SAT-based Model Checking for C programs

Moonzoo Kim
Provable Software Lab.
CS Division of EECS Dept. KAIST
Formal Methods

• Definition in Wikiopedia
  – “Formal methods are mathematically-based techniques for the specification, development and verification of software … performing appropriate mathematical analyses can contribute to the reliability and robustness of a design.”
  • Rigor, completeness
  • Automation Ad-hoc manual development
  • Scalability

• Brief History
  – Early days: 1960~: Theorem proving
    • Hoare logic – pre & post condition, loop invariant
    • Heavily depends on human expertise
  – Mid days: 1980~: Model checking
    • Design an abstract model and check the model with requirements
    • Fully automatic analysis
  – Recent days: 2000~: Software model checking
    • (semi) automatic model extraction from C code
    • Not yet reliable and weak tool supports
Overview of SAT-based Bounded Model Checking

Requirements $\downarrow$
- Formal Requirement Properties $\Box(\Phi \rightarrow \Diamond \Omega)$

C Program $\downarrow$
- Abstract Model

Model Checker
- Satisfied
- Not satisfied
- Okay
- Counter example

Requirements $\downarrow$
- Formal Requirement Properties in C (ex. assert( $x < a[i]$ ); )

Translation to SAT formula
- Satisfied
- Not satisfied
- Okay
- Counter example
SAT Basics (1/2)

- SAT = Satisfiability
  = Propositional Satisfiability
- **NP-Complete problem**
  - We can use SAT solver for many NP-complete problems
    - Hamiltonian path
    - 3 coloring problem
    - Traveling sales man’s problem
- **Recent interest as a verification engine**
SAT Basics (2/2)

• A set of propositional variables and clauses involving variables
  – \((x_1 \lor x_2' \lor x_3) \land (x_2 \lor x_1' \lor x_4)\)
  – \(x_1, x_2, x_3\) and \(x_4\) are variables (true or false)

• Literals: Variable and its negation
  – \(x_1\) and \(x_1'\)

• A CNF clause is satisfied if one of the literals is true
  – \(x_1 = \text{true}\) satisfies clause 1
  – \(x_1 = \text{false}\) satisfies clause 2

• Solution: An assignment that satisfies all clauses
Basic SAT Solving Mechanism (1/2)

/* The Quest for Efficient Boolean Satisfiability Solvers
   * by L.Zhang and S.Malik, Computer Aided Verification 2002 */

DPLL(a formula $\phi$, assignment) {
    necessary = deduction($\phi$, assignment);
    new_asgnment = union(necessary, assignment);
    if (is_satisfied($\phi$, new_asgnment))
        return SATISFIABLE;
    else if (is_conflicting($\phi$, new_asgnment))
        return UNSATISFIABLE;
    var = choose_free_variable($\phi$, new_asgnmnt);
    asgn1 = union(new_asgnmnt, assign(var, 1));
    if (DPLL($\phi$, asgn1) == SATISFIABLE)
        return SATISFIABLE;
    else {
        asgn2 = union (new_asgnmnt, assign(var,0));
        return DPLL ($\phi$, asgn2);
    }
Basic SAT Solving Mechanism (2/2)

\[(p \lor r) \land (\neg p \lor \neg q \lor r) \land (p \lor \neg r)\]

\[\begin{align*}
&p=T \\
&\ (T \lor r) \land (\neg T \lor \neg q \lor r) \land (T \lor \neg r) \\
&\text{SIMPLIFY} \\
&\neg q \lor r
\end{align*}\]

\[\begin{align*}
&p=F \\
&\ (F \lor r) \land (\neg F \lor \neg q \lor r) \land (F \lor \neg r) \\
&\text{SIMPLIFY} \\
&\ r \land \neg r \\
&\text{SIMPLIFY} \\
&\text{false}
\end{align*}\]
Model Checking as a SAT problem (1/4)

- **CBMC (C Bounded Model Checker, In CMU)**
  - Handles function calls using inlining
  - Unwinds the loops a fixed number of times (bounded MC)
    - Thus, a user has to know a upper bound of each loop
      - In practice, it does not cause serious limitation since
        » Most loops has clear upper bounds
        » Even when we do not know the upper bound, we can still get analysis result with possibility of false alarms
  - Allows user input to be modeled using non-determinism
    - So that a program can be checked for a set of inputs rather than a single input
  - Allows specification of assertions which are checked using the bounded model checking
Model Checking as a SAT problem (2/4)

- **Unwinding Loop**

  Original code
  
  ```
  x=0;
  while (x < 2) {
    y=y+x;
    x++;
  }
  ```

  Unwinding assertion:
  ```
  assert (! (x < 2))
  ```

  Unwinding the loop 3 times
  ```
  x=0;
  if (x < 2) {
    y=y+x;
    x++;
  }
  if (x < 2) {
    y=y+x;
    x++;
  }
  if (x < 2) {
    y=y+x;
    x++;
  }
  ```
Model Checking as a SAT problem (3/4)

• From C Code to SAT Formula

Original code

```c
int x = 0;
if (x != 1)
    x = 2;
else
    x++;
assert (x <= 3);
```

Convert to static single assignment

```c
int x0 = 0, x1 = x0 + y0;
if (x1 != 1)
    x2 = 2;
else
    x3 = x1 + 1;
x4 = (x1 != 1) ? x2 : x3;
assert (x4 <= 3);
```

Generate constraints

\[
C \equiv x_1 = x_0 + y_0 \land x_2 = 2 \land x_3 = x_1 + 1 \land (x_1 != 1 \land x_4 = x_2 \lor x_1 = 1 \land x_4 = x_3)
\]

\[
P \equiv x_4 \leq 3
\]

Check if \( C \land \neg P \) is satisfiable, if it is then the assertion is violated.

\( C \land \neg P \) is converted to Boolean logic using a bit vector representation for the integer variables \( y_0, x_0, x_1, x_2, x_3, x_4 \).
Model Checking as a SAT problem (4/4)

- Example of arithmetic encoding into pure propositional formula

Assume that $x, y, z$ are three bits positive integers represented by propositions $x_0x_1x_2, y_0y_1y_2, z_0z_1z_2$

$C \equiv z = x + y \equiv (z_0 \leftrightarrow (x_0 \oplus y_0) \oplus (x_1 \land y_1)) \lor (((x_1 \oplus y_1) \land (x_2 \land y_2)))$

$\land (z_1 \leftrightarrow (x_1 \oplus y_1) \oplus (x_2 \land y_2))$

$\land (z_2 \leftrightarrow (x_2 \oplus y_2))$

Half adder circuit diagram

Full adder circuit diagram

Inputs: {A, B, CarryIn} \rightarrow Outputs: {Sum, CarryOut}
#include<stdio.h>

int many_branch(int x) {
    int y=0; /* Required to show meaningful counter
   * example for non-deterministic value*/
    x=x;

    if ( x>0 )
        if ( x > 10)
            if (x >20)
                y = 4;
            else y=3;
        else y=2;
    else y=1;

    #ifdef verification
    assert( 2<= y && y <= 4);
    #endif
    return y;
}
int main() {
    int input, output;
    printf("Input x:");
    scanf("%d", &input);
    output = many_branch(input);
    printf("Output y:%d\n", output);
}

Cyclomatic complexity = 4

[moonzoo@verifier cs350]$  a.out
Input x:-5
Output y:1
[moonzoo@verifier cs350]$  a.out
Input x:5
Output y:2
[moonzoo@verifier cs350]$  a.out
Input x:15
Output y:3
[moonzoo@verifier cs350]$  a.out
Input x:25
Output y:4
Model Checking Example (3/3)

[moonzoo@verifier cs350]$ cbmc -D verification --function many_branch test.c
Starting Bounded Model Checking
size of program expression: 30 assignments
388 variables, 897 clauses
SAT checker: negated claim is SATISFIABLE, i.e., does not hold
Building error trace

Counterexample:
Initial State file /usr/include/bits/sys_errlist.h line 27 thread 0
----------------------------------------------------
sys_nerr=0

State 2 file test-all-branch.c line 4 function many_branch thread 0
----------------------------------------------------
test-all-branch::many_branch::1::y=0

State 3 file test-all-branch.c line 7 function many_branch thread 0
----------------------------------------------------
test-all-branch::many_branch::x=-2147483617

State 5 file test-all-branch.c line 15 function many_branch thread 0
----------------------------------------------------
test-all-branch::many_branch::1::y=1

Violated property:
file test-all-branch.c line 18 function many_branch
assertion
2 <= y && y <= 4
VERIFICATION FAILED
Model Checking Performances: Sort (1/2)

- Suppose that we have an array of 4 elements each of which is 32 bits long
  - int a[4];
- We want to verify sort.c works correctly
  - main() { int a[4]; sort(); assert(a[0]<= a[1]<= a[2]<=a[3]);}
  - Note that local variables are initialized with non-deterministic values.
- The # of all possible test cases is $5.4 \times 10^{39} (=2^{32 \times 4})$
  - Exhaustive testing will take $1.8 \times 10^{32}$ seconds ($= 5.8 \times 10^{24}$ years)
    - $5.4 \times 10^{39}$ cases x 100 instruction/case x $1/(3 \times 10^9)$ sec/instruction
```c
#include <stdio.h>
#define N 5
int main(){
    int data[N], i, j, tmp;

    /* if a function body is not provided, */
    /* function invocation of the function is ignored */
    for (i=0; i<N; i++) {
        data[i] = non_det();
        data[i]=data[i];
    }

    /* It misses the last element, i.e., data[N0]*/
    for (i=0; i<N-1; i++)
        for (j=i+1; j<N-1; j++)
            if (data[i] > data[j]){
                tmp = data[i];
                data[i] = data[j];
                data[j] = tmp;
            }

    /* Check the array is sorted */
    for (i=0; i<N-1; i++)
        assert(data[i] <= data[i+1]);
}
```

• Total 6224 CNF clause with 19099 boolean propositional variables

• Theoretically, $2^{19099} \times (2.35 \times 10^{5749})$ choices should be evaluated!!!

SAT-based Model Checking for C Programs
Looking forward to 2016: Provable systems

- We are now able to prove significant properties of programs with millions of lines of code
- Software proof tools already used on large scale in Windows Vista
- Significant progress in specification and proof technologies
- New architectures for provable systems