Program Verification using Small Models

Nakshatra Gupta, Prajkta Kodavade, Shrawan Kumar,
Abhisekh Sankaran, Akshatha Shenoy, R. Venkatesh

TCS Research, Pune

RHPL 2023
IIIT Hyderabad

Dec 20, 2023

Introduction

Program Verification problem

Given a program P and an assertion A, is it the case that for every input I, the assertion A holds at the end of the execution of P on I?

- This problem is undecidable (Turing '36).
- We explore the use of the methods from model theory in investigating this problem.

Translation into logic

- Given the program P and assertion A, we encode these respectively as sentences φ_P and φ_A of first order logic (FO) over a suitable vocabulary.
- The sentences would be such that, over the class $\mathrm{Mod}(\varphi_{\mathsf{ax}})$ of finite models of a small set φ_{ax} of axioms (i.e. modulo φ_{ax}), the following hold:
  - the models of φ_P capture precisely (suitably abstract representations of) the executions of P for arbitrary inputs
  - the models of $\varphi_A \wedge \varphi_P$ capture precisely the said executions of P that satisfy A

FO satisfiability

- Verifying whether P satisfies A then translates to checking modulo φ_{ax}, the validity of the FO sentence

$$\varphi_P \to \varphi_A$$

- Equivalently, this verification translates to checking modulo φ_{ax}, the satisfiability of

$$\varphi_P \wedge \neg\varphi_A$$

- FO satisfiability in the finite is undecidable (r.e.) in general (Trakhtenbrot '50).
- We explore the use of a particular model-theoretic condition known in the logic literature to give decidability results, called the small model property (SMP).

Small model property (SMP)

- We say that (a class of sentences of the form) $\beta := \varphi_P \wedge \neg\varphi_A$ has the small model property if there is a "nice" function $f : \mathbb{N} \to \mathbb{N}$, such that the following holds modulo φ_{ax}:

$$\text{There is a model for } \beta \quad \to \quad \text{There is a model for } \beta \text{ of size } \leq f(|\beta|)$$

- From the perspective of P and A, this translates to asserting the existence of a small execution of P violating A on some input (and hence on a small input), if at all there is some finite length execution of P violating A.

Small model property (SMP)

- We say that (a class of sentences of the form) $\beta := \varphi_P \wedge \neg\varphi_A$ has the small model property if there is a "nice" function $f : \mathbb{N} \rightarrow \mathbb{N}$, such that the following holds modulo φ_{ax}:

$$\text{There is a model for } \beta \quad \rightarrow \quad \text{There is a model for } \beta \text{ of size } \leq f(|\beta|)$$

- This gives a complete decision procedure to check if β is satisfiable – enumerate all structures of size $\leq f(\beta)$ and check for the truth of $\alpha := \varphi_{\mathsf{ax}} \wedge \beta$. This can be implemented using a SAT solver like Z3.

Small model property (SMP)

- We say that (a class of sentences of the form) $\beta := \varphi_P \wedge \neg\varphi_A$ has the small model property if there is a "nice" function $f : \mathbb{N} \rightarrow \mathbb{N}$, such that the following holds modulo φ_{ax}:

$$\text{There is a model for } \beta \quad \rightarrow$$
$$\text{There is a model for } \beta \text{ of size } \leq f(|\beta|)$$

- Above "nice" in theory usually means "computable", but for our purposes it means bounds that would make SAT solvers checking for small models of α, to run within competitive time (w.r.t. say SV-COMP).

Min program P and min assertion A

Program P:

```
m = a[0];
i = 0;
while (i < n)
{
    if (m > a[i])
        m = a[i];
    i++;
}
```

Assertion A:

```
i = 0;
while (i < n)
{
    assert(m <= a[i]);
    i = i + 1;
}
return 0;
```

- The sentences $\varphi_{\mathsf{ax}}, \varphi_P$ and φ_A are multi-sorted FO sentences over a suitably chosen vocabulary.

Main results for min program and assertion

Let P = min program and A = min assertion.

Theorem (Small models for α).

If there is a finite model for $\alpha := \varphi_{\mathsf{ax}} \land \varphi_P \land \neg\varphi_A$, then there is also such a model in which the sizes of (the domains interpreting) the sorts are all bounded by 7.

Equivalently:

Theorem (Small models for min).

If for some input array, the execution of the min program violates the min assertion, then there is also such an input array of size at most 7.

Main results for min program and assertion

Let P = min program and A = min assertion.

Theorem (Small models for α).

If there is a finite model for $\alpha := \varphi_{\mathsf{ax}} \wedge \varphi_P \wedge \neg\varphi_A$, then there is also such a model in which the sizes of (the domains interpreting) the sorts are all bounded by 7.

Equivalently:

Theorem (Smaller models for min).

If for some input array, the execution of the min program violates the min assertion, then there is also such an input array of size at most 3.

Intuitive idea

- Consider an array a with say 3 elements.

7	3	5

- The "m-unitialized" min program defines a function f from input values of m to output values of m.
- Can we construct another array b of size lesser than a such that the min program computes the same function f from input m to output m?

Intuitive idea

- Consider an array a with say 3 elements.

7	3	5

- The "m-unitialized" min program defines a function f from input values of m to output values of m.
- Can we construct another array b of size lesser than a such that the min program computes the same function f from input m to output m?
- Yes! Let b be the array as below:

$\min\{7, 3\}$	5

Intuitive idea

- Consider an array a with say 3 elements.

7	3	5

- The "m-unitialized" min program defines a function f from input values of m to output values of m.
- Can we construct another array b of size lesser than a such that the min program computes the same function f from input m to output m?
- Exactly. Let b be the array as below:

$\min\{7, 3, 5\}$

Intuitive idea

- Suppose now there is a large array $\mathcal{A}$ on which the min program execution violates the min assertion.

$\mathcal{A}$	v_0	$\cdots$	v_k	v_{k+1}	v_{k+2}	v_{k+3}	$\cdots$	v_n

$$v_k < m_n$$

m_{-1}	m_0	$\cdots$	m_k	m_{k+1}	m_{k+2}	m_{k+3}	$\cdots$	m_n

The m_i's represent the values of m across iterations of the min program. So $m_{-1} = v_0$ and $m_i = \min\{m_{i-1}, v_i\}$.

Intuitive idea

- Suppose now there is a large array $\mathcal{A}$ on which the min program execution violates the min assertion.

$\mathcal{A}$	v_0	$\cdots$	v_k	v_{k+1}	v_{k+2}	v_{k+3}	$\cdots$	v_n

$$v_k < m_n$$

m_{-1}	m_0	$\cdots$	m_k	m_{k+1}	m_{k+2}	m_{k+3}	$\cdots$	m_n

Intuitive idea

- Suppose now there is a large array $\mathcal{A}$ on which the min program execution violates the min assertion.

$\mathcal{A}'$

v_0	$\cdots$	v_k	$min\{v_{k+1}, v_{k+2}, v_{k+3}\}$	$\cdots$	v_n

$$v_k < m_n$$

m_{-1}	m_0	$\cdots$	m_k	m_{k+3}	$\cdots$	m_n

Intuitive idea

- Suppose now there is a large array $\mathcal{A}$ on which the min program execution violates the min assertion.

$\mathcal{A}'$	v_0	$\cdots$	v_k	$min\{v_{k+1}, v_{k+2}, v_{k+3}\}$	$\cdots$	v_n

$$v_k < m_n \ \wedge \ |\mathcal{A}'| < |\mathcal{A}|$$

m_{-1}	m_0	$\cdots$	m_k	m_{k+3}	$\cdots$	m_n

Intuitive idea

- Suppose now there is a large array $\mathcal{A}$ on which the min program execution violates the min assertion.

$\mathcal{A}'$	v_0	$\cdots$	v_k	$min\{v_{k+1}, v_{k+2}, v_{k+3}\}$	$\cdots$	v_n

$$v_k < m_n \ \wedge \ |\mathcal{A}'| < |\mathcal{A}|$$

m_{-1}	m_0	$\cdots$	m_k	m_{k+3}	$\cdots$	m_n

- Recursively doing the reduction above on either side of $a[k]$, we get the array below which also witnesses the min assertion violation.

$\mathcal{A}^*$	$min\{v_0, \ldots, v_{k-1}\}$	v_k	$min\{v_{k+1}, \ldots, v_n\}$

Intuitive idea

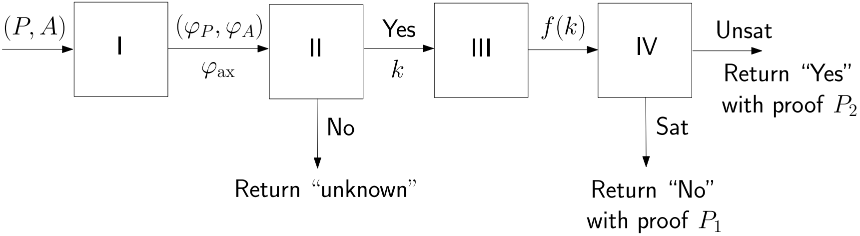
- The min program and assertion pair then has a small array of size 3 witnessing the assertion violation.
- There are however infinitely many arrays of even size 1.
- How do we check all arrays of size 3 for assertion violations?

Observation.

1. The min program does only comparisons of values.
2. The number of "order types" of 3 element arrays is small. (Specifically: 13)

- We check all arrays of size 3 whose elements take values from $\{0, 1, 2\}$. $\square$

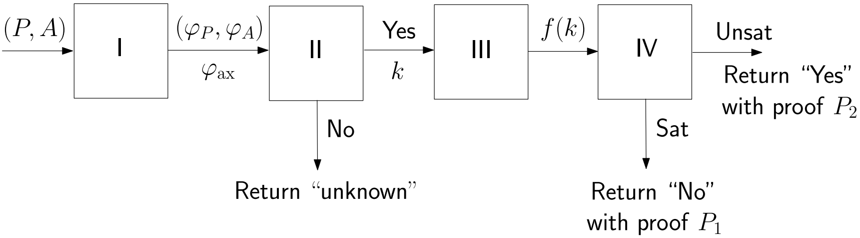
Overall approach



I: Program + assertion to logic conversion

- Converts the input program P and assertion A to FO sentences φ_P and φ_A. Also generates a small set φ_{ax} of axioms that define the class within which to investigate the models of $\beta := \varphi_P \wedge \neg\varphi_A$.

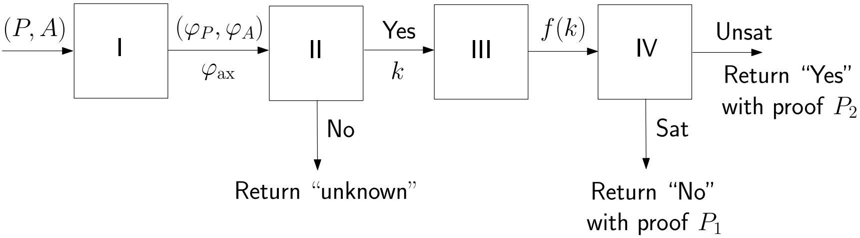
Overall approach



II: Removal of iterations

- Find k such that executions of the loop on arrays of size k and arbitrary initializations of the other variables, can be reduced strictly preserving the input-output behaviour of the loop. This reduction is done using Z3 that attempts a "removal of iterations" modifying the arrays suitably.

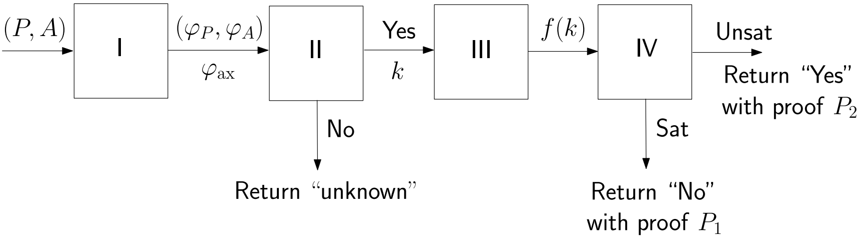
Overall approach



III: Small model bound $f(k)$ computation

- Computing a number $f(k)$ such that if $\alpha := \varphi_{\text{ax}} \wedge \beta$ has a model of size $> f(k)$, then it also has a model of size $\leq f(k)$.

Overall approach



IV: Searching for a model in structures of size $\leq f(k)$

- Checking for the satisfiability of α in structures of size $\leq f(k)$ using Z3.

P_1 and P_2

- As seen earlier, for the min program and min assertion, we can take $k = 3$ and $f(k) = 7$ (better still $f(k) = 3$).
- In case stage IV returns "Sat", the proof P_1 is the model M^* of α of size $\leq f(k)$ returned by Z_3.
- We can also return P_1 as the translation of M^* to an input array a^* and the execution trace of P on a^*.

P_1 and P_2

- As seen earlier, for the min program and min assertion, we can take $k = 3$ and $f(k) = 7$ (better still $f(k) = 3$).
- In case stage IV returns "Unsat", then the proof P_2 can be returned as:
  - the unsat core of α (over structures of sizes $\leq f(k)$)

$$+$$

  - an inductive proof over $l \geq f(k)$ of the statement that:

    no model of size $\leq l \quad \longrightarrow \quad$ no model of size $\leq l + 1$
- The inductive proof can be presented by say a program written in Coq.

Experiments

- We analyzed about 10 programs, primarily variants of min and similar programs (search, sortedness, etc.)
- The k value for most of these programs is between 3 and 4, and the $f(k)$ value between 7 and 9.
- On some of these (“reverse” min and “bubblesort” min), VeriAbs returned ‘Unknown’.
- The runtimes for the small model check are usually very small ($\sim$ 0.1 seconds).
- An end-to-end implementation of the overall approach is still under progress, so a comparison of runtimes with existing tools remains to be done.

Future work

- We have a formulation of a class of programs generalizing min for which we believe our analysis lifts as is. This just needs to be confirmed and written down.
- Extensions to investigate:
    - “Series compositions” of loops
      (Would require inferring conditions to check of the individual loops from the given assert condition)
    - Nested loops
    - Some arithmetic
    - Graph algorithms

Dhanyavād

- If you are interested in this work, TCS Research would be happy to collaborate.
- The reports for this work can found at `https://abhisekhs.github.io`.

Appendix

Variants of min: Reverse min

```
Revmin(n, a[n]):
++++++++++++++++
m = a[n-1];
i = 0;
while (i < n)
{
    if (m > a[n-1-i])
        m = a[n-1-i];
    i++;
}
assert(forall j. (j >= 0 && j < n)
                  --> m <= a[j]);
```

Variants of min: Bubblesort min

```
Bubblesortmin(n, a[n]):
+++++++++++++++++++++++
i = 0;
t = 0;
while (i < n) {
    if (a[i] < a[i+1]) {
        t = a[i+1];
        a[i+1] = a[i];
        a[i] = t;
    }
}
m = a[n-1];
assert(forall j. (j >= 0 && j < n)
                  --> m <= a[j]);
```

Beyond min: Relativized min I

```
Relmin-I(n, a[n], p):
++++++++++++++++++++++
m = a[0];
i = 0;
while (i < n)
{
    if (m > a[i] && a[i] != p)
        m = a[i];
    i++;
}
assert(forall j. (j >= 0 && j < n && a[j] != p)
                   --> m <= a[j]);
```

Beyond min: Relativized min II

```
Relmin-II(n, a[n], p[n]):
++++++++++++++++++++++++++
m = a[0];
i = 0;
while (i < n)
{
    if (p[i] == 1)
        if (m > a[i])
            m = a[i];
    i++;
}

assert(forall j. (j >= 0 && j < n && p[j] = 1)
                   --> m <= a[j]);
```

Beyond min: Search

```
Search(n, a[n], p):
+++++++++++++++++++
f = 0;
i = 0;
while (i < n)
{
    if (a[i] == p)
        f = 1;
    i++;
}
assert(f = 0 ||
       forall j. (j >= 0 && j < n && a[j] = p)
                    --> f = 1);
```

Generalizing min to a class of programs: MLL

MLL = Monotone-loops without Lookback or Lookahead

```
// Variable and array declarations
// Initializations that are either of the form
// x = const or x = y where x and y are either
// variables or array elements

// A loop-free set of statements that could
// involve conditions

i := 0;
while (i < n)
{
    // MLL loop body

    i++;
}
```

Generalizing min to a class of programs: MLL

MLL = Monotone-loops without Lookback or Lookahead

MLL loop body:

```
// A sequence of statements in SSA form that
// could involve conditions and that satisfy
// the following constraints:

// a. feature only i as index variable
// b. do not modify i
// c. use only comparison as an operator
// d. refer only to a[i] for an array a if
//    they at all refer to any element of a
// e. Assignment statements appear only
//    at the ends of branches of the
//    control flow graph of the main
//    loop body
```