whereas even if our system is unable to solve recurrence equations, our system can still complete the proof.

- Some existing methods try to use recurrence solver for inferring all invariants they can find. However our approach is property-guided (or goal-directed) and we only aim to verify the condition.
### SV-COMP 2019 Result

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Total Task</th>
<th>correct</th>
<th>correct-unconfirmed</th>
<th>incorrect</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>True</td>
<td>False</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>Array</td>
<td>231</td>
<td>175</td>
<td>28</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>Loop</td>
<td>208</td>
<td>62</td>
<td>28</td>
<td>26</td>
<td>8</td>
</tr>
<tr>
<td>Recursive</td>
<td>96</td>
<td>37</td>
<td>42</td>
<td>8</td>
<td>2</td>
</tr>
</tbody>
</table>
Strength and Weakness

- **Strength**
  - This approach comes with a clean separation between the translation (semantics) and the use of the translation in proving the properties (computation).
  - This approach proves an assertion without explicitly generating loop invariants.
  - VIAP can effectively verify a number of C programs from Arrays, Loops and Recursive sub-categories of ReachSafety category.

- **Weakness**
  - VIAP provides little or no support for translation and reasoning about dynamic linked data structures or programs with floating points.
  - It consumes more CPU time. The main overhead of VIAP comes from solving recurrences and in some cases applying the proof strategy.
Thanks