Software Model Checking I
## Dynamic v.s. Static Analysis

<table>
<thead>
<tr>
<th>Pros</th>
<th>Static Analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Dynamic Analysis</strong> (i.e., testing)</td>
<td><strong>Static Analysis</strong> (i.e. model checking)</td>
</tr>
<tr>
<td>• Real result</td>
<td>• Complete analysis result</td>
</tr>
<tr>
<td>• No environmental limitation</td>
<td>• Fully automatic</td>
</tr>
<tr>
<td>• Binary library is ok</td>
<td>• Concrete counter example</td>
</tr>
<tr>
<td>Cons</td>
<td></td>
</tr>
<tr>
<td>• Incomplete analysis result</td>
<td>• Consumed huge memory space</td>
</tr>
<tr>
<td>• Test case selection</td>
<td>• Takes huge time for verification</td>
</tr>
<tr>
<td></td>
<td>• False alarms</td>
</tr>
</tbody>
</table>

**Pros**
- Real result
- No environmental limitation
- Binary library is ok

**Cons**
- Incomplete analysis result
- Test case selection
- Consumed huge memory space
- Takes huge time for verification
- False alarms
Motivation for Software Model Checking

• Data flow analysis (DFA): fastest & least precision
  – “May” analysis,
• Abstract interpretation (AI): fast & medium precision
  – Over-approximation & under-approximation
• Model checking (MC): slow & complete
  – Complete value analysis
  – No approximation

• Static analyzer & MC as a C debugger
  • Handling complex C structures such as pointer and array
    • DFA: might-be
    • AI: may-be
    • MC: can-be or should-be
Model Checking Background

- Undergraduate CS classes contributing to this area

- Discrete math
- Algorithm
- PL
- Automata

- OS
- System programming
- Cyber physical system
- Intro. to SE

- Embedded Systems
- Software Engineering
- Programming Languages
- Algorithms

- Requirement properties
- System modeling
- System spec.
- Req. spec.
- Logic: $\square(\Phi \rightarrow \Diamond \Omega)$

Model Checking

Counter example(s)
OK or
Operational Semantics of Software

• A system execution $\sigma$ is a sequence of states $s_0s_1...$
  – A state has an environment $\rho: \text{Var} \rightarrow \text{Val}$

• A system has its semantics as a set of system executions
active type A() {
byte x;
again:
    x++;
    goto again;
}

active type A() {
byte x;
again:
    x++;
    goto again;
}

active type B() {
byte y;
again:
    y++;
    goto again;
}

Example

\[\text{Diagram showing state transitions for active types A and B.} \]
Pros and Cons of Model Checking

• Pros
  – Fully automated and provide complete coverage
  – Concrete counter examples
  – Full control over every detail of system behavior
    • Highly effective for analyzing
      – embedded software
      – multi-threaded systems

• Cons
  – State explosion problem
  – An abstracted model may not fully reflect a real system
  – Needs to use a specialized modeling language
    • Modeling languages are similar to programming languages, but simpler and clearer
Companies Working on Model Checking
Model Checking History

1981  Clarke / Emerson: CTL Model Checking
      Sifakis / Quielle
1982  EMC: Explicit Model Checker
      Clarke, Emerson, Sistla
1990  Symbolic Model Checking
      Burch, Clarke, Dill, McMillan
1992  SMV: Symbolic Model Verifier
      McMillan
1998  Bounded Model Checking using SAT
      Biere, Clarke, Zhu
2000  Counterexample-guided Abstraction Refinement
      Clarke, Grumberg, Jha, Lu, Veith
Example. Sort (1/2)

• Suppose that we have an array of 4 elements each of which is 1 byte long
  – unsigned char a[4];
• We want to verify sort.c works correctly
• Hash table based explicit model checker (ex. Spin) generates at least $2^{32}$ (= $4 \times 10^9 = 4G$) states
  • 4G states x 4 bytes = 16 Gbytes, no way...
• Binary Decision Diagram (BDD) based symbolic model checker (ex. NuSMV) takes 200 MB in 400 sec
1. #include <stdio.h>
2. #define N 5
3. int main(){
4.     int data[N], i, j, tmp;
5.     /* Assign random values to the array*/
6.     for (i=0; i<N; i++){
7.         data[i] = nondet_int();
8.     }
9.     /* It misses the last element, i.e., data[N-1]*/
10.    for (i=0; i<N-1; i++)
11.       for (j=i+1; j<N-1; j++)
12.           if (data[i] > data[j]){
13.               tmp = data[i];
14.               data[i] = data[j];
15.               data[j] = tmp;
16.           }
17.     /* Check the array is sorted */
18.     for (i=0; i<N-1; i++)
19.         assert(data[i] <= data[i+1]);
20. }
21. }

• SAT-based Bounded Model Checker
  • Total 19099 CNF clause with 6224 boolean propositional variables
  • Theoretically, $2^{6224}$ choices should be evaluated!!!
Overview of SAT-based Bounded Model Checking

Requirements → Formal Requirement Properties
\( \square (\Phi \rightarrow \Diamond \Omega) \)

C Program → Abstract Model

Model Checker

- Satisfied
- Not satisfied
- Okay
- Counter example

Requirements → Formal Requirement Properties in C
(ex. assert( x < a[i]); )

C Program

Translation to SAT formula → SAT Solver

- Satisfied
- Not satisfied
- Okay
- Counter example
SAT Basics (1/3)

- **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 (2/3)

• A set of propositional variables and Conjunctive Normal Form (CNF) 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 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
SAT Basics (3/3)

• DIMACS SAT Format

  – Ex. \((x_1 \lor x_2' \lor x_3)\)
    \(\land (x_2 \lor x_1' \lor x_4)\)

\[
p \text{ cnf 4 2} \\
1 -2 3 0 \\
2 -1 4 0
\]
Software Model Checking as a SAT problem (1/4)

• Control-flow simplification
  – All side effect are removed
    • \texttt{i++} \Rightarrow \texttt{i=i+1};
  – Control flow is made explicit
    • \texttt{continue, break} \Rightarrow \texttt{goto}
  – Loop simplification
    • \texttt{for(;;), do {...} while()} \Rightarrow \texttt{while()}

Software Model Checking as a SAT problem (2/4)

• Unwinding Loop

Original code

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

Unwinding the loop 1 times

```plaintext
x=0;
if (x < 2) {
    y = y + x;
    x++;
}
/* Unwinding assertion */
assert(! (x < 2))
```

Unwinding the loop 3 times

```plaintext
x=0;
if (x < 2) {
    y = y + x;
    x++;
}
if (x < 2) {
    y = y + x;
    x++;
}
if (x < 2) {
    y = y + x;
    x++;
}
/* Unwinding assertion */
assert (! (x < 2))
```
Examples

/* Straight-forward constant upperbound */
for(i=0, j=0; i < 5; i++) {
    j = j + i;
}

/* Constant upperbound */
for(i=0, j=0; j < 10; i++) {
    j = j + i;
}

/* Complex upperbound */
for(i=0; i < 5; i++) {
    for(j=i; j < 5; j++) {
        for(k = i+j; k < 5; k++) {
            m += i+j+k;
        }
    }
}

/* Upperbound unknown */
for(i=0, j=0; i^6-4*i^5 -17*i^4 != 9604 ; i++) {
    j = j + i;
}
Model Checking as a SAT problem (3/4)

- From C Code to SAT Formula

Original code

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

Convert to static single assignment (SSA)

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

Generate constraints

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

\[
P \equiv x_4 <= 3
\]

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 $x_0x_1x_2, y_0y_1y_2, z_0z_1z_2$

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

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

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

/* Assume that x and y are 2 bit unsigned integers */
/* Also assume that x+y <= 3 */
void f(unsigned int y) {
    unsigned int x=1;
    x=x+y;
    if (x==2)
        x+=1;
    else
        x=2;
    assert(x ==2);
}

C Bounded Model Checker

• Targeting arbitrary ANSI-C programs
  – Bit vector operators (>>, <<, |, &)
  – Array
  – Pointer arithmetic
  – Dynamic memory allocation
  – Floating #

• Can check
  – Array bound checks (i.e., buffer overflow)
  – Division by 0
  – Pointer checks (i.e., NULL pointer dereference)
  – Arithmetic overflow/underflow
  – User defined assert(cond)

• Handles function calls using inlining
• Unwinds the loops a fixed number of times
Modeling with CBMC

• Models an environment (i.e., various scenarios) using non-determinism
  1. By using undefined functions
  2. By using uninitialized local variables
  3. By using function parameters
  4. By explicitly using `__CPROVER_assume()`

```c
foo(int x) {
    __CPROVER_assume (0<x && x<10);
    x++;
    assert (x*x <= 100);
}

bar() {
    int y=0;
    __CPROVER_assume ( y > 10);
    assert(0);
}

int x = nondet();
bar() {
    int y;
    __CPROVER_assume (0<x && 0<y);
    if(x < 0 && y < 0)
        assert(0);
}
```