Computer-Aided Reasoning for Software

Practical Applications of SAT

Emina Torlak
emina@cs.washington.edu
Today
Today

Past 2 lectures

- The theory and mechanics of SAT solving
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
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

But first …
• A brief Q&A session for Homework 1
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

But first ...
  • A brief Q&A session for Homework 1

Reminder
  • Email us your project topics by 11pm tonight
A brief history of SAT solving and applications

Based on a slide from Vijay Ganesh
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.

Based on a slide from Vijay Ganesh
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.
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

Based on a slide from Vijay Ganesh
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.

MiniSAT, ’03

zChaff, ‘01

Based on a slide from Vijay Ganesh
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.

MiniSAT, ’03

Concolic Testing, Program Analysis, Mercedes Product Configuration

zChaff, ‘01
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**
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 \( \leq n \).
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

Low confidence

High confidence

Low human labor

High human labor
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 says that many bugs can be triggered with small inputs and executions.

low confidence → low human labor

The small scope hypothesis says that many bugs can be triggered with small inputs and executions.

high confidence → high human labor
BMC by example
BMC by example

```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;
}
```
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;
}

The Zune Bug: on December 31, 2008, all first generation Zune players from Microsoft became unresponsive because of this code. What’s wrong?
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;
}

Infinite loop triggered on the last day of every leap year.
int daysToYear(int days) {
    int year = 1980;
    while (days > 365) {
        int oldDays = days;
        if (isLeapYear(year)) {
            if (days > 366) {
                days -= 366;
                year += 1;
            }
        } else {
            days -= 365;
            year += 1;
        }
        assert days < oldDays;
    }
    return year;
}
int daysToYear(int days) {
    int year = 1980;
    while (days > 365) {
        int oldDays = days;
        if (isLeapYear(year)) {
            if (days > 366) {
                days -= 366;
                year += 1;
            }
        } else {
            days -= 365;
            year += 1;
        }
        assert days < oldDays;
    }
    return year;
}
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;
        }
    } 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 after each.
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 after each.
• If a CEX violates a program assertion, we have found a buggy behavior of length \(\leq k\).
BMC step 1 of 4: finitize loops & inline calls

```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 after each.
- 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.
```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 after each.
- 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;
        }
    } 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

```c
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;
```

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

```c
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;
```

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).
BMC step 2 of 4: eliminate side effects

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 \( \varphi \) nodes.

```plaintext
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 = \varphi(g1 && g2, days1, days0)
year2 = \varphi(g1 && g2, year1, year0)
days3 = days0 - 365;
year3 = year0 + 1;
days4 = \varphi(g1, days2, days3)
year4 = \varphi(g1, year2, year3)
assert days4 < oldDays0;
assert days4 <= 365;
year5 = \varphi(g0, year4, year0)
return year5;
```
BMC step 2 of 4: eliminate side effects

```plaintext
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 year4;
```

```plaintext
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);
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;
```
BMC step 3 of 4: convert into equations

```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;
```
BMC step 3 of 4: convert into equations

\[
\begin{align*}
\text{year}_0 &= 1980 \land \\
\text{g}_0 &= (\text{days}_0 > 365) \land \\
\text{oldDays}_0 &= \text{days}_0 \land \\
\text{g}_1 &= \text{isLeapYear}(\text{year}_0) \land \\
\text{g}_2 &= \text{days}_0 > 366 \land \\
\text{days}_1 &= \text{days}_0 - 366 \land \\
\text{year}_1 &= \text{year}_0 + 1 \land \\
\text{days}_2 &= \text{ite}(\text{g}_1 \land \text{g}_2, \text{days}_1, \text{days}_0) \land \\
\text{year}_2 &= \text{ite}(\text{g}_1 \land \text{g}_2, \text{year}_1, \text{year}_0) \land \\
\text{days}_3 &= \text{days}_0 - 365 \land \\
\text{year}_3 &= \text{year}_0 + 1 \land \\
\text{days}_4 &= \text{ite}(\text{g}_1, \text{days}_2, \text{days}_3) \land \\
\text{year}_4 &= \text{ite}(\text{g}_1, \text{year}_2, \text{year}_3) \land \\
\text{year}_5 &= \text{ite}(\text{g}_0, \text{year}_4, \text{year}_0) \land \\
(\neg(\text{days}_4 < \text{oldDays}_0) \lor \\
\neg(\text{days}_4 \leq 365))
\end{align*}
\]

A solution to these equations is a sound counterexample: an interpretation for all logical variables that satisfies the program semantics (for up to k unwindings) but violates at least one of the assertions.
BMC step 4 of 4: convert into CNF

\[ \text{year}_1 = \text{year}_0 + 1 \]
BMC step 4 of 4: convert into CNF

\[
\text{year}_1 = \text{year}_0 + 1
\]

\[
\text{year}_0 = \underbrace{000 \ldots 000}_{31 \ 30 \ 29 \ 2 \ 1 \ 0}
\]

Represent numbers as arrays of bits, and create one propositional variable per bit for each number.
BMC step 4 of 4: convert into CNF

\[ \text{year}_1 = \text{year}_0 + 1 \]

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

Represent numbers as arrays of bits, and create one propositional variable per bit for each number.

\[ c_{32} \]
\[ c_{31} \]
\[ c_2 \]
\[ c_1 \]
\[ s_{31} \]
\[ s_1 \]
\[ s_0 \]
BMC step 4 of 4: convert into CNF

\[ \text{year}_1 = \text{year}_0 + 1 \]

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

Represent numbers as arrays of bits, and create one propositional variable per bit for each number.

\[ \text{year}_{0:31} 0 \]
\[ \text{year}_{0:1} 0 \]
\[ \text{year}_{0:0} 1 \]

\[ \text{year}_{1:31} \Leftrightarrow \text{s}_{31} \land \ldots \land \text{year}_{1:1} \Leftrightarrow \text{s}_{1} \land \text{year}_{1:0} \Leftrightarrow \text{s}_0 \]

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) {
        int oldDays = days;
        if (isLeapYear(year)) {
            if (days > 366) {
                days -= 366;
                year += 1;
            }
        } else {
            days -= 365;
            year += 1;
        }
        assert days < oldDays;
    }
    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.
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.
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.

SAT

Pseudo-Boolean Constraints
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
Deciding if a component can be installed

z already installed.
Deciding if a component can be installed

- a depends on b, c, z.
- z already installed.
Deciding if a component can be installed

- a depends on b, c, z.
- z already installed.
- c needs f or g.
Deciding if a component can be installed

Conflict: d and e cannot both be installed.
Deciding if a component can be installed

Conflict: d and e cannot both be installed.

To install a, CNF constraints are:

- a depends on b, c, z.
- z already installed.
- c needs f or g.
Deciding if a component can be installed

Conflict: d and e cannot both be installed.

To install a, CNF constraints are:

$$(\neg a \lor b) \land (\neg a \lor c) \land (\neg a \lor z) \land$$
Deciding if a component can be installed

To install a, CNF constraints are:
\[
(\neg a \lor b) \land (\neg a \lor c) \land (\neg a \lor z) \land \\
(\neg b \lor d) \land \\
\]

Conflict: d and e cannot both be installed.
Deciding if a component can be installed

To install a, CNF constraints are:
\[(\neg a \lor b) \land (\neg a \lor c) \land (\neg a \lor z) \land (\neg b \lor d) \land (\neg c \lor d \lor e) \land (\neg c \lor f \lor g) \land z \text{ already installed.}\]

Conflict: d and e cannot both be installed.
Deciding if a component can be installed

To install a, CNF constraints are:

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

Conflict: d and e cannot both be installed.
Deciding if a component can be installed

Conflict: d and e cannot both be installed.

To install a, CNF constraints are:
\[
\neg a \lor b \land (\neg a \lor c) \land (\neg a \lor z) \land \\
(\neg b \lor d) \land \\
(\neg c \lor d \lor e) \land (\neg c \lor f \lor g) \land \\
(\neg d \lor \neg e) \land \\
a \land z
\]
Optimal installation
Assume $f$ and $g$ are 5MB and 2MB each, and all other components are 1MB. To install $a$, while minimizing total size, pseudo-boolean constraints are:

Pseudo-boolean solvers accept a linear function to minimize, in addition to a (weighted) CNF.
Assume f and g are 5MB and 2MB each, and all other components are 1MB. To install a, while minimizing total size, pseudo-boolean constraints are:

\[
\min c_1 x_1 + \ldots + c_n x_n \\
a_1 x_1 + \ldots + a_1 n x_n \geq b_1 \\
\ldots \\
a_k x_1 + \ldots + a_k n x_n \geq b_k
\]
Assume f and g are 5MB and 2MB each, and all other components are 1MB. To install a, while minimizing total size, pseudo-boolean constraints are:

\[
\min a + b + c + d + e + 5f + 2g + y
\]

- \[
\min c_1 x_1 + \ldots + c_n x_n
\]
- \[
a_1 x_1 + \ldots + a_1 n x_n \geq b_1
\]
- \[
\ldots
\]
- \[
a_k x_1 + \ldots + a_k n x_n \geq b_k
\]
Assume f and g are 5MB and 2MB each, and all other components are 1MB. To install a, while minimizing total size, pseudo-boolean constraints are:

\[
\begin{align*}
\text{min } a + b + c + d + e + 5f + 2g + y \\
(-a + b \geq 0) \land (-a + c \geq 0) \land (-a + z \geq 0) \land \\
(-b + d \geq 0) \land \\
(-c + d + e \geq 0) \land (-c + f + g \geq 0) \land \\
(-d + -e \geq -1) \land \\
(a \geq 1) \land (z \geq 1)
\end{align*}
\]

\[
\begin{align*}
\text{min } c_1x_1 + \ldots + c_nx_n \\
a_1x_1 + \ldots + a_{1n}x_n \geq b_1 \\
\ldots \\
a_kx_1 + \ldots + a_{kn}x_n \geq b_k
\end{align*}
\]
Installation in the presence of conflicts
Installation in the presence of conflicts

A cannot be installed because it requires b, which requires d, which conflicts with e.
To install a, while minimizing the number of removed components, Partial MaxSAT constraints are:

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

**soft:** $e \land z$

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.
Installation in the presence of conflicts

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

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

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

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.
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)