Practical Applications of SAT
Today

Past 2 lectures

• The theory and mechanics of SAT solving

Today

• Practical applications of SAT
• Variants of the SAT problem
• Motivating the next lecture on SMT
A brief history of SAT solving and applications

Bounded Model Checking.
First presented at FMCAD’98. In an unusual move, the Chairs included an extra talk on BMC. A 1999 paper describes its application at Motorola to verify a PowerPC processor.

SAT solver on board Deep Space One.

zChaff, ‘01

MiniSAT, ’03

Concolic Testing, Program Analysis, Mercedes Product Configuration

Synthesis, Type Systems, Bio, Configuration Management, SMT

Based on a slide from Vijay Ganesh
Bounded Model Checking (BMC) & Configuration Management
Bounded Model Checking (in general)

Given a system and a property, BMC checks if the property is satisfied by all executions of the system with \( \leq k \) steps, on all inputs of size \( \leq n \).

We will focus on safety properties (i.e., making sure a bad state, such as an assertion violation, is not reached).
Bounded Model Checking (in general)

Testing: checks a few executions of arbitrary size

BMC: checks all executions of size $\leq k$

Verification: checks all executions of every size

The **small scope hypothesis**: most bugs can be triggered with small inputs and executions.

- **low confidence**: low human labor
- **high confidence**: high human labor
BMC by example
The Zune Bug: on December 31, 2008, all first generation Zune players from Microsoft became unresponsive because of this code. What’s wrong?

Infinite loop triggered on the last day of every leap year.

A desired safety property: the value of the days variable decreases in every loop iteration.
BMC step 1 of 4: finitize loops

```c
int daysToYear(int days) {
    int year = 1980;
    if (days > 365) {
        int oldDays = days;
        if (isLeapYear(year)) {
            if (days > 366) {
                days -= 366;
                year += 1;
            }
        } else {
            days -= 365;
            year += 1;
        }
        assert days < oldDays;
    }
    assert days <= 365;
    return year;
}
```

- Unwind all loops \( k \) times (e.g., \( k=1 \)), and add an **unwinding assertion** at the end.
- If a CEX violates a program assertion, we have found a buggy behavior of length \( \leq k \).
- If a CEX violates an unwinding assertion, the program has no buggy behavior of length \( \leq k \), but it may have a longer one.
- If there is no CEX, the program is correct for all \( k \)!
int daysToYear(int days) {
    int year = 1980;
    if (days > 365) {
        int oldDays = days;
        if (isLeapYear(year)) {
            if (days > 366) {
                days -= 366;
                year += 1;
            }
        } else {
            days -= 365;
            year += 1;
        }
        assert days < oldDays;
    }
    assert days <= 365;
    return year;
}
BMC step 2 of 4: eliminate side effects

```c
int daysToYear(int days) {
    int year = 1980;
    if (days > 365) {
        int oldDays = days;
        if (isLeapYear(year)) {
            if (days > 366) {
                days -= 366;
                year += 1;
            }
        } else {
            days -= 365;
            year += 1;
        }
        assert days < oldDays;
        assert days <= 365;
    }
    return year;
}
```
int days;
int year = 1980;
if (days > 365) {
    int oldDays = days;
    if (isLeapYear(year)) {
        if (days > 366) {
            days = days - 366;
            year = year + 1;
        }
    } else {
        days = days - 365;
        year = year + 1;
    }
    assert days < oldDays;
    assert days <= 365;
}
return year;

BMC step 2 of 4: eliminate side effects

Convert to Static Single Assignment (SSA) form:
int days₀;
int year₀ = 1980;
if (days₀ > 365) {
    int oldDays₀ = days₀;
    if (isLeapYear(year₀)) {
        if (days₀ > 366) {
            days₁ = days₀ - 366;
            year₁ = year₀ + 1;
        }
    } else {
        days₃ = days₀ - 365;
        year₃ = year₀ + 1;
    }
    assert days₄ < oldDays₀;
    assert days₄ <= 365;
}
return year₅;
BMC step 2 of 4: eliminate side effects

```java
int days0;
int year0 = 1980;
boolean g0 = (days0 > 365);
int oldDays0 = days0;
boolean g1 = isLeapYear(year0);
boolean g2 = days0 > 366;
days1 = days0 - 366;
year1 = year0 + 1;
days2 = \phi(g1 && g2, days1, days0);
year2 = \phi(g1 && g2, year1, year0);
days3 = days0 - 365;
year3 = year0 + 1;
days4 = \phi(g1, days2, days3);
year4 = \phi(g1, year2, year3);
assert days4 < oldDays0;
assert days4 <= 365;
year5 = \phi(g0, year4, year0);
return year5;
```

Convert to **Static Single Assignment** (SSA) form:

- Replace each assignment to a variable \( v \) with a definition of a fresh variable \( v_i \).
- Change uses of variables so that they refer to the correct definition (version).
- Make conditional dependences explicit with gated \( \phi \) nodes.
BMC step 2 of 4: eliminate side effects

```java
int days0;
int year0 = 1980;
boolean g0 = (days0 > 365);
int oldDays0 = days0;
boolean g1 = isLeapYear(year0);
boolean g2 = days0 > 366;
days1 = days0 - 366;
year1 = year0 + 1;
days2 = φ(g1 && g2, days1, days0);
year2 = φ(g1 && g2, year1, year0);
days3 = days0 - 365;
year3 = year0 + 1;
days4 = φ(g1, days2, days3);
year4 = φ(g1, year2, year3);
assert days4 < oldDays0;
assert days4 <= 365;
year5 = φ(g0, year4, year0);
return year5;
```
We can now read off the equations that encode the program semantics, and the assertions to be checked.
int year_0 = 1980;
boolean g_0 = (days_0 > 365);
int oldDays_0 = days_0;
boolean g_1 = isLeapYear(year_0);
boolean g_2 = days_0 > 366;
days_1 = days_0 - 366;
year_1 = year_0 + 1;
days_2 = φ(g_1 && g_2, days_1, days_0);
year_2 = φ(g_1 && g_2, year_1, year_0);
days_3 = days_0 - 365;
year_3 = year_0 + 1;
days_4 = φ(g_1, days_2, days_3);
year_4 = φ(g_1, year_2, year_3);
assert days_4 < oldDays_0;
assert days_4 <= 365;

We can now read off the equations that encode the program semantics …
BMC step 3 of 4: convert into equations

\[
\begin{align*}
\text{year}_0 &= 1980; \\
g_0 &= (\text{days}_0 > 365); \\
\text{oldDays}_0 &= \text{days}_0; \\
g_1 &= \text{isLeapYear(}\text{year}_0); \\
g_2 &= \text{days}_0 > 366; \\
\text{days}_1 &= \text{days}_0 - 366; \\
\text{year}_1 &= \text{year}_0 + 1; \\
\text{days}_2 &= \phi(g_1 \&\& g_2, \text{days}_1, \text{days}_0); \\
\text{year}_2 &= \phi(g_1 \&\& g_2, \text{year}_1, \text{year}_0); \\
\text{days}_3 &= \text{days}_0 - 365; \\
\text{year}_3 &= \text{year}_0 + 1; \\
\text{days}_4 &= \phi(g_1, \text{days}_2, \text{days}_3); \\
\text{year}_4 &= \phi(g_1, \text{year}_2, \text{year}_3); \\
\text{assert} \; \text{days}_4 &< \text{oldDays}_0; \\
\text{assert} \; \text{days}_4 &\leq 365;
\end{align*}
\]
We can now read off the equations that encode the program semantics …

\[
\begin{align*}
\text{year}_0 &= 1980 \land \\
g_0 &= (\text{days}_0 > 365) \land \\
\text{oldDays}_0 &= \text{days}_0 \land \\
g_1 &= \text{isLeapYear}(\text{year}_0) \land \\
g_2 &= \text{days}_0 > 366 \land \\
\text{days}_1 &= \text{days}_0 - 366 \land \\
\text{year}_1 &= \text{year}_0 + 1 \land \\
\text{days}_2 &= \phi(g_1 \land g_2, \text{days}_1, \text{days}_0) \land \\
\text{year}_2 &= \phi(g_1 \land g_2, \text{year}_1, \text{year}_0) \land \\
\text{days}_3 &= \text{days}_0 - 365 \land \\
\text{year}_3 &= \text{year}_0 + 1 \land \\
\text{days}_4 &= \phi(g_1, \text{days}_2, \text{days}_3) \land \\
\text{year}_4 &= \phi(g_1, \text{year}_2, \text{year}_3) \land \\
\text{assert} \; \text{days}_4 &< \text{oldDays}_0; \\
\text{assert} \; \text{days}_4 &\leq 365;
\end{align*}
\]
year₀ = 1980 ∧
g₀ = (days₀ > 365) ∧
oldDays₀ = days₀ ∧
g₁ = isLeapYear(year₀) ∧
g₂ = days₀ > 366 ∧
days₁ = days₀ - 366 ∧
year₁ = year₀ + 1 ∧
days₂ = \text{ite}(g₁ ∧ g₂, \text{days₁}, \text{days₀}) ∧
year₂ = \text{ite}(g₁ ∧ g₂, \text{year₁}, \text{year₀}) ∧
days₃ = days₀ - 365 ∧
year₃ = year₀ + 1 ∧
days₄ = \text{ite}(g₁, \text{days₂}, \text{days₃}) ∧
year₄ = \text{ite}(g₁, year₂, year₃) ∧
assert days₄ < oldDays₀;
assert days₄ <= 365;
BMC step 3 of 4: convert into equations

\[
\begin{align*}
\text{year}_0 &= 1980 \land \\
\text{g}_0 &= (\text{days}_0 > 365) \land \\
\text{oldDays}_0 &= \text{days}_0 \land \\
\text{g}_1 &= \text{isLeapYear}(\text{year}_0) \land \\
\text{g}_2 &= \text{days}_0 > 366 \land \\
\text{days}_1 &= \text{days}_0 - 366 \land \\
\text{year}_1 &= \text{year}_0 + 1 \land \\
\text{days}_2 &= \text{ite}(\text{g}_1 \land \text{g}_2, \text{days}_1, \text{days}_0) \land \\
\text{year}_2 &= \text{ite}(\text{g}_1 \land \text{g}_2, \text{year}_1, \text{year}_0) \land \\
\text{days}_3 &= \text{days}_0 - 365 \land \\
\text{year}_3 &= \text{year}_0 + 1 \land \\
\text{days}_4 &= \text{ite}(\text{g}_1, \text{days}_2, \text{days}_3) \land \\
\text{year}_4 &= \text{ite}(\text{g}_1, \text{year}_2, \text{year}_3) \land \\
(\neg(\text{days}_4 < \text{oldDays}_0) \lor \\
\neg(\text{days}_4 \leq 365))
\end{align*}
\]
Represent numbers as arrays of bits. Use one boolean variable per bit for each number.

\[
\text{year}_0 = 000 \ldots 000
\]

Construct an adder circuit for \(\text{year}_0 + 1\).

Introduce new clauses to constrain bits in \(\text{year}_1\) to match bits in the sum.
BMC counterexample for $k=1$

```c
int daysToYear(int days) {
    int year = 1980;
    while (days > 365) {
        if (isLeapYear(year)) {
            if (days > 366) {
                days -= 366;
                year += 1;
            }
        } else {
            days -= 365;
            year += 1;
        }
    }
    return year;
}
```

`days = 366`
Bounded Model Checking (BMC) & Configuration Management
Configuration Management

Given a configuration, consisting of a set of components, their dependencies, and conflicts:

• Decide if a new component can be added to the configuration.

• Add the component while optimizing some linear function.

• If the component cannot be added, find a way to add it by removing as few conflicting components from the current configuration as possible.
Deciding if a component can be installed

To install a, CNF constraints are:

\[(\neg a \lor b) \land (\neg a \lor c) \land (\neg a \lor z) \land (\neg b \lor d) \land (\neg c \lor d \lor e) \land (\neg c \lor f \lor g) \land (\neg d \lor \neg e) \land (\neg y \lor z) \land a \land z\]

Conflict: d and e cannot both be installed.
Assume f and g are 5MB and 2MB each, and all other components are 1MB. How to install a, while minimizing total size?

\[(\neg a \lor b) \land (\neg a \lor c) \land (\neg a \lor z) \land (\neg b \lor d) \land (\neg c \lor d \lor e) \land (\neg c \lor f \lor g) \land (\neg d \lor \neg e) \land (\neg y \lor z) \land a \land z\]
Assume f and g are 5MB and 2MB each, and all other components are 1MB. How to install a, while minimizing total size?

Pseudo-boolean solvers accept a linear function to minimize, in addition to a (weighted) CNF.

\[ \text{min } c_1x_1 + \ldots + c_nx_n \]
\[ a_1x_1 + \ldots + a_nx_n \geq b_1 \land \ldots \land \]
\[ a_kx_1 + \ldots + a_kx_n \geq b_k \]

\[ (\neg a \lor b) \land (\neg a \lor c) \land (\neg a \lor z) \land \]
\[ (\neg b \lor d) \land \]
\[ (\neg c \lor d \lor e) \land (\neg c \lor f \lor g) \land \]
\[ (\neg d \lor \neg e) \land \]
\[ (\neg y \lor z) \land \]
\[ a \land z \]
Optimal installation

Assume $f$ and $g$ are 5MB and 2MB each, and all other components are 1MB. How to install $a$, while minimizing total size?

Pseudo-boolean solvers accept a linear function to minimize, in addition to a (weighted) CNF.

\[\text{min } a + b + c + d + e + 5f + 2g + y + 0z \]
\[(-a + b \geq 0) \land (-a + c \geq 0) \land (-a + z \geq 0) \land (-b + d \geq 0) \land (-c + d + e \geq 0) \land (-c + f + g \geq 0) \land (-d + -e \geq -1) \land (-y + z \geq 0) \land (a \geq 1) \land (z \geq 1)\]
Installation in the presence of conflicts

Partial MaxSAT solver takes as input a set of **hard** clauses and a set of **soft** clauses, and it produces an assignment that satisfies all hard clauses and the greatest number of soft clauses.

To install a, while minimizing the number of removed components, Partial MaxSAT constraints are:

**hard:** \((\neg a \lor b) \land (\neg a \lor c) \land (\neg a \lor z) \land (\neg b \lor d) \land (\neg c \lor d \lor e) \land (\neg c \lor f \lor g) \land (\neg d \lor \neg e) \land (\neg y \lor z) \land a\)

**soft:** \(e \land z\)

a cannot be installed because it requires b, which requires d, which conflicts with e.
Summary

Today

- SAT solvers have been used successfully in many applications & domains
- But reducing problems to SAT is a lot like programming in assembly …
- We need higher-level logics!

Next lecture

- On to richer logics: introduction to Satisfiability Modulo Theories (SMT)