18. Bounded Model Checking of Software

Computer-Aided Verification

Dave Parker

University of Birmingham

2017/18

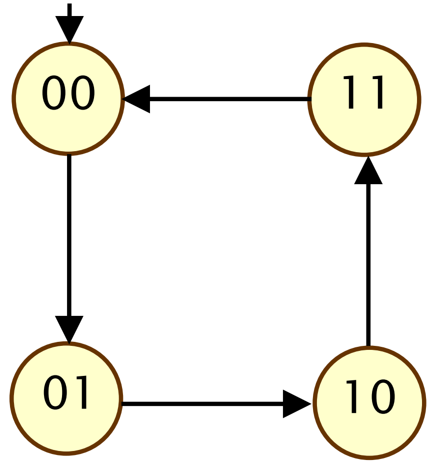
Module syllabus

- Modelling sequential and parallel systems
  - labelled transition systems, linear-time properties
- Temporal logic
  - LTL, CTL and CTL*, etc.
- Model checking
  - CTL model checking
  - automata-theoretic model checking (LTL)
- Quantitative verification
  - probabilistic (and timed) systems
- Verification tools
- Advanced verification techniques
  - bounded model checking via propositional satisfiability

Bounded model checking via SAT

- Main steps
  - bounded unfolding (depth k)
  - encode in propositional logic (CNF)
  - reduction to (efficient) SAT problem

- Key ideas
  - verification/falsification
  - bug hunting
  - negate property to check
  - symbolic approach



invariant: “the counter is always less than 3”

$P_i = \neg(l_i \wedge r_i)$

$\text{Init} \wedge (T_1 \wedge \ldots \wedge T_k) \wedge (\neg P_0 \vee \neg P_1 \vee \ldots \vee \neg P_k)$

E.g. (k=2): $(\neg l_0 \wedge \neg r_0) \wedge (l_1 = (l_0 \neq r_0) \wedge r_1 = \neg r_0) \wedge (l_2 = (l_1 \neq r_1) \wedge r_2 = \neg r_1) \wedge ((l_0 \wedge r_0) \vee (l_1 \wedge r_1) \vee (l_2 \wedge r_2))$

Software model checking

- Simple example program
  - x and y are integer variables

```
x := x + y;
if (x ≠ 1) {
        x := 2;
} else {
        x++;
}
assert (x≤3);
```

- Notice
  - simple imperative language (close to Java, C++)
  - variables can be uninitialised
  - properties specified as assertions

Overview

- Bounded model checking of software

- Main steps:
  - control flow simplification
  - loop unwinding
  - conversion to single static assignment form
  - conversion to conjunctive normal form (CNF)
  - solution using SAT (or SMT) solvers

Control flow simplification

- Simplify structure of code for easier analysis
  - convert to programs comprising while loops, ifs and gotos
  - ensure expressions are side-effect free

```
count := 0;
for (i := 1; i ≤ n; i++) {
	if (a[i] = x) {
		j := count++;
		if (count ≥ 10) {
			break;
		}
	}
}
```

Control flow simplification

- Simplify structure of code for easier analysis
  - convert to programs comprising while loops, ifs and gotos
  - ensure expressions are side-effect free

```
count := 0;
for (i := 1; i ≤ n; i++) {
    if (a[i] = x) {
        j := count++;
        if (count ≥ 10) {
            break;
        }
    }
}
```

→

```
count := 0;
i := 1;
while (i ≤ n) {
    if (a[i] = x) {
        j := count;
        count := count+1;
        if (count ≥ 10) {
            goto loop_exit;
        }
    }
    i := i+1;
}
loop_exit:
```

Loop unwinding

- Convert to loop-free program
  - unwinding loops to a fixed depth k
  - (recall: only need to consider while loops + gotos)

- 1 unwinding:

```
while (condition) {
      body
}
statements
```

→

```
if (condition)
      body
      while (condition) {
            body
      }
}
statements
```

Loop unwinding

- 2 unwindings:

```
while (condition) {
    body
}
statements
```

→

```
if (condition) {
    body
    if (condition)
        body
        while (condition) {
            body
        }
    }
}
statements
```

Loop unwinding

- 2 unwindings (assume k=2):

```
while (condition) {
    body
}
statements
```

```
if (condition) {
    body
    if (condition)
        body
    }
}
statements
```

Loop unwinding

- 2 unwindings (k=2, with unwinding assertion):

```
while (condition) {
        body
}
statements
```

```
if (condition)
        body
        if (condition)
                body
                assert (!condition)
        }
}
statements
```

- Bounded model checking: k unwindings of all loops
  - note: <u>not</u> just depth-bounded search of program state space

Single static assignments

- Single static assignment (SSA) form
  - used as an intermediate representation in program analysis
  - every variable seen exactly once
  - multiple versions of each variable created
- Conversion to SSA:
  - each assignment of variable uses a new version
  - each access of a variable uses the latest version

```
x := 2 * y;
x := x * z;
x++;
```

→

```
x1 := 2 * y0;
x2 := x1 * z0;
x3 := x2 + 1;
```

Single static assignments

- SSA form conversion for conditionals:
  - (recall: while loops unwound into if statements)

```
if (x > z) {
        y := x;
} else {
        y := z + 1;
}
w := 2 * y;
```

→

```
if (x0 > z0) {
        y1 := x0;
} else {
        y2 := z0 + 1;
}
y3 := (x0 > z0) ? y1 : y2;
w1 := 2 * y3;
```

Conversion to CNF

- Conversion to conjunctive normal form (CNF)
- Simple example program from earlier

```
x := x + y;
if (x ≠ 1) {
        x := 2;
} else {
        x++;
}
assert (x≤3);
```

→

```
x1 := x0 + y0;
if (x1 ≠ 1) {
        x2 := 2;
} else {
        x3 := x1+1;
}
x4 := (x1≠1) ? x2 : x3
assert (x4≤3);
```

→

$(x_1 = x_0 + y_0) \land$
$(x_2 = 2) \land$
$(x_3 = x_1 + 1) \land$
$(x_4 = (x_1 \neq 1) ? x_2 : x_3) \land$
$\neg(x_4 \leq 3)$

$T \land \neg P$

- Formula over predicates, not Boolean variables

Checking satisfiability

- Checking satisfiability of formulae over predicates

- Bit blasting
  - convert integers to usual binary encoding
  - map integer variables from predicates to Boolean variables
  - solve using standard SAT solver
  - precise, faithful encoding of program

- SMT (satisfiability modulo theories) solvers
  - as for SAT, many efficient algorithms/tools developed
  - infinite-ranging variables, richer data types
  - but may not match e.g. execution of real C code

Example: Z3 (SMT)

```
(declare-const x0 Int)
(declare-const y0 Int)
(declare-const x1 Int)
(declare-const x2 Int)
(declare-const x3 Int)
(declare-const x4 Int)
(define-fun conjecture () Bool
  (and
    (= x1 (+ x0 y0))
  and
    (= x2 2)
  and
    (= x3 (+ x1 1))
  and
    (= x4 (ite (not (= x1 1)) x2 x3))
    (not (<= x4 3))
  )
)
(assert conjecture)
(check-sat)
(get-model)
```

Example

- Apply bounded model check, unrolling loops to depth 2

```
for (i := 1; i ≤ 10; i++) {
        if (a[i] > j) {
                j := a[i];
        }
}
assert j>0;
```

Example

- Apply bounded model check, unrolling loops to depth 2

```
for (i := 1; i ≤ 10; i++) {
        if (a[i] > j) {
                j := a[i];
        }
}
assert j>0;
```

→

```
i := 1;
while (i ≤ 10) {
        if (a[i] > j) {
                j := a[i];
        }
        i := i + 1;
}
assert j>0
```

```
for (i := 1; i ≤ 10; i++) {
        if (a[i] > j) {
                j := a[i];
        }
}
assert j>0;
```

```
i := 1;
while (i ≤ 10) {
        if (a[i] > j) {
                j := a[i];
        }
        i := i + 1;
}
assert j>0
```

```
i1 := 1;
if (i1 ≤ 10) {
        if (a0[i1] > j0) {
                j1 := a0[i0];
        }
        j2 := (a0[i1] > j0) ? j1 : j0
        i2 := i1 + 1;
        if (i2 ≤ 10) {
                if (a[i2] > j2) {
                        j3 := a0[i2];
                }
                j4 := (a0[i2] > j2) ? j3 : j2
                i3 := i2 + 1;
        }
        j5 := (i2 ≤ 10) ? j4 : j2
        i4 := (i2 ≤ 10) ? i3 : i2
}
j6 := (i1 ≤ 10) ? j5 : j0
i5 := (i1 ≤ 10) ? i4 : i1
assert j6>0;
```

$$(i_1 = 1) \wedge (j_1 = a_0[i_0]) \wedge (j_2 = a_0[i_0] > j_0 \,?\, j_1 : j_0) \wedge \ldots \wedge \neg(j_6 > 0)$$

Summary

- Bounded model checking (for software)
  - bounded search for bugs by unwind program loops
  - incomplete (without extensions) but effective in practice
  - relies on translation to efficiently solvable problem (SAT/SMT)

- Main steps:
  - control flow simplification
  - loop unwinding
  - conversion to single static assignment form
  - conversion to conjunctive normal form (CNF)
  - solution using SAT (or SMT) solvers