Bounded Model Checking of C Programs

Carsten Sinz • University of Karlsruhe / KIT

Introductory Example

- What does this function compute?

```
int next_power_of_two(int x)
{
  int i;
  x--;
  for(i=1; i < sizeof(int)*8; i *= 2)
    x = x | (x >> i);
  return x+1;
}
```

- next_power_of_two(5)=8
- next_power_of_two(1024)=1024
- How can we verify that this function is correct?

C Bounded Model Checking: Outline

0. Add specification using *assume/assert* statements
1. Inline functions (up to a fixed bound *b*)
2. Unroll loops (up to a fixed bound *b*)
   - All program paths are now finite
3. Convert program to *single static assignment* (SSA) form (by renaming variables)
   - Each variable (besides arrays) is written at most once
4. Convert program to set of bit-vector equations
5. Pass equations to an SMT solver *or*
   Convert equations to propositional logic and use a SAT solver

Software BMC by Example: Add Specification

```
int next_power_of_two(int x)
{
  int i;
  x--;
  for(i=1; i < sizeof(int)*8; i *= 2)
    x = x | (x >> i);
  return x+1;
}

int main(void)
{
  int x;
  assume(4096 < x && x <= 8192);
  assert(next_power_of_two(x) == 8192);
}
```

Software BMC by Example: Inline Functions

```
int next_power_of_two(int x)
{
  int i;
  x--;
  for(i=1; i < sizeof(int)*8;
      i *= 2)
    x = x | (x >> i);
  return x+1;
}

int main(void)
{
  int x;
  assume(4096 < x && x <= 8192);
  assert(next_power_of_two(x) ==
                           8192);
}
```

```
int main(void)
{
  int x, i, rt;
  assume(4096 < x && x <= 8192);
  x--;
  for(i=1; i < sizeof(int)*8;
      i *= 2)
    x = x | (x >> i);
  rt = x+1;
  assert(rt == 8192);
}
```

Software BMC by Example: Unroll Loops

```
int main(void)
{
  int x, i, rt;
  assume(4096 < x && x <= 8192);
  x--;
  for(i=1; i < sizeof(int)*8;
      i *= 2)
    x = x | (x >> i);
  rt = x+1;
  assert(rt == 8192);
}
```

```
int main(void)
{
  int x, rt;
  assume(4096 < x &&
                x <= 8192);
  x--;
  x = x | (x >> 1);
  x = x | (x >> 2);
  x = x | (x >> 4);
  x = x | (x >> 8);
  x = x | (x >> 16);
  rt = x+1;
  assert(rt == 8192);
}
```

Software BMC by Example: Convert to Singe Static Assignment Form

```
int main(void)
{
  int x, rt;
  assume(4096 < x &&
                x <= 8192);
  x--;
  x = x | (x >> 1);
  x = x | (x >> 2);
  x = x | (x >> 4);
  x = x | (x >> 8);
  x = x | (x >> 16);
  rt = x+1;
  assert(rt == 8192);
}
```

```
int main(void)
{
  int x0, x1, x2, x3, x4, x5,
      x6, rt;
  assume(4096 < x0 && x0 <= 8192);
  x1 = x0-1;
  x2 = x1 | (x1 >> 1);
  x3 = x2 | (x2 >> 2);
  x4 = x3 | (x3 >> 4);
  x5 = x4 | (x4 >> 8);
  x6 = x5 | (x5 >> 16);
  rt = x6+1;
  assert(rt == 8192);
}
```

Software BMC by Example: Convert to Bit-Vector Equations

```
int main(void)
{
  int x0, x1, x2, x3, x4, x5,
      x6, rt;
  assume(4096 < x0 &&
                x0 <= 8192);
  x1 = x0-1;
  x2 = x1 | (x1 >> 1);
  x3 = x2 | (x2 >> 2);
  x4 = x3 | (x3 >> 4);
  x5 = x4 | (x4 >> 8);
  x6 = x5 | (x5 >> 16);
  rt = x6+1;
  assert(rt == 8192);
}
```

```
bv32lt(4096, x0)
bv32le(x0, 8192)
x1 = bv32sub(x0, 1)
x2 = bv32or(x1, bv32ashr(x1, 1))
x3 = bv32or(x2, bv32ashr(x2, 2))
x4 = bv32or(x3, bv32ashr(x3, 4))
x5 = bv32or(x4, bv32ashr(x4, 8))
x6 = bv32or(x5, bv32ashr(x5, 16))
rt = bv32add(x6, 1)
rt ≠ 8192
```

How to Handle Arrays, Pointers, …

- Use theories for arrays, pointers, structs, unions,…
- For example, theory of arrays [McCarthy, 1962]:

$$
\begin{array}{lcl}
(\mathbf{A1}) & a = b \wedge i = j & \Rightarrow read(a,i) = read(b,j) \\
(\mathbf{A2}) & i = j & \Rightarrow read(write(a,i,e),j) = e \\
(\mathbf{A3}) & i \neq j & \Rightarrow read(write(a,i,e),j) = read(a,j) \\
(\mathbf{A4}) & \forall i\ (read(a,i) = read(b,i)) & \Rightarrow a = b
\end{array}
$$

- Alternative:
  - Use compiler front-end (e.g. LLVM)
  - Convert to assembler-like intermediate language (*abstract assembler*) and check abstract assembler program

Handling Pointers by Conversion to Abstract Assembler

```
int main(void)
{
        int x=nondet_int(), y=nondet_int();
        int *p = &x;
        int **q = &p;
        p = &y;
        assert(**q == y);
}
```

```
store i32* %x, i32** %p
store i32** %p, i32*** %q
store i32* %y, i32** %p
%tmp1 = load i32*** %q          ; <i32**> [#uses=1]
%tmp2 = load i32** %tmp1        ; <i32*> [#uses=1]
%tmp3 = load i32* %tmp2         ; <i32> [#uses=1]
%tmp4 = load i32* %y            ; <i32> [#uses=1]
assert(tmp3 == tmp4)
```

Abstract Assembler: Convert to Theory of Arrays

```
store i32* %x, i32** %p
store i32** %p, i32*** %q
store i32* %y, i32** %p
%tmp1 = load i32*** %q          ; <i32**> [#uses=1]
%tmp2 = load i32** %tmp1       ; <i32*> [#uses=1]
%tmp3 = load i32* %tmp2        ; <i32> [#uses=1]
%tmp4 = load i32* %y           ; <i32> [#uses=1]
assert(tmp3 == tmp4)
```

```
m1 = write(m0, p.adr, x)
m2 = write(m1, q.adr, p.adr)
m3 = write(m2, p.adr, y)
t1 = read(m3, q.adr)
t2 = read(m3, t1)
t3 = read(m3, t2)
t4 = read(m3, y)
t3 ≠ t4
```

Apply Array Axioms as Rewrite Rules

```
m1 = write(m0, p.adr, x)
m2 = write(m1, q.adr, p.adr)
m3 = write(m2, p.adr, y)
t1 = read(m3, q.adr)
t2 = read(m3, t1)
t3 = read(m3, t2)
t4 = read(m3, y)
t3 ≠ t4
```

```
t3 = read(m3, t2) = read(write(m2, p.adr, y), t2)
   = ITE(p.adr = t2; y; read(m2, t2))
   = ...
t4 = ...
t3 ≠ t4
```

How to Deal with Large Programs?

- Modular verification
  - Verify only smaller fractions of the program (write "generalized test cases")
- Abstraction:
  - Replace functions by specifications (pre/post-conditions)
  - Use loop invariants
- But what about soundness / completeness?
  - Soundness: *all reported errors are real errors (no false positives)*
  - Completeness: *no errors are missed (no false negatives)*
  - **Abstraction can lead to both unsound (false positives) and incomplete (false negatives) behavior!**

False Positives / Negatives

- Example: memcpy function
  - `void *memcpy(void *target, void *source, int n)`
  - specification: target[0] = source[0]
  - Specification misses the fact that target[i]=source[i] for $0<i\leq n$.
- False Positive:
  - assert(n==0 || target[1] == source[1])
  - Assertion violation impossible in real program, but possible with given specification (*incomplete specification*)
- False Negative:
  - A function may have side effects, modifying global variables
  - Such a side effect can lead to an error in the program, but not when the specification is used (and thus the global variable is not modified)

Insufficient Loop Bound

- May lead to false negatives
- **Example**: `while(x > n) { b; }`
- `if(x>n) b; …; if(x>n) b; assert(x ≤ n);`
  → detects insufficient loop bound
  (but interrupts verification at this point)
- `if(x>n) b; ...; if(x>n) b; assume(x ≤ n);`
  → may lead to false negatives
  (some computation paths may be ignored)

Applications of Software BMC

- Verification of Linux device drivers
  [Post, Sinz, Küchlin, 2007 & 2009]
- Verification of automotive embedded software (driver assistance systems)
  [Post, Sinz; 2008; Post, Sinz, Kaiser, Gorges; 2008]
- Equivalence checking of C implementations (cryptographic software)
  [Post, Sinz; 2009]