Building an Automated Verifier for a Procedural Programming Language by Applying Propositional and Predicate Logic

Chen-Wei Wang, Xiang Chen
Lassonde School of Engineering, York University
{ jackie@eecs.yorku.ca, xiang2@my.yorku.ca }

Abstract

- The teaching of logic and software engineering courses has always been a challenge since the lacking of an efficient automated tool.
- We present an automated verifier:
  - to designing a language, more suitable for the purpose of modeling, which uses an SMT solver (i.e., Z3) as its backend tool.
  - to verifying if a valid propositional or predicate formula is a tautology, and provide a counterexample otherwise.
  - to automatically transforming the routine of a procedural programming language into a number of Hoare Triple.
  - to proving if the specific Hoare Triple is a tautology by systematically calculating the weakest precondition of every routine, given its implementation and postcondition.

Background

- *Tautology* means a valid (i.e., without any syntax of type errors) formula holds for every interpretation.
- *Hoare Triple:* $\{Q\}S\{R\}$ is the center feature of the Hoare logic, which is proposed in 1969 by the British computer scientist and logician Tony Hoare, and subsequently refined by Hoare and other researchers. Hoare Triple is a boolean predicate that either can be proved of disproved for verifying the correctness of a computer program.
- *Weakest Precondition (wp)* can be used for proving the Hoare Triple. $\{Q\}S\{R\} \equiv Q \Rightarrow wp(S, R)$

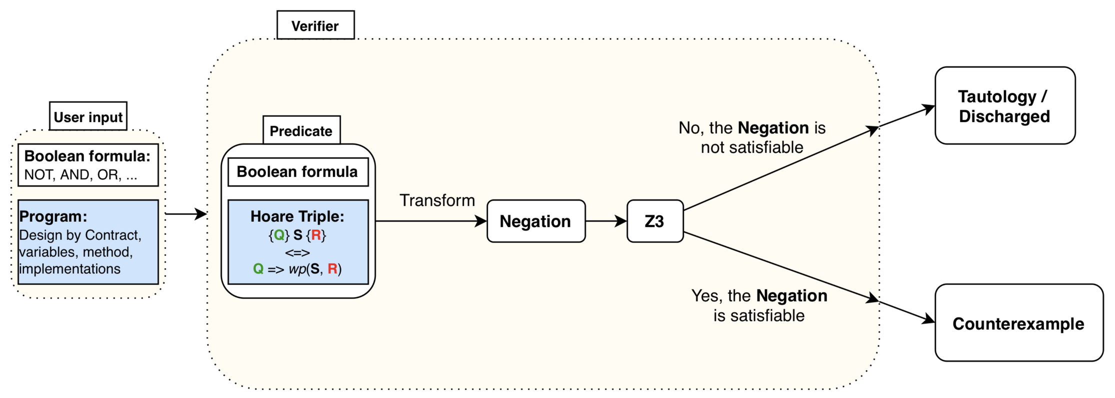
Motivation

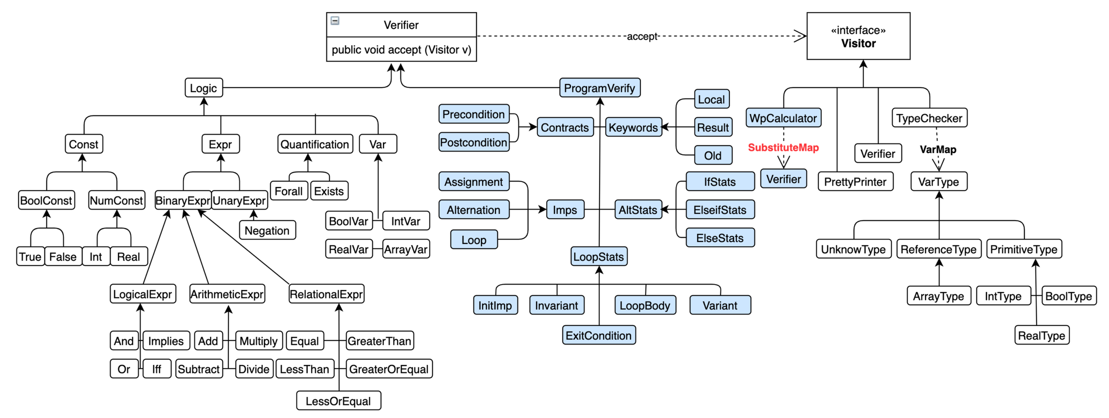
- An efficient tool for the teaching of logic and software engineering courses is necessary. However, most of the available tool are not designed for teaching.
- The input language for an SMT solver (e.g., Z3) may not be so user-friendly.

Contributions

1. An automated verifier for propositional and predicate formula.
2. An automated verifier for program verification, including the steps of :
   (a) transforming the program routine into a number of Hoare Triple
   (b) automatically calculating the weakest precondition
   (c) proving of disproving the Hoare Triple by using the calculated weakest precondition
3. Providing counterexamples if the formula or Hoare Triple is not tautology.

Methodology





1. Specifying a contex-free grammar by using Antlr 4, supporting the input of propositions, predicates, and programs.
2. Parsing user input and generate the verifier objects.
3. Apply the visitor design pattern to the verifier hierarchy for different purpose (e.g., type checking, pretty printing, wp calculating, etc.)
4. Send the generated output to the backend tool (i.e., Z3 SMT Solver) for verification.

Example of Propositional and Predicate Formula Verification

User input file:

```
// Golden rule
p: BOOLEAN
q: BOOLEAN

// verify golden rule
verify (p and q <=> p) <=> (q <=> p or q)

// verify wrong version of golden rule
verify (p and q <=> p) <=> (q <=> p and q)
```

Program output:

```
(((p and q) = p) = (q = (p or q)))
Is a tautology.

(((p and q) = p) = (q = (p and q)))
Where:
    p : BOOLEAN
    q : BOOLEAN

Is not a tautology. Here is a counter example:
    p : false
    q : true
```

- Example of formula verification: testing the Golden rule and its wrong version.
- My Verifier will give the result if the formula is a tautology and provide counterexample otherwise.

Example of Program Verification (1)

User input file:

```
-- Change the value of array at index i

change_at(a:ARRAY[REAL]; i : INTEGER; nv: REAL)
require
  valid_index: a.lower <= i and i <= a.upper
local
  x: INTEGER
  y: BOOLEAN
do
  a[i] := nv;
ensure
  changed: a[i] = nv
  unchanged: forall j:INTEGER; |(a.lower <= j and j <= a.upper and
    not i = j) => old a[j] = a[j]
  test_wrong_postcondition: not a[i] = nv
end

verify change_at
```

Program output:

```
change_at(a : ARRAY[REAL];i : INTEGER;nv : REAL)
  local
    x : INTEGER
    y : BOOLEAN
  require
    valid_index : ((a.lower <= i) and (i <= a.upper))
  do
    a[i] := nv;
  ensure
    changed : (a[i] = nv)
    unchanged : forall j | ((((a.lower <= j) and (j <= a.upper))
      and (not (i = j))) => (old a[j] = a[j]))
    test_wrong_postcondition : (not (a[i] = nv))
  end

Where:
Precondition(Q) :
    valid_index : ((a.lower <= i) and (i <= a.upper))
Postcondition(R) :
    changed : (a[i] = nv)
    unchanged : forall j | ((((a.lower <= j) and (j <= a.upper))
      and (not (i = j))) => (old a[j] = a[j]))
    test_wrong_postcondition : (not (a[i] = nv))
Implementation(S) :
    a[i] := nv;

wp(S, changed)
  (nv = nv)
wp(S, unchanged)
  forall j | ((((a.lower <= j) and (j <= a.upper)) and (not (i = j)
    )) => (old a[j] = a[j]))
wp(S, test_wrong_postcondition)
  (not (nv = nv))

Proof Obligation:
(valid_index) => wp(S, changed)
Discharged.
(valid_index) => wp(S, unchanged)
Discharged.
(valid_index) => wp(S, test_wrong_postcondition)
Not discharged.
Counterexample is not available.
```

- My Verifier supports the **Old** keyword.
- My Verifier provides the detail of program specification and weakest postcondition (wp).
- My Verifier requires user to specify the tag for each precondition and postcondition, for the result of verification, my Verifier will indicates the status of wp for each postcondition (e.g., discharged / not discharged). User of my Verifier could easily locate which postcondition is violated.

Example of Program Verification (2)

User input file:

```
-- Find the maximum element in array a and return it

find_max (a: ARRAY[INTEGER]) : INTEGER
require
  no_restriction: a.count > 0
local
  i: INTEGER
do
  from
    i := a.lower;
    Result := a[i];
  invariant
    loop_invariant: forall j: INTEGER; | (a.lower <= j and j < i)
      => (Result >= a[j])
  until
    i > a.upper
  loop
    if a[i] > Result then
      Result := a[i];
    end
    i := i + 1;
  variant
    loop_variant: a.upper - i + 1
  end
ensure
  correct_result: forall k: INTEGER; | (a.lower <= k and k <= a.
    upper) => (Result >= a[k])
  not_changed: forall s:INTEGER; | (a.lower <= s and s <= a.upper)
    => (a[s] = old a[s])
end

verify find_max
```

Program output:

```
find_max(a : ARRAY[INTEGER]) : INTEGER
  local
    i : INTEGER
  require
    no_restriction : (a.count > 0)
  do
from
    i := a.lower;
    Result := a[i];
invariant
  loop_invariant : forall j | (((a.lower <= j) and (j < i)) => (
    Result >= a[j]))
until
  (i > a.upper)
loop
    if (a[i] > Result) then
      Result := a[i];
```

```
    end
    i := (i + 1);
variant
  loop_variant : ((a.upper - i) + 1)
end
  ensure
    correct_result : forall k | (((a.lower <= k) and (k <= a.upper)
      ) => (Result >= a[k]))
    not_changed : forall s | (((a.lower <= s) and (s <= a.upper))
      => (a[s] = old a[s]))
  end

Where:
Precondition(Q) :
    no_restriction : (a.count > 0)
Postcondition(R) :
    correct_result : forall k | (((a.lower <= k) and (k <= a.upper)
      ) => (Result >= a[k]))
    not_changed : forall s | (((a.lower <= s) and (s <= a.upper))
      => (a[s] = old a[s]))
Implementation(S) :
from
    i := a.lower;
    Result := a[i];
invariant
  loop_invariant : forall j | (((a.lower <= j) and (j < i)) => (
    Result >= a[j]))
until
  (i > a.upper)
loop
    if (a[i] > Result) then
      Result := a[i];
    end
    i := (i + 1);
variant
  loop_variant : ((a.upper - i) + 1)
end
Correctness conditions :
1. Given precondition Q, the initialization step Sinit establishes
  LI I : {Q} Sinit {I}
  ((a.count > 0) => forall j | (((a.lower <= j) and (j < a.lower))
    => (a[a.lower] >= a[j])))

2. At the end of Sbody, if not yet to exit, LI I is maintained : {I
   and (not B)} Sbody {I}
  ((forall j | (((a.lower <= j) and (j < i)) => (Result >= a[j]))
    and (not (i > a.upper))) => (((a[i] > Result) => forall j | (((
    a.lower <= j) and (j < (i + 1))) => (a[i] >= a[j]))) and ((not
    (a[i] > Result)) => forall j | (((a.lower <= j) and (j < (i +
    1))) => (Result >= a[j])))))

3. If ready to exit and LI I maintained, postcondition R is
  established : I and B => R

  3-1. I and B => correct_result : ((forall j | (((a.lower <= j)
    and (j < i)) => (Result >= a[j])) and (i > a.upper)) => forall
    k | (((a.lower <= k) and (k <= a.upper)) => (Result >= a[k])))

  3-2. I and B => not_changed : ((forall j | (((a.lower <= j) and (
    j < i)) => (Result >= a[j])) and (i > a.upper)) => forall s |
    (((a.lower <= s) and (s <= a.upper)) => (a[s] = old a[s])))

4. Given LI I, and not yet to exit, Sbody maintains LV V as non-
  negative : {I and (not B)} Sbody {V >= 0}
  ((forall j | (((a.lower <= j) and (j < i)) => (Result >= a[j]))
    and (not (i > a.upper))) => (((a[i] > Result) => (((a.upper - (
    i + 1)) + 1) >= 0)) and ((not (a[i] > Result)) => (((a.upper -
    (i + 1)) + 1) >= 0))))

5. Given LI I, and not yet to exit, Sbody decrements LV V : {I and
  (not B)} Sbody {V < V0}
  ((forall j | (((a.lower <= j) and (j < i)) => (Result >= a[j]))
    and (not (i > a.upper))) => (((a[i] > Result) => (((a.upper - (
    i + 1)) + 1) < ((a.upper - i) + 1))) and ((not (a[i] > Result))
     => (((a.upper - (i + 1)) + 1) < ((a.upper - i) + 1)))))

Condition 1 is discharged.
Condition 2 is discharged.
Condition 3 postcondition correct_result is discharged.
Condition 3 postcondition not_changed is discharged.
Condition 4 is discharged.
Condition 5 is discharged.
```

- My Verifier supports the **Result** keyword.
- For programs that contains the loop, my Verifier will follow the wp calculation rule and provide the result and status of the five conditions.

Forthcoming Research

- Verification of the computer program that contains nested loop and recursive methods.
- Providing counterexample for array and quantifications.