VS^3: SMT Solvers for Program Verification

Saurabh Srivastava[1,⋆], Sumit Gulwani[2], and Jeffrey S. Foster[1]

[1] University of Maryland, College Park, {saurabhs,jfoster}@cs.umd.edu
[2] Microsoft Research, Redmond, sumitg@microsoft.com

Abstract. We present VS^3, a tool that automatically verifies complex properties of programs and infers maximally weak preconditions and maximally strong postconditions by leveraging the power of SMT solvers. VS^3 discovers program invariants with arbitrary, but prespecified, quantification and logical structure. The user supplies VS^3 with a set of predicates and invariant templates. VS^3 automatically finds instantiations of the unknowns in the templates as subsets of the predicate set. We have used VS^3 to automatically verify $\forall\exists$ properties of programs and to infer worst case upper bounds and preconditions for functional correctness.

1 Introduction

There are two major hurdles in the widespread adoption of current verification tools for program analysis. The first is their inability to express invariants with the detailed logical structure (specifically, quantifiers and disjunctions) that is required for almost all non-trivial programs. The second is the annotation burden and effort required on the part of the programmer in reasoning formally about inductive invariants. The sophisticated invariants required for most non-trivial programs are hard for the programmer to come up with and impose a significant annotation burden.

In this tools paper, we describe VS^3 (Verification and Synthesis using SMT Solvers), a tool that partly addresses the above mentioned limitations by leveraging the engineering advances made in constraint solving in the past decade. VS^3 takes as input a set of invariant templates and a set of candidate predicates to populate the templates' unknowns. VS^3 then infers required program invariants and, optionally, required maximally weak preconditions. By allowing the invariants to have an arbitrary, but prespecified, logical structure, VS^3 can verify sophisticated (e.g., $\forall$ and $\forall\exists$) properties of programs. Additionally, by automatically inferring maximally weak preconditions, VS^3 reduces the burden on the programmer, both in coming up with difficult preconditions and in specifying them alongside the code. In the future, we plan to augment VS^3 with template inference and abstraction refinement to build a powerful system for software verification with minimal user input.

We have used VS^3 with great success to verify a wide array of difficult program properties, such as the full functional correctness of standard implementations of all major sorting algorithms [1], and have additionally used it to infer difficult preconditions for functional correctness. VS^3 interfaces with off-the-shelf

⋆ The work reported here was supported in part by NSF CCF-0430118 and was in part done during an internship at Microsoft Research.

(a) `BinarySearch(Array` A, `int` e, `int` n)

```
1   low := 0; high := n − 1;
2   while (low ≤ high)
3       mid := ⌈(low + high)/2⌉;
4       if (A[mid] < e)
5           low := mid + 1;
6       else if (A[mid] > e)
7           high := mid − 1;
8       else return true;
9   Assert (∀j : (0 ≤ j < n) ⇒ A[j] ≠ e)
10  return false;
```

User Input:

Template: $v_1 \ \wedge \ (\forall j : v_2 \Rightarrow v_3) \ \wedge \ (\forall j : v_4 \Rightarrow v_5) \ \wedge \ (\forall j : v_6 \Rightarrow v_7)$

Predicate Set:

`AllPreds`$(\{j, n, low, high\}, \{0\}, \{\leq, <\}) \ \cup$
`AllPreds`$(\{A[t] \mid t \in \{j, j \pm 1\}\} \cup \{e\}, \{0\}, \{\leq, \neq\})$

Tool Output: "Precondition required":

Precondition:

$\forall j : (0 \leq j < n) \Rightarrow A[j] \leq A[j+1]$

Loop Invariant:

$0 \leq low \wedge high < n$
$\forall j : (low \leq j \leq high) \Rightarrow A[j] \leq A[j+1]$
$\forall j : (0 \leq j < low) \Rightarrow A[j] \neq e$
$\forall j : (high < j < n) \Rightarrow A[j] \neq e$

(b) `SetInclusion(Array` A, `int` n, `Array` B, `int` m)

```
1   for (i = 0; i < n; i++)
2       exists := false;
3       for (j = 0; j < m; j++)
4           if (A[i] = B[j])
5               exists := true; break;
6       if (¬exists) return false;
7   Assert (∀y∃x : (0 ≤ y < n)
8       ⇒ (A[y] = B[x] ∧ 0 ≤ x < m))
9   return true;
```

User Input:

Template: $v_1 \ \wedge \ (\forall y \exists x : v_2 \Rightarrow v_3) \ \wedge \ (\forall y \exists x : v_4 \Rightarrow v_5)$

Predicate Set:

`AllPreds`$(\{x, y, i, j, m\}, \{0\}, \{\leq, <\}) \ \cup$
`AllPreds`$(\{exists\}, \{\texttt{true}, \texttt{false}\}, \{=\}) \ \cup$
`AllPreds`$(\{A[t], B[t] \mid t \in \{x, y\}\}, \{0\}, \{=\})$

Tool Output: "Assertion valid":

Outer loop invariant:

$\forall y \exists x : 0 \leq y < i \Rightarrow (A[y] = B[x] \wedge 0 \leq x < m)$

Inner loop invariant:

$j \geq 0$
$\forall y \exists x : 0 \leq y < i \Rightarrow (A[y] = B[x] \wedge 0 \leq x < m)$
$\forall y \exists x : (y = i \wedge exists = \texttt{true}) \Rightarrow (A[y] = B[x] \wedge 0 \leq x < m)$

Fig. 1. (a) VS^3 computes the maximally weak precondition for correctness of binary search (b) VS^3 computes the $\forall\exists$ invariants required to prove the correctness of a program that checks set inclusion. (`AllPreds`(Z, C, R) denotes the set of predicates $\{z \ \texttt{op} \ z' \mid z, z' \in Z \cup C, \texttt{op} \in R\}$.)

SMT solvers for its core reasoning and provides a rich source for SMT benchmark instances from the domain of verification and property inference of programs.

1.1 Motivating Examples and Tool Usage

We now illustrate the power of VS^3 by considering two motivating examples.

Precondition Inference. Consider the program shown in Fig. 1(a), which implements a binary search for the element e in an array A. The functional specification of the program is given as the assertion[3] on Line 9, which essentially states that if the procedure returns false, then A indeed does not contain e.

[3] VS^3 allows the user to specify assertions and assumptions with arbitrary logical structure up to those expressible in the underlying SMT solver. Assumptions may be required to model expressions not handled by the solver. For instance, in the current system, the assignment on Line 3 is modeled as `Assume`$(low \leq mid \leq high)$.

For this function, VS^3 automatically infers the maximally weak precondition for functional correctness, shown in Fig. 1(a), which is that the input array is sorted. VS^3 also infers the loop invariant, also shown in Fig. 1(a), encoding the semantics of binary search (that the array elements between *low* and *high* are sorted and those outside do not equal e).

In automatic `CUTPOINT` mode, VS^3 searches for inductive program invariants at loop headers. Alternatively, in some cases the invariants are simpler if inferred at specific locations, which should form a valid cutset such that each cycle in the CFG contains at least one location [2]. VS^3 also supports a `MANUAL` mode for user-specified cutsets.

The user also specifies a *global* invariant template and a *global* predicate set, as shown. The template is used for invariants at each cutpoint and the predicate set specifies the candidate predicates for the unknowns in the template. In practice, the user starts by guessing simple templates and predicates. If the analysis fails then the user iteratively increases the sophistication of the templates and predicates until VS^3 finds a solution.

In practice, the logical structure (quantification and boolean connectives) of the templates is easily derived from given program assertions and discovered by iterative guessing. For instance, for `BinarySearch` we first tried a template with one unquantified and one quantified conjunct, but VS^3 failed to infer an instantiation. We then iteratively increased the number of quantified conjuncts until a solution was found. Also, typically we used a predicate set consisting of inequality relations between relevant program and bound variables. In our experience (over a large set of programs [1]), coming up with the templates and predicate set is typically easy for the programmer.

Verification using invariants with arbitrary logical structure. Consider the program shown in Fig. 1(b), which checks whether all elements of A are contained in B. The loop invariant required contains $\forall\exists$ quantification, which VS^3 infers. We do not know of any other tool that can automatically discover such invariants. Note how the conjuncts in the invariant template in this case follow the schematic of the given assertion and therefore are $\forall\exists$-quantified. As before, we discovered the appropriate number of conjuncts by iterative guessing.

2 Tool Architecture

VS^3 uses Microsoft's Phoenix compiler framework [3] as a front end parser for ANSI-C programs and Z3 [4] to solve the SMT queries. Our implementation is approximately 15K non-blank, non-comment lines of C# code.

The tool architecture is shown in Fig. 2. VS^3 uses Phoenix to generate the control flow graph (CFG) of the program. The CFG is then split into simple paths using a cutset (either generated automatically with a cutpoint at each loop header or specified by the user). The tool then generate a verification condition (VC) corresponding to each simple path. For fixed-point computation the tool provides two alternatives:

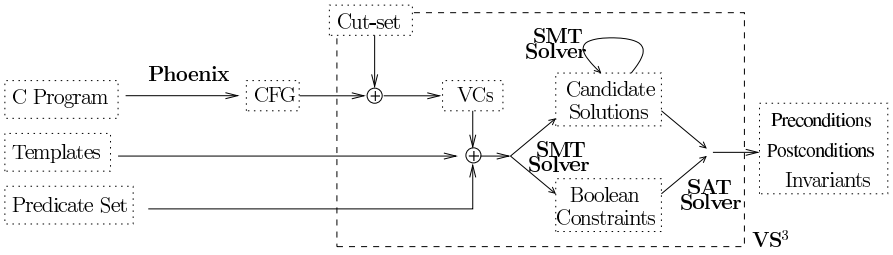

Fig. 2. The VS^3 tool. In addition to the ANSI-C program, the user provides the templates, the predicate sets, and optionally, a cutset. The tool provides the option of choosing between an iterative and a constraint-based fixed-point computation.

Iterative Fixed-Point [1]. The iterative scheme performs a variant of a standard dataflow analysis. It maintains a set of candidate solutions, and by using the SMT solver to compute the best transformer it iteratively improves the candidates until a solution is found. See [1] for details.

Constraint-based Fixed-Point [5, 1]. In the constraint-based scheme, a predicate p at location l is identified by boolean indicator variables $b_{p,l}$. For verification condition vc, VS^3 generates the minimal set of constraints over the indicator variables that ensures that vc is satisfiable. These constraints are accumulated and solved using a SAT solver that yields a fixed-point solution. See [1] for details.

2.1 Solver Interface

Compensating for current limitations of SMT solvers The generic primitives provided by SMT solvers are expressive but lacking in some aspects that are needed for our application.[4] We augment the solver by providing a wrapper interface that preprocesses the SMT queries and adds hints for the solver.

Patterns for quantifier instantiation. The current state-of-art for reasoning over quantified facts uses the now commonly known technique of E-matching [6] for quantifier instantiation. E-matching uses *patterns* to match against ground terms. Because individual SMT queries in our system are over simple quantified terms, a simple heuristic to automatically generate patterns suffices: Given a quantified fact with bound variables $\overline{k}$ and bound boolean term F, VS^3 recursively parses F and returns valid patterns from F's subterms. A subterm f is a valid pattern if it contains all the bound variables and at least one subterm of f does not contain all the variables. For example, for the fact $\forall k : k > 10 \Rightarrow A[k] < A[k+1]$, VS^3 computes the set of patterns $\{\{k > 10\}, \{A[k]\}, \{A[k+1]\}\}$, and for $\forall k : k \geq 0 \wedge k < v \Rightarrow A[k] < min$, VS^3 computes the set $\{\{k \geq 0\}, \{k < v\}, \{A[k] < min\}\}$. This simple heuristic is potentially expensive, but allows for automatic and, in practice, fast proofs or disproofs of the implication checks.

[4] Our current implementation uses the Z3 solver, but the limitations we discuss here apply to all state-of-the-art solvers, as far as we know.

Saturating inductive facts. SMT solvers have difficulty instantiating relevant facts from inductive assumptions. For instance, in our experiments, we encountered assumptions of the form $k_n \geq k_0 \wedge \forall k : k \geq k_0 \Rightarrow A[k] \leq A[k+1]$ from which $A[k_0] \leq A[k_n+1]$ was needed for the proof. Z3 times out without finding a proof or disproof of whether $A[k_0] \leq A[k_n+1]$ follows from the assumption. Notice that the pattern $k \geq k_0$ will only allow the prover to instantiate $A[k_n] \leq A[k_n+1]$ from the ground fact, which does not suffice to prove $A[k_0] \leq A[k_n+1]$.

VS^3 therefore syntactically isolates inductive facts and saturates them. To do this, it pattern matches quantified assumptions such as the above (consisting of a base case in the antecedent and the induction step in the consequent of the implication) and asserts the quantified inductive result. For example, for the case above, the saturated fact consists of $\forall k_2, k_1 : k_2 \geq k_1 \geq k_0 \Rightarrow A[k_1] \leq A[k_2+1]$. Using the ground term $k_n \geq k_0$, the tool can now instantiate $A[k_0] \leq A[k_n+1]$. This heuristic allows the tool to deal with inductive reasoning for particular cases that arise in program analysis.

Explicit Skolemization for $\forall\exists$. Z3 v1.0 does not correctly instantiate global skolemization functions for existentials under a quantifier and so VS^3 infers these functions from the program[5]. An approach that suffices for all our benchmark examples is to rename the skolemization functions at the endpoints of a verification condition and to insert axioms relating the two functions. VS^3 syntactically infers the appropriate axioms over the skolem functions for the two specific cases that we describe below.

VS^3 can infer appropriate skolemization functions for the two cases of the verification condition containing array updates and assumptions equating array values. Suppose in the quantified formulae at the beginning and end of a simple path, the skolemization functions are $\texttt{skl}$ and $\texttt{skl}'$, respectively. For the case of array updates, suppose that locations $\{l_1, l_2, \ldots, l_n\}$ are overwritten with values from locations $\{r_1, r_2, \ldots, r_n\}$. Then the tool introduces two axioms. The first axiom states that the skolemization remains unchanged for locations that are not modified (Eq. 1), and the second axiom defines the (local) changes to the skolemization function for the array locations that are modified (Eq. 2):

$$\forall y : (\wedge_i(\texttt{skl}(y) \neq r_i \wedge \texttt{skl}(y) \neq l_i)) \Rightarrow \texttt{skl}'(y) = \texttt{skl}(y) \quad (1)$$

$$\bigwedge\nolimits_i \forall y : \texttt{skl}(y) = r_i \Rightarrow \texttt{skl}'(y) = l_i \quad (2)$$

For the case of assumptions equating array values, VS^3 asserts the corresponding facts on $\texttt{skl}'$, e.g., if $\texttt{Assume}(A[i] = B[j])$ occurs and $\texttt{skl}'$ indexes the array B then the tool adds the axiom $\texttt{skl}'(i) = j$.

Axiomatic support for additional theories Some program verification tasks require support for non-standard expressions, e.g., reachability in linked-list or tree data structures. SMT solvers, and in particular Z3, support the addition of axioms to support these kind of predicates.

[5] We are aware of work being pursued in the solving community that will eliminate this restriction. Therefore in the future we will not need to infer skolemization functions.

There are two steps towards the verification of such programs. First, VS^3 defines the semantics of field accesses and updates on record datatypes using `sel` and `upd`. A field access $s \rightarrow f$, is encoded as $\texttt{sel}(f, s)$ and an update $s \rightarrow f := e$ is encoded as $\texttt{upd}(f, s, e)$. Second, VS^3 asserts a set of axioms to define the semantics of higher level predicates, such as reachability, in terms of the constructs that appear in the program. Let $x \rightsquigarrow y$ denote that y can be reached by following pointers starting at x. Then for the case of reasoning about singly linked lists connected through `next` fields, the tool augments the SMT solver with the following reachability axioms:

$$\begin{array}{rll}
\forall x \,.\, x \rightsquigarrow x & & \text{Reflexivity} \\
\forall x, y, z \,.\, x \rightsquigarrow y \wedge y \rightsquigarrow z & \Rightarrow \quad x \rightsquigarrow z & \text{Transitivity} \\
\forall x \,.\, x \neq \bot & \Rightarrow \quad x \rightsquigarrow (x \rightarrow \texttt{next}) & \text{Step: Head} \\
\forall x, y \,.\, x \rightsquigarrow y & \Rightarrow \quad x = y \vee (x \rightarrow \texttt{next}) \rightsquigarrow y & \text{Step: Tail} \\
\forall x \,.\, \bot \rightsquigarrow x & \Rightarrow \quad x = \bot & \text{End}
\end{array}$$

Using these axioms the solver can, for instance, prove that $\mathit{head} \rightsquigarrow \mathit{tail} \wedge \mathit{tail} \rightsquigarrow n \wedge n \neq \bot \Rightarrow \mathit{head} \rightsquigarrow (n \rightarrow \texttt{next})$.

3 Summary of Experiments and Conclusions

We have used VS^3 to verify a wide variety of properties of programs manipulating arrays. We have verified the full correctness of standard implementations of five major sorting algorithms. Showing sortedness for these examples required $\forall$ invariants, and showing permutation required $\forall\exists$ invariants (which have received little previous attention [7,8]).

Additionally, we have successfully used VS^3 for inferring maximally weak preconditions. We have derived maximally weak preconditions for functional correctness and for proving worst case upper bounds for programs, e.g., inferring the input that yields the worst case run of a sorting program. See [1] for details.

Our experiments with VS^3 have demonstrated its promise for the verification and inference of difficult program properties.

References

1. Srivastava, S., Gulwani, S.: Program verication using templates over predicate abstraction. In: PLDI. (2009)
2. Gulwani, S., Srivastava, S., Venkatesan, R.: Program analysis as constraint solving. In: PLDI. (2008) 281–292
3. Microsoft Research: Phoenix (2008) `http://research.microsoft.com/Phoenix/`.
4. de Moura, L., Bjørner, N.: Z3: Efficient SMT solver. In: TACAS'08. (2008)
5. Gulwani, S., Srivastava, S., Venkatesan, R.: Constraint based invariant inference over predicate abstraction. In: VMCAI. (2009)
6. Moura, L., Bjørner, N.: Efficient E-matching for smt solvers. In: CADE-21. (2007)
7. Balaban, I., Fang, Y., Pnueli, A., Zuck, L.D.: IIV: An invisible invariant verifier. In: CAV. (2005) 408–412
8. Kovács, L., Voronkov, A.: Finding loop invariants for programs over arrays using a theorem prover. In: FASE. (2009)