Computer-Aided Reasoning for Software

CSE507

Practical Applications of SAT

courses.cs.washington.edu/courses/cse507/17wi/

Emina Torlak
emina@cs.washington.edu

Today

Past 2 lectures

- The theory and mechanics of SAT solving

Today

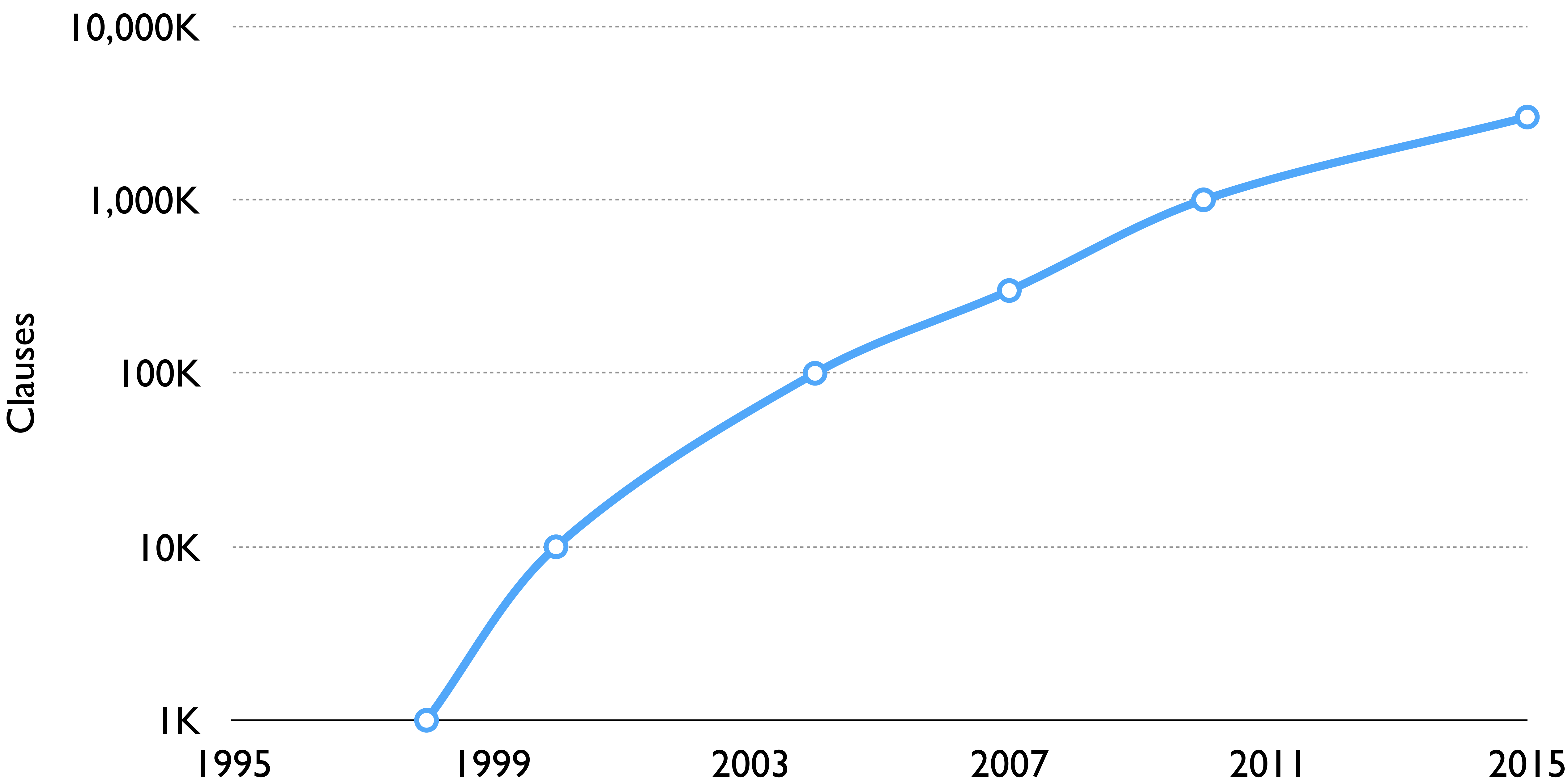
- Practical applications of SAT
- Variants of the SAT problem
- Motivating the next lecture on SMT

But first …

- A brief Q&A session for Homework 1

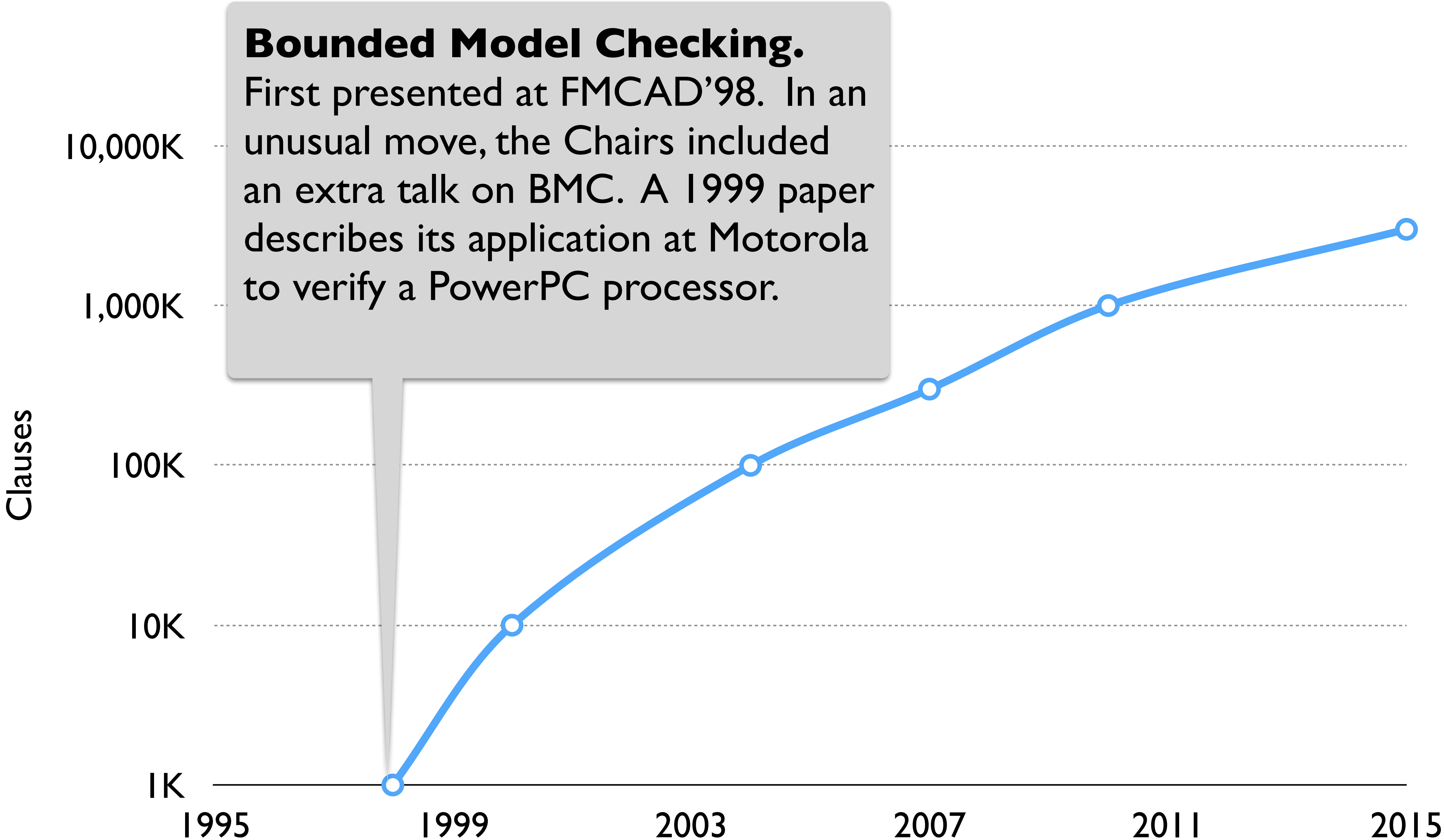
topics

A brief history of SAT solving and applications



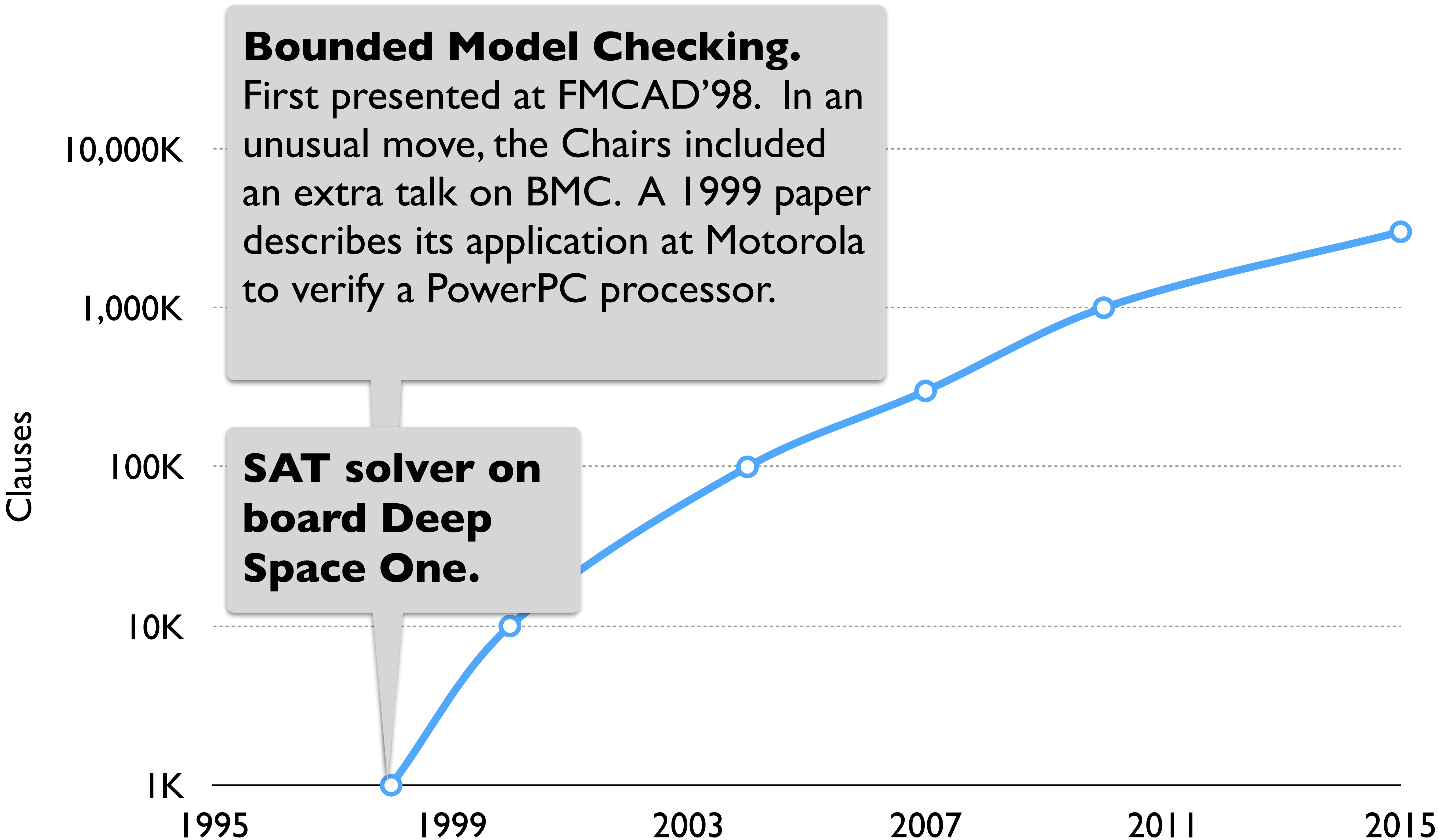
Based on a slide from Vijay Ganesh

A brief history of SAT solving and applications

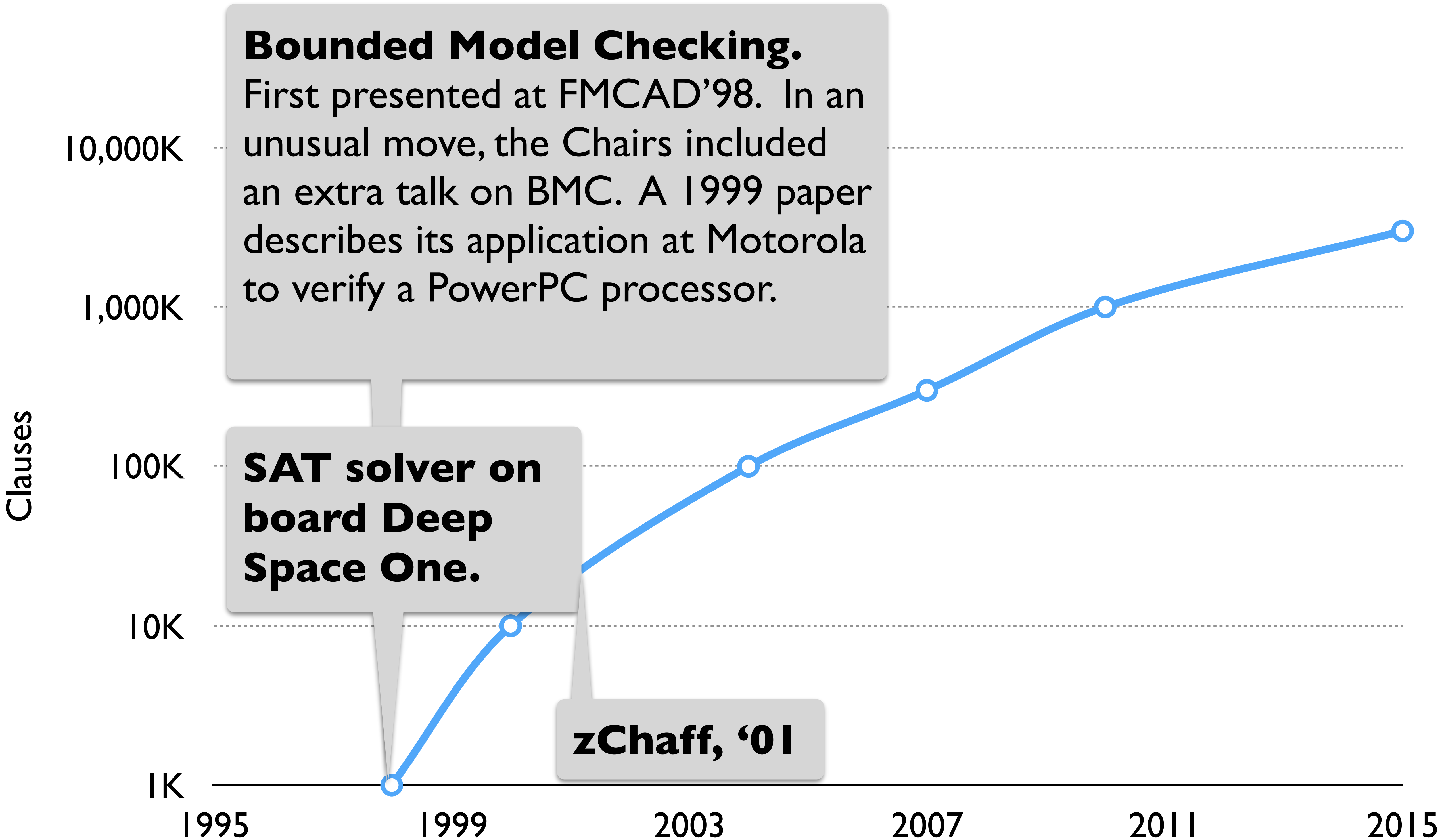


Based on a slide from Vijay Ganesh

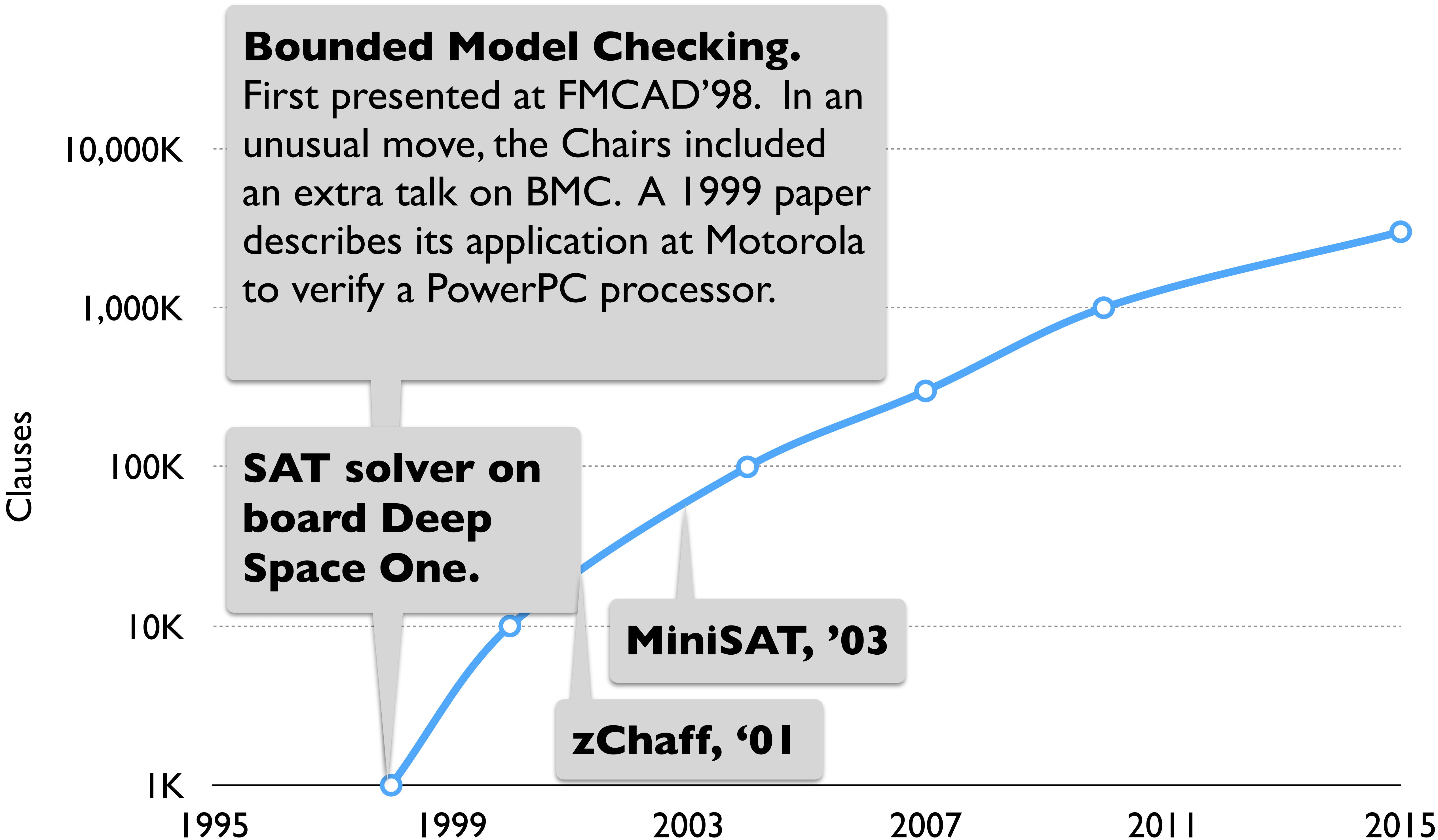
A brief history of SAT solving and applications



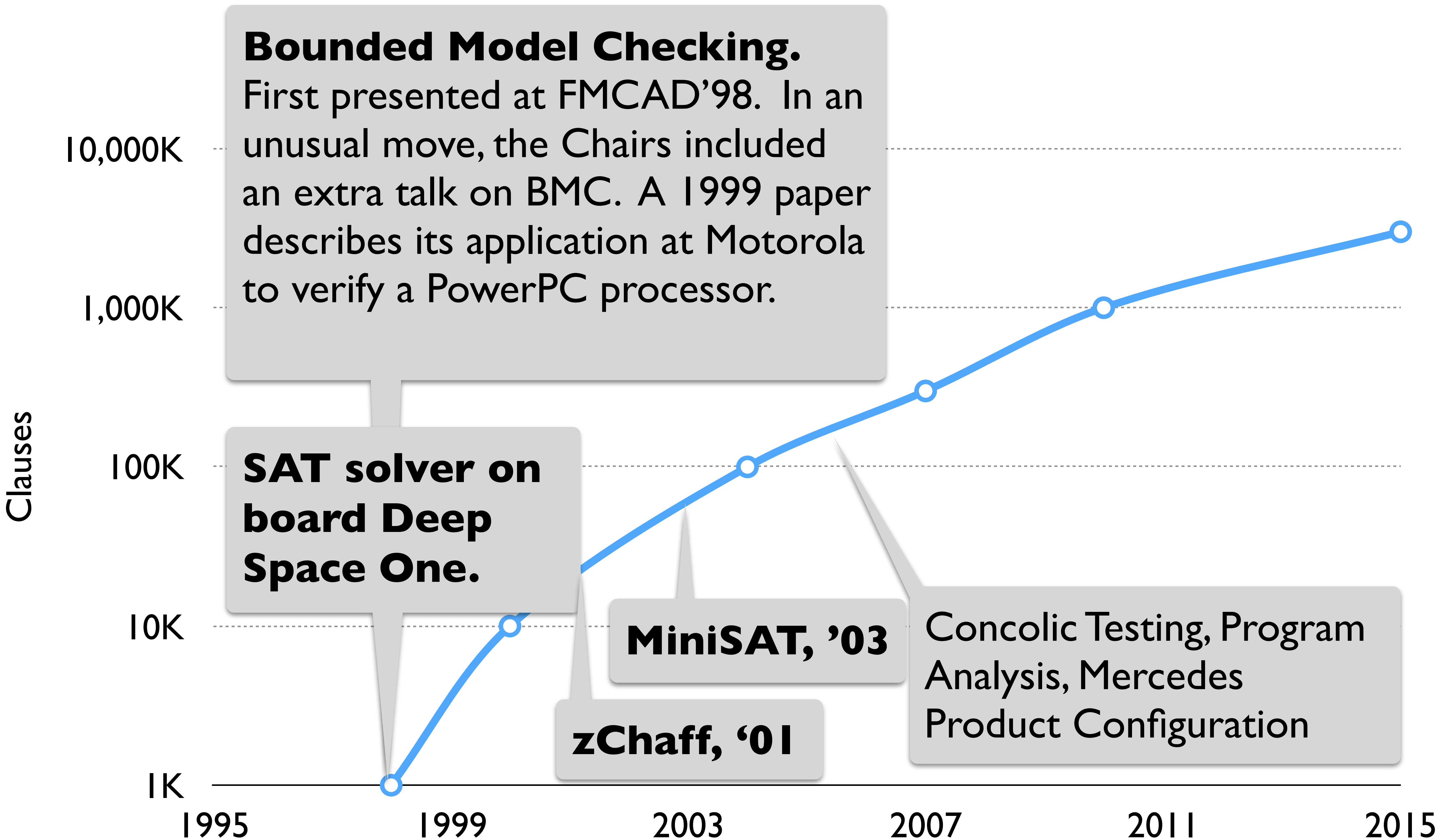
A brief history of SAT solving and applications



A brief history of SAT solving and applications

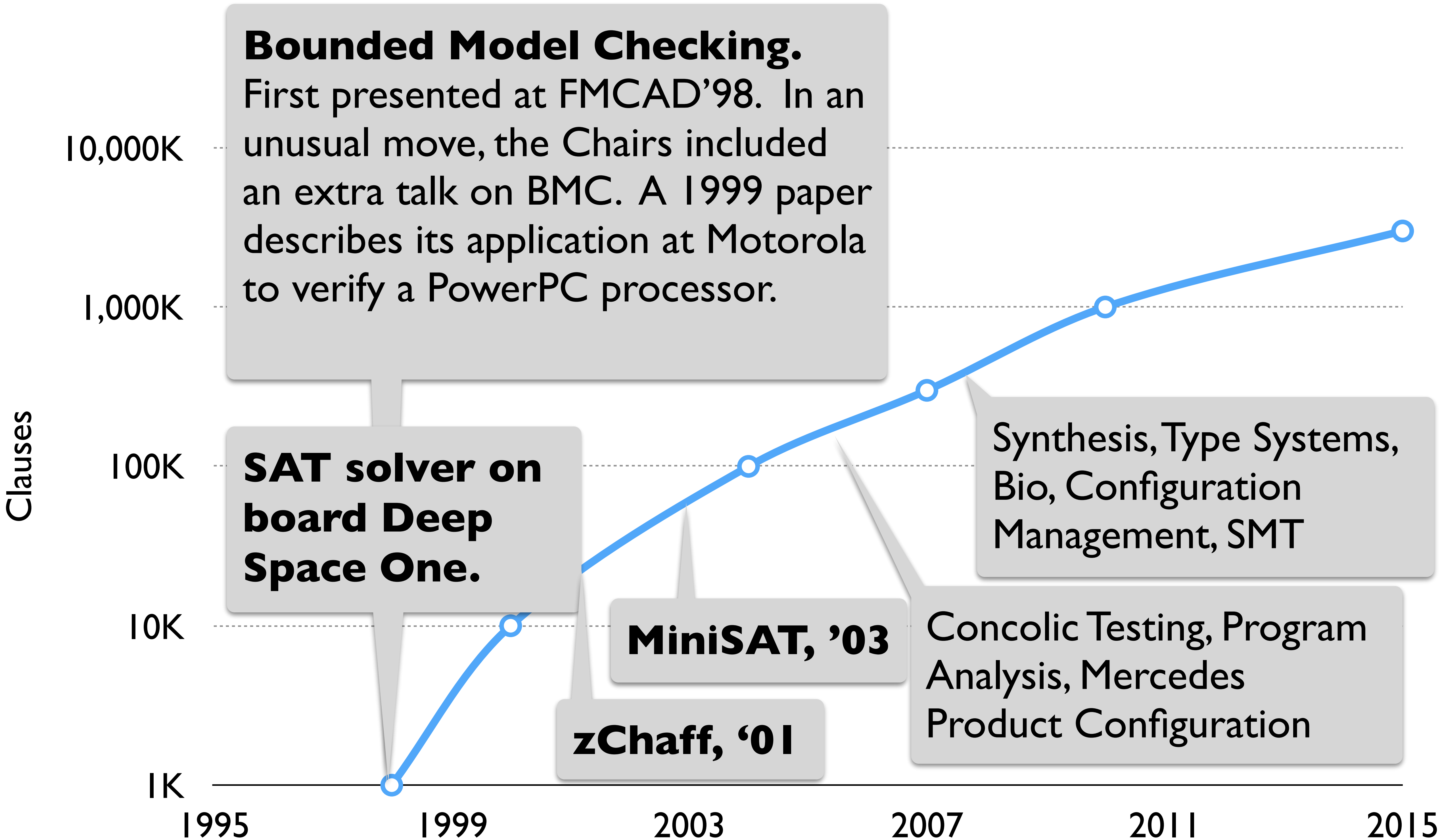


A brief history of SAT solving and applications



Based on a slide from Vijay Ganesh

A brief history of SAT solving and applications



Based on a slide from Vijay Ganesh

today

Bounded Model Checking (BMC) & Configuration Management

Bounded Model Checking (in general)

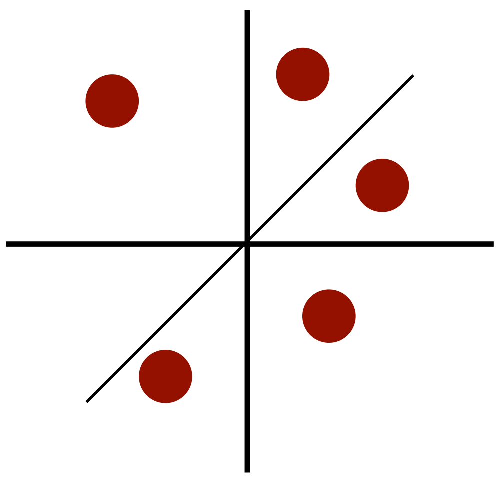
Given a system and a property, BMC checks if the property is satisfied by all executions of the system with ≤k steps, on all inputs of size ≤n.

Bounded Model Checking (in general)

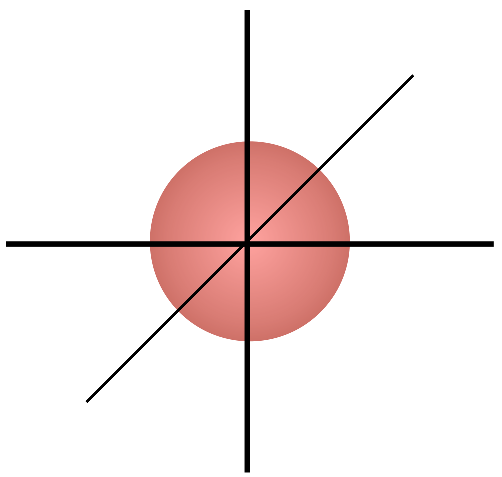
Given a system and a property, BMC checks if the property is satisfied by all executions of the system with $\leq k$ steps, on all inputs of size $\leq n$.

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

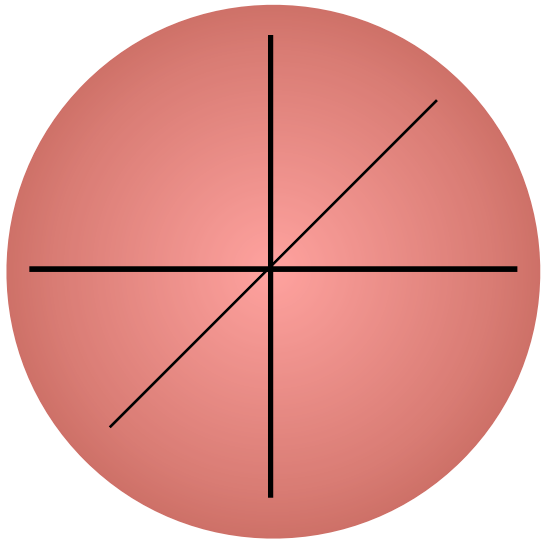
Bounded Model Checking (in general)



Testing: checks a few executions of arbitrary size

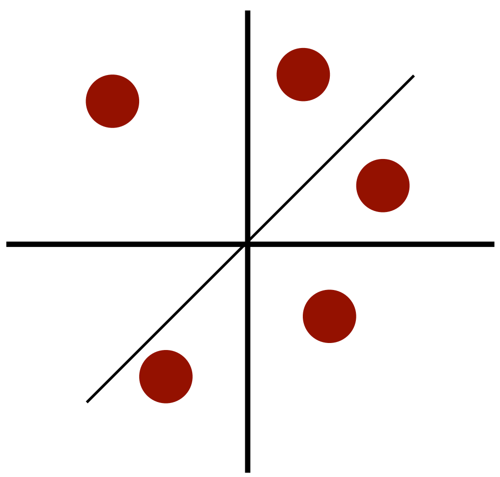


BMC: checks all executions of size ≤k

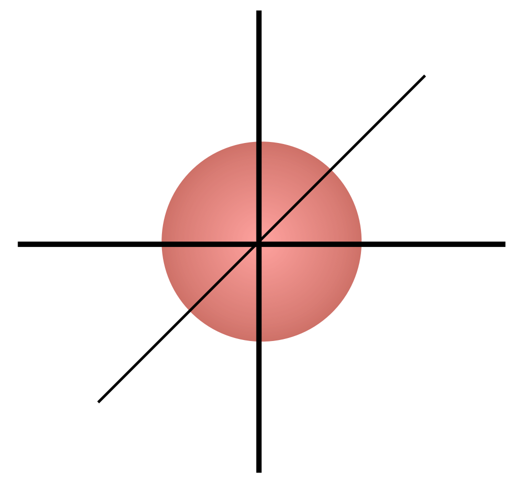


Verification: checks all executions of every size

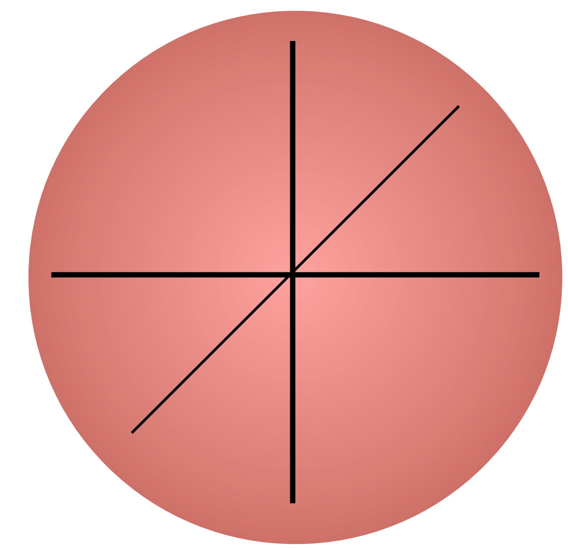
Bounded Model Checking (in general)



Testing: checks a few executions of arbitrary size



BMC: checks all executions of size ≤k



Verification: checks all executions of every size

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

```
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?

BMC by example

```
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.

BMC by example

```
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;
}
```

A desired safety property: the value of the days variable decreases in every loop iteration.

BMC step 1 of 4: finitize loops & inline calls

```
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;
}
```

BMC step 1 of 4: finitize loops & inline calls

```
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.

BMC step 1 of 4: finitize loops & inline calls

```
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 ≤k.

BMC step 1 of 4: finitize loops & inline calls

```
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 ≤k.
- If a CEX violates an unwinding assertion, the program has no buggy behavior of length ≤k, but it may have a longer one.

BMC step 1 of 4: finitize loops & inline calls

```
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!

BMC step 1 of 4: finitize loops & inline calls

```
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;
}
```

Assume call to isLeapYear is inlined (replaced with the procedure body). We'll keep it for readability.

BMC step 2 of 4: eliminate side effects

```
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

```
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

```
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 φ nodes.

BMC step 2 of 4: eliminate side effects

```
int days_0;
int year_0 = 1980;
if (days_0 > 365) {
  int oldDays_0 = days_0;
  if (isLeapYear(year_0)) {
    if (days_0 > 366) {
      days_1 = days_0 - 366;
      year_1 = year_0 + 1;
    }
  } else {
    days_3 = days_0 - 365;
    year_3 = year_0 + 1;
  }
  assert days_4 < oldDays_0;
  assert days_4 <= 365;
}
return year_5;
```

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 φ nodes.

BMC step 2 of 4: eliminate side effects

```
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;
```

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 φ nodes.

BMC step 2 of 4: eliminate side effects

```
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;
```

```
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

```
int days_0;
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;
year_5 = φ(g_0, year_4, year_0);
return year_5;
```

BMC step 3 of 4: convert into equations

$$
\begin{aligned}
&\mathtt{year}_0 = 1980 \land \\
&g_0 = (\mathtt{days}_0 > 365) \land \\
&\mathtt{oldDays}_0 = \mathtt{days}_0 \land \\
&g_1 = \mathit{isLeapYear}(\mathtt{year}_0) \land \\
&g_2 = \mathtt{days}_0 > 366 \land \\
&\mathtt{days}_1 = \mathtt{days}_0 - 366 \land \\
&\mathtt{year}_1 = \mathtt{year}_0 + 1 \land \\
&\mathtt{days}_2 = \mathtt{ite}(g_1 \land g_2, \mathtt{days}_1, \mathtt{days}_0) \land \\
&\mathtt{year}_2 = \mathtt{ite}(g_1 \land g_2, \mathtt{year}_1, \mathtt{year}_0) \land \\
&\mathtt{days}_3 = \mathtt{days}_0 - 365 \land \\
&\mathtt{year}_3 = \mathtt{year}_0 + 1 \land \\
&\mathtt{days}_4 = \mathtt{ite}(g_1, \mathtt{days}_2, \mathtt{days}_3) \land \\
&\mathtt{year}_4 = \mathtt{ite}(g_1, \mathtt{year}_2, \mathtt{year}_3) \land \\
&\mathtt{year}_5 = \mathtt{ite}(g_0, \mathtt{year}_4, \mathtt{year}_0) \land \\
&(\neg(\mathtt{days}_4 < \mathtt{oldDays}_0) \lor \\
&\ \neg(\mathtt{days}_4 <= 365))
\end{aligned}
$$

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

```
year₁ = year₀ + 1
```

BMC step 4 of 4: convert into CNF

```
year₁ = year₀ + 1
```

$year_0$ = 000 … 000

31 30 29 … 2 1 0

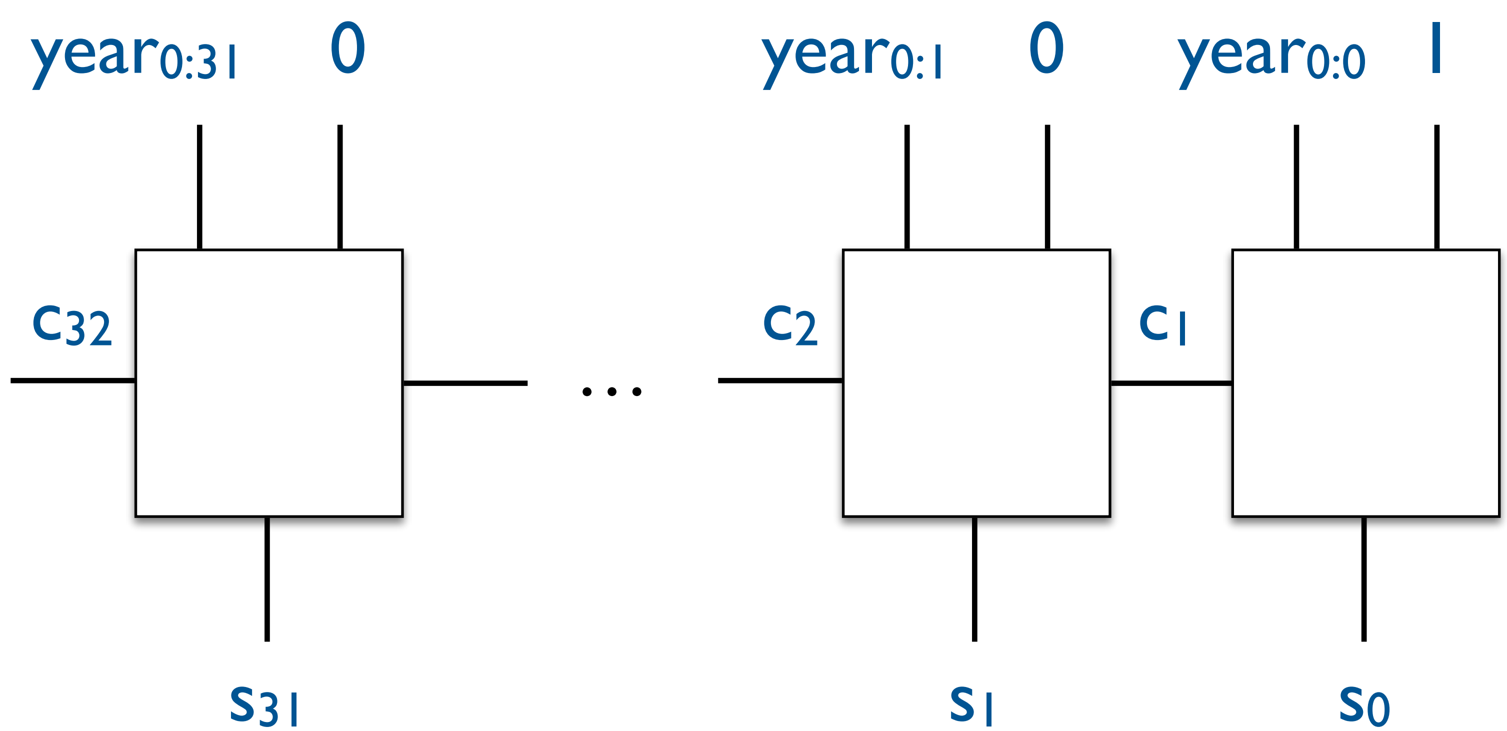
Represent numbers as arrays of bits, and create one propositional variable per bit for each number.

BMC step 4 of 4: convert into CNF

$year_1 = year_0 + 1$

$year_0 = 000 \dots 000$ (bits $31\ 30\ 29 \dots 2\ 1\ 0$)

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

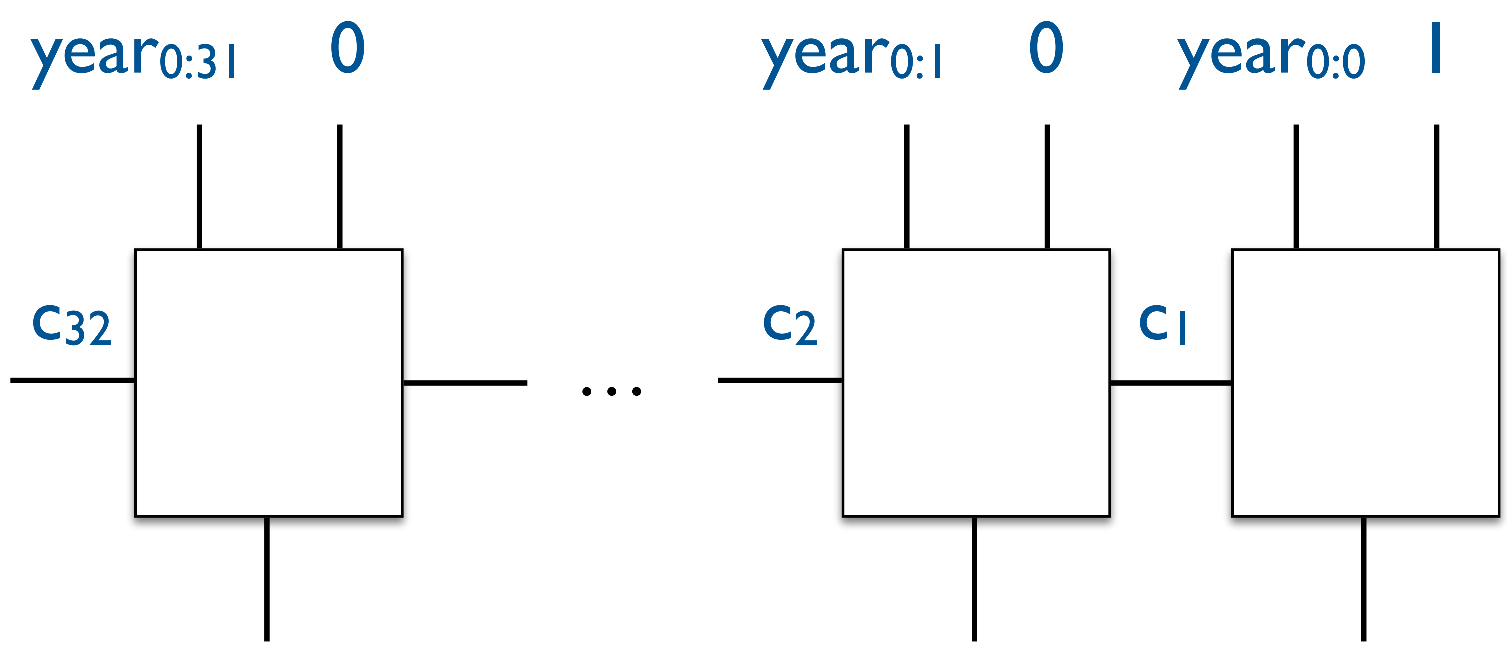


BMC step 4 of 4: convert into CNF

```
year1 = year0 + 1
```

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

$year_0 = \underset{31}{0}\underset{30}{0}\underset{29}{0} \ldots \underset{2}{0}\underset{1}{0}\underset{0}{0}$



$year_{1:31} \iff s_{31} \wedge \ldots \wedge year_{1:1} \iff s_1 \wedge s_0 \iff year_{1:0}$

Introduce new clauses to constrain bits in $year_1$ to match bits in the sum.

BMC counterexample for k=1

```
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

today

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.

SAT

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

Configuration Management

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

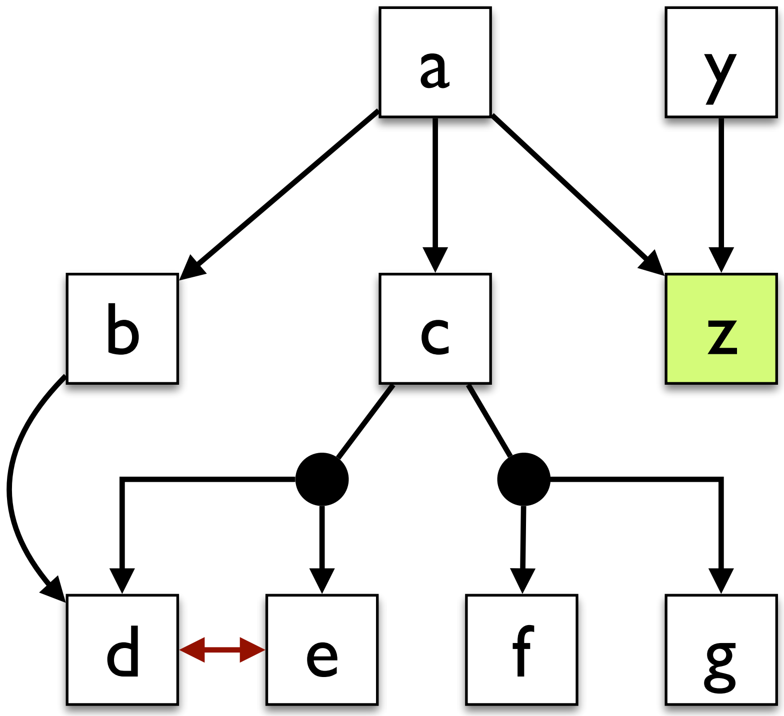
- Decide if a new component can be added to the configuration.
- Add the component while optimizing some linear function.
- If the component cannot be added, find a way to add it by removing as few conflicting components from the current configuration as possible.

SAT

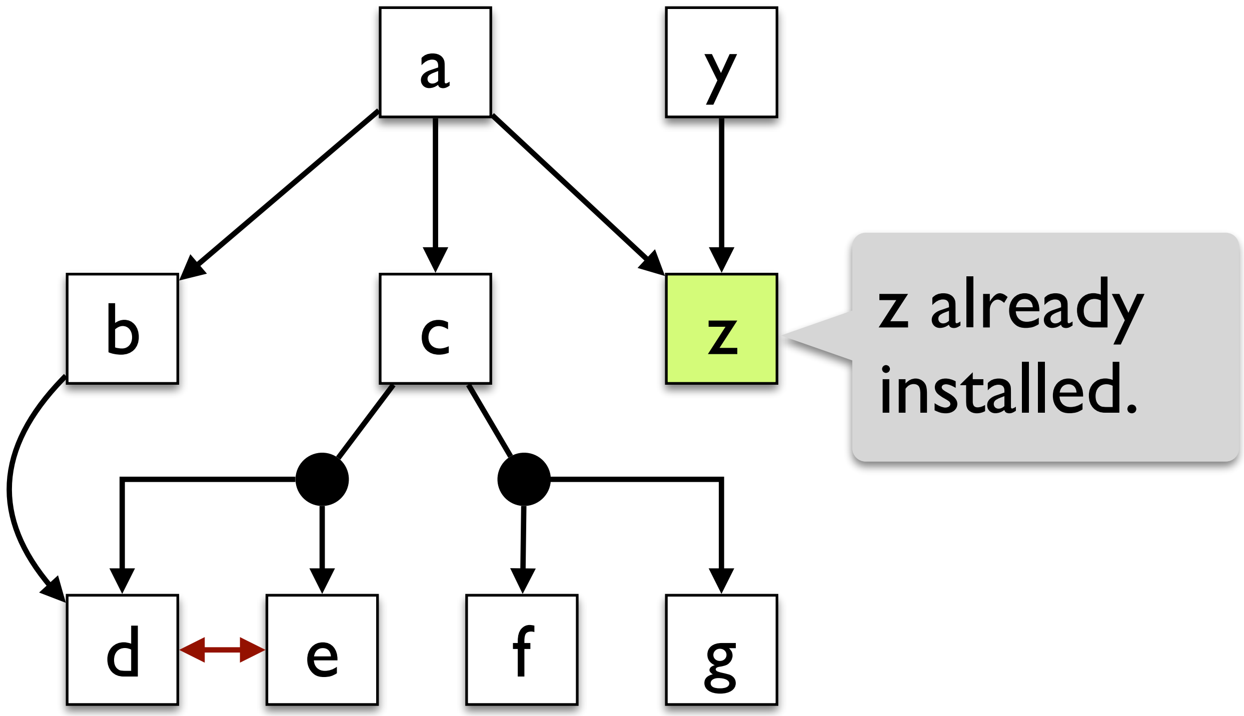
Pseudo-Boolean Constraints

Partial (Weighted) MaxSAT

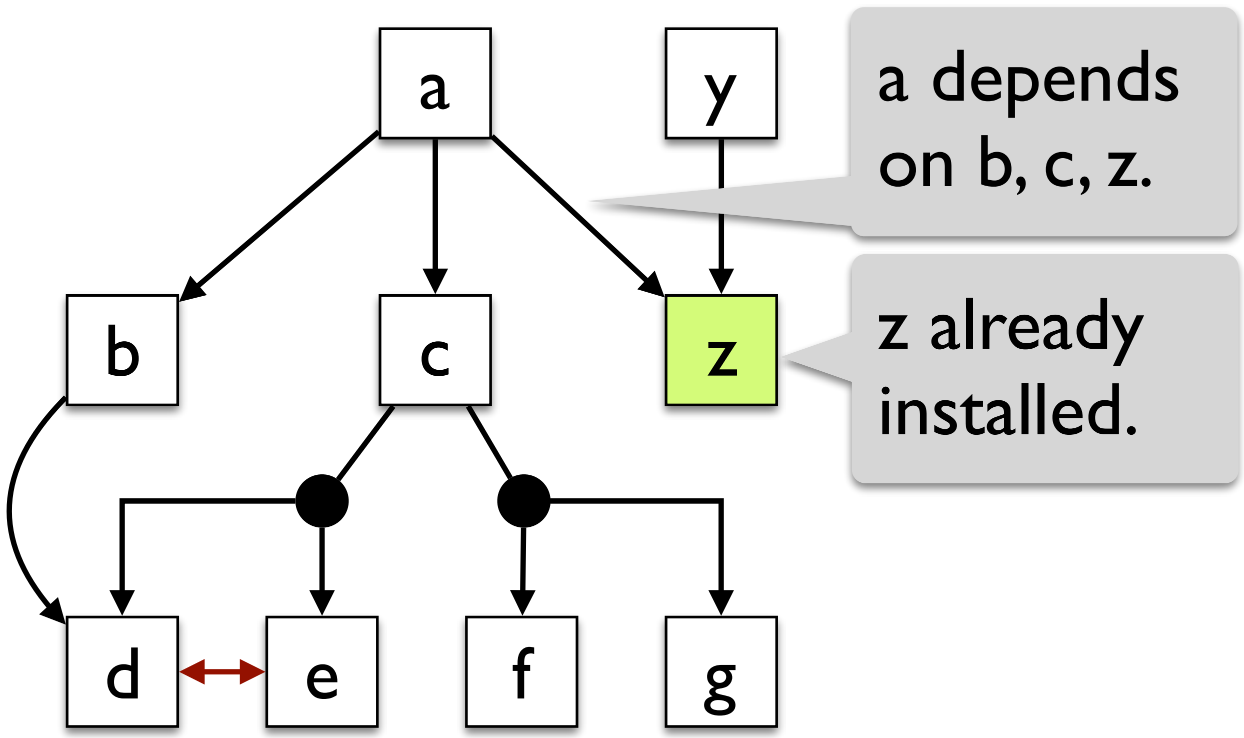
Deciding if a component can be installed



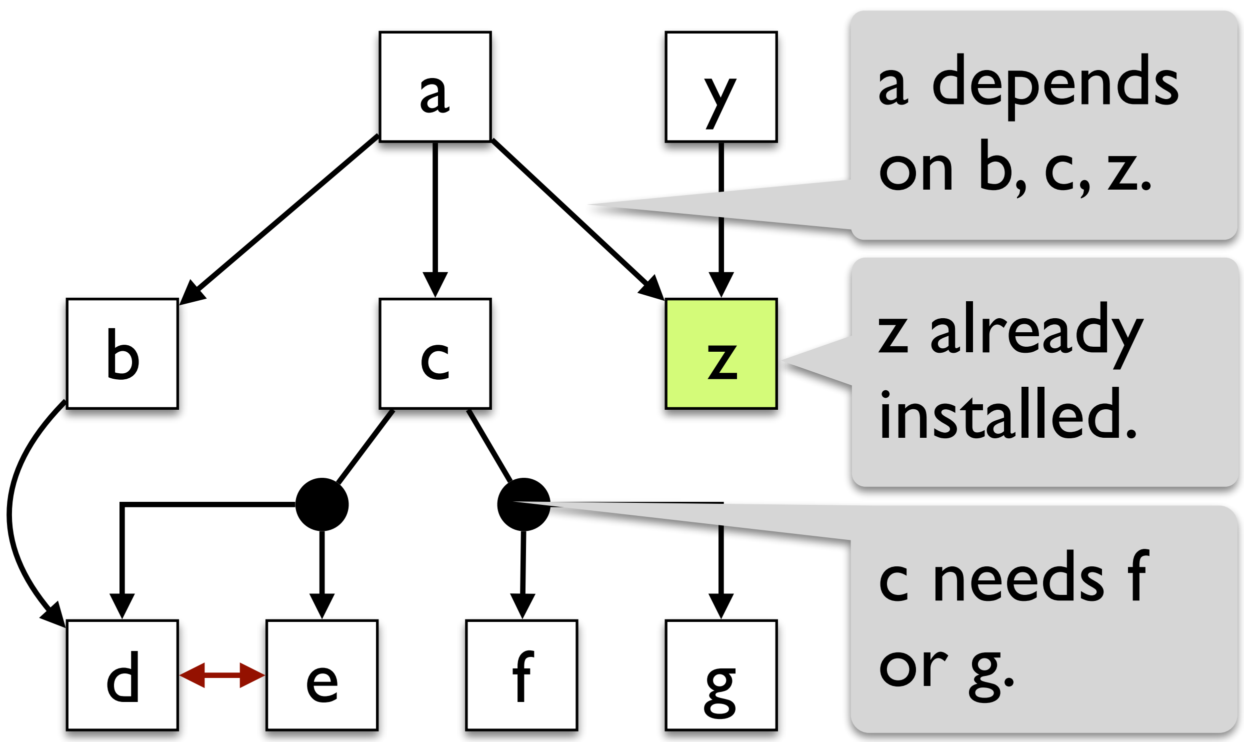
Deciding if a component can be installed



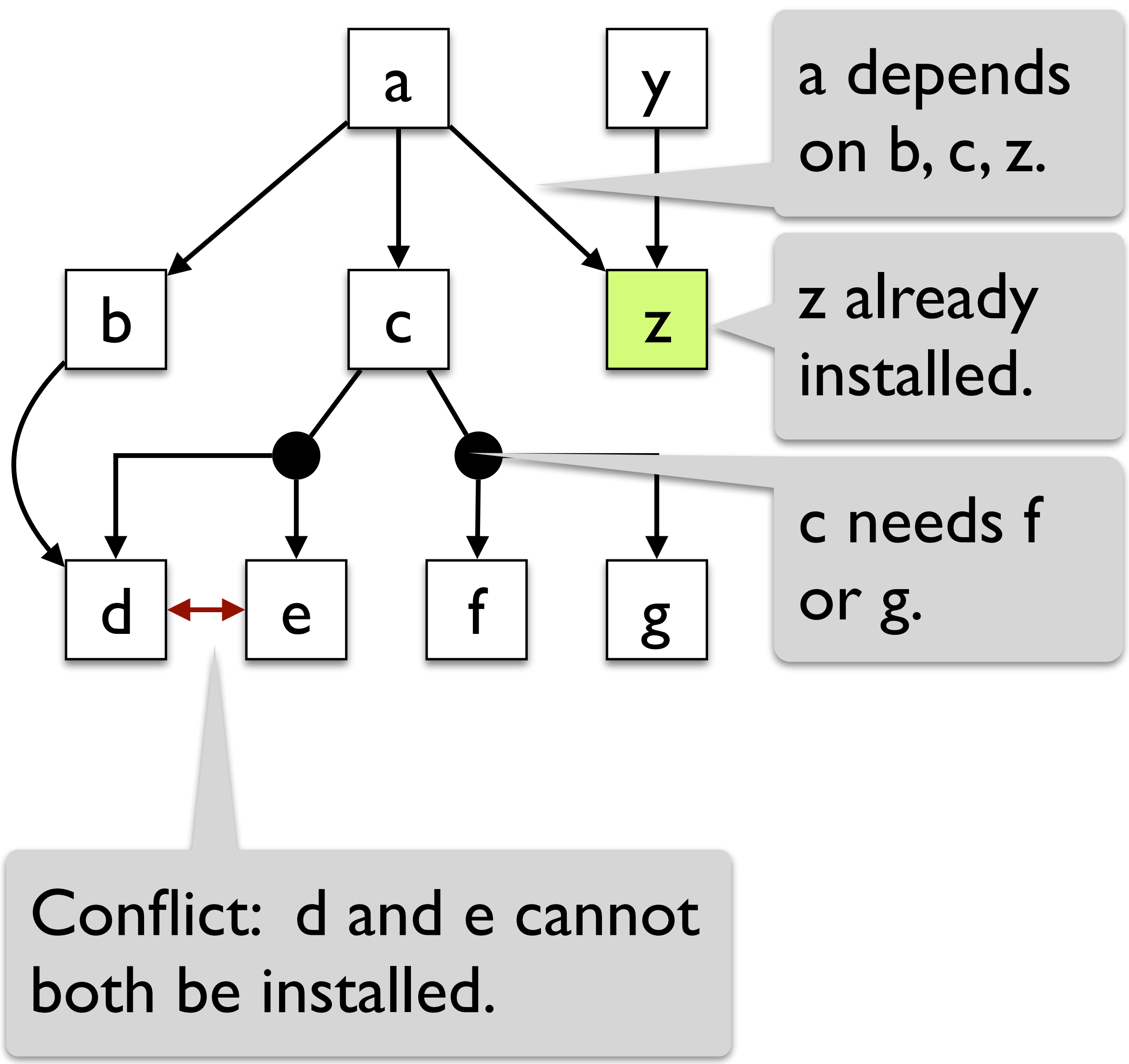
Deciding if a component can be installed



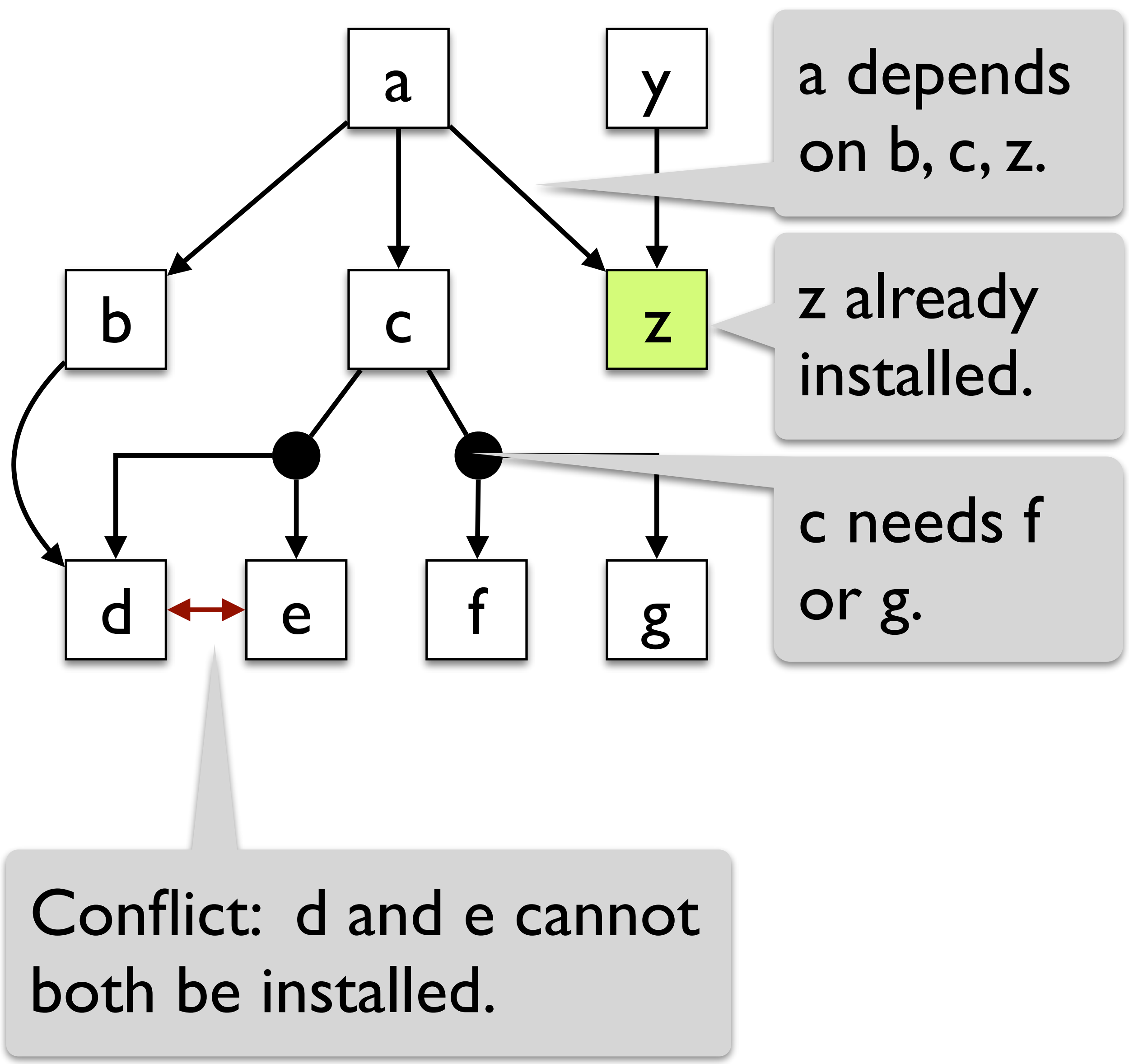
Deciding if a component can be installed



Deciding if a component can be installed

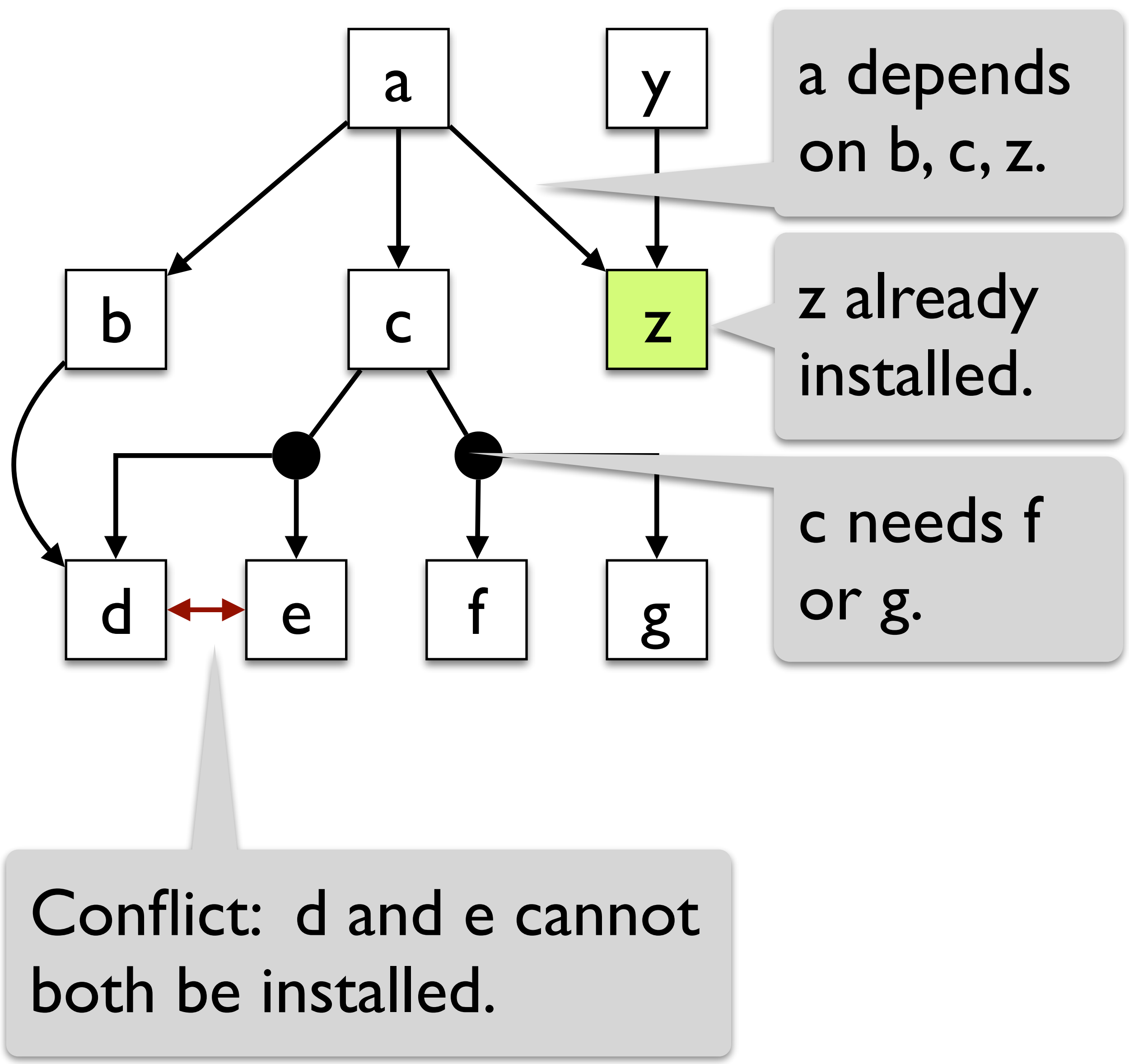


Deciding if a component can be installed



To install a, CNF constraints are:

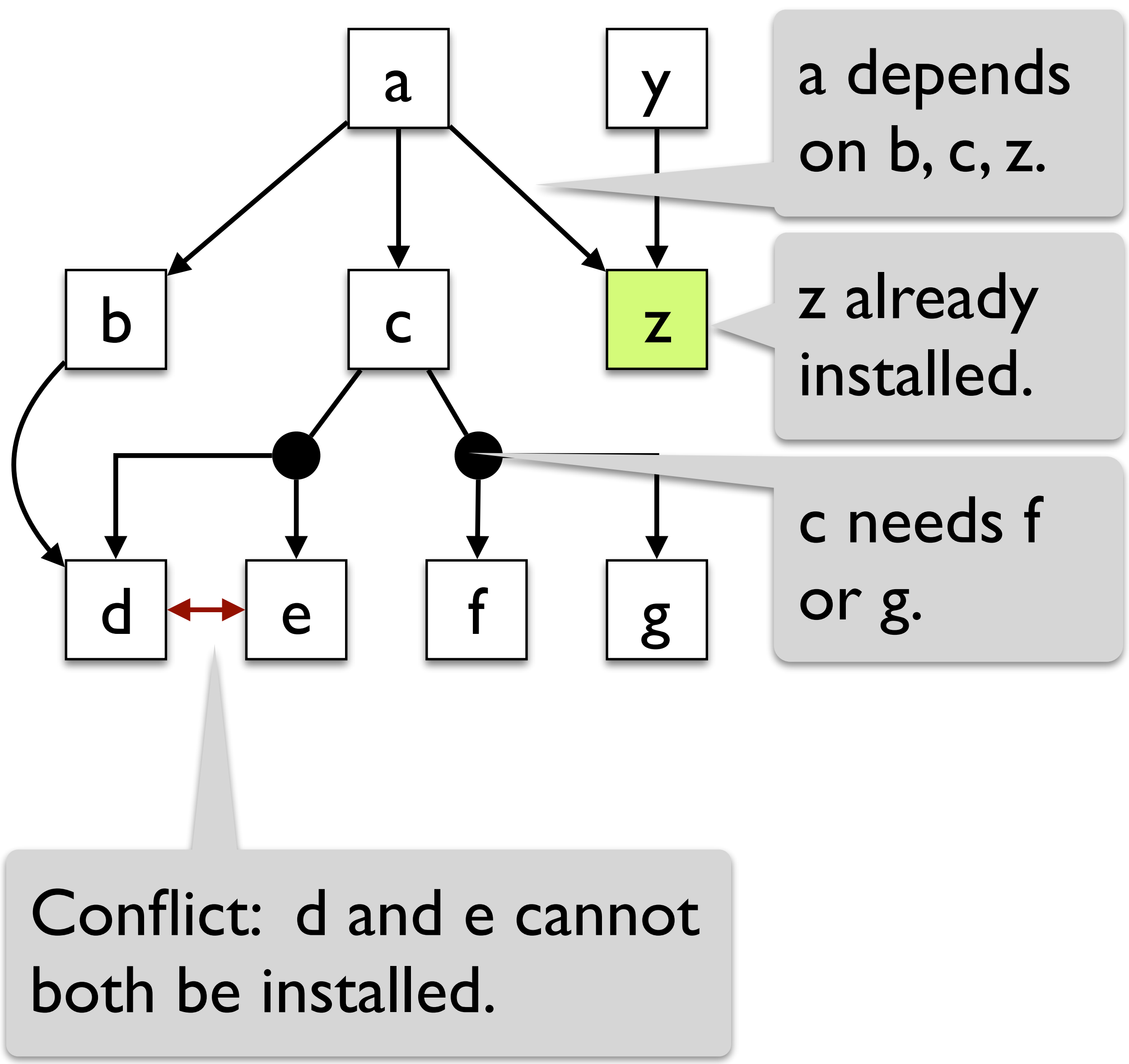
Deciding if a component can be installed



To install a, CNF constraints are:

$(\neg a \vee b) \wedge (\neg a \vee c) \wedge (\neg a \vee z) \wedge$

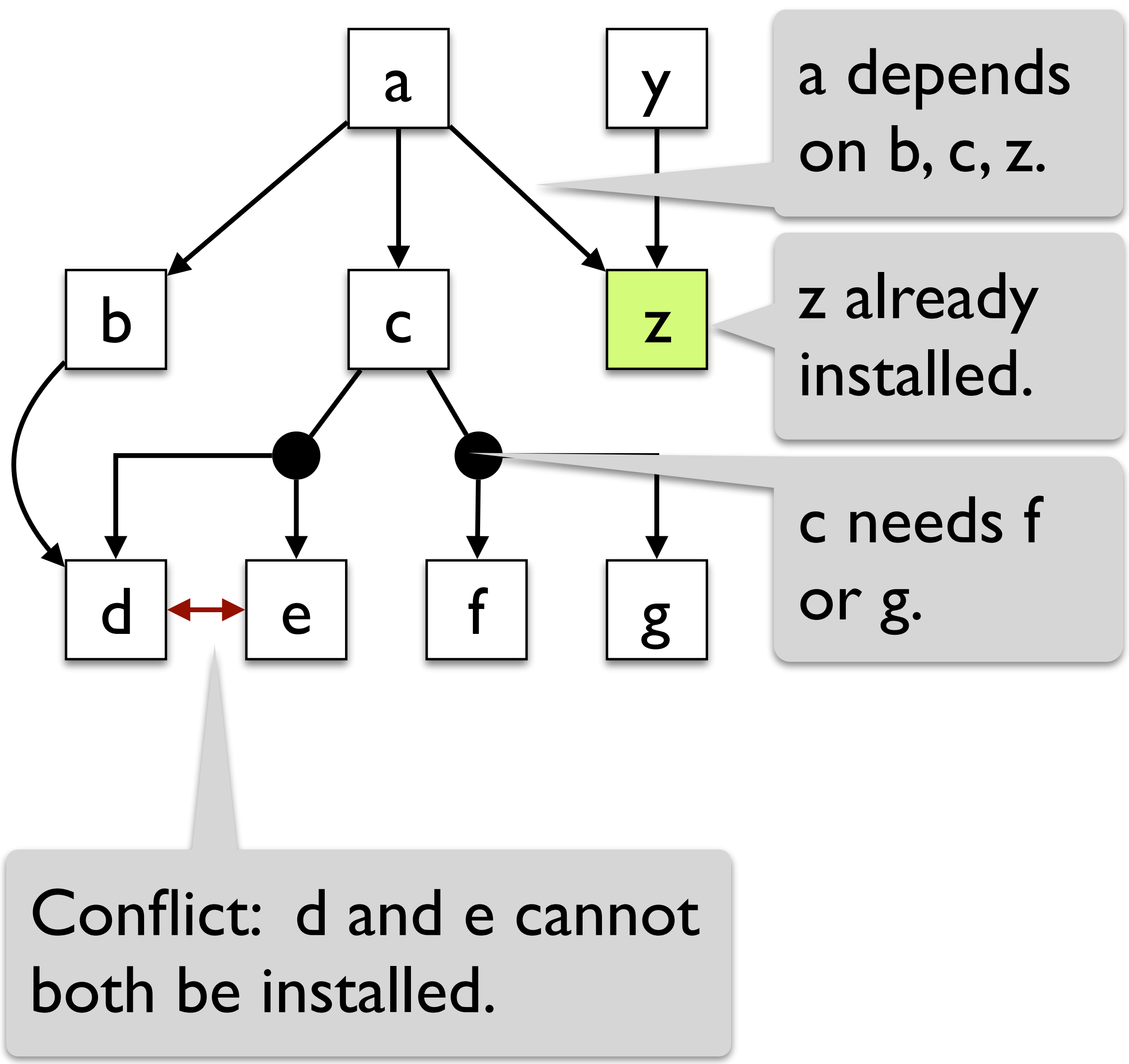
Deciding if a component can be installed



To install a, CNF constraints are:

$(\neg a \vee b) \wedge (\neg a \vee c) \wedge (\neg a \vee z) \wedge$
$(\neg b \vee d) \wedge$

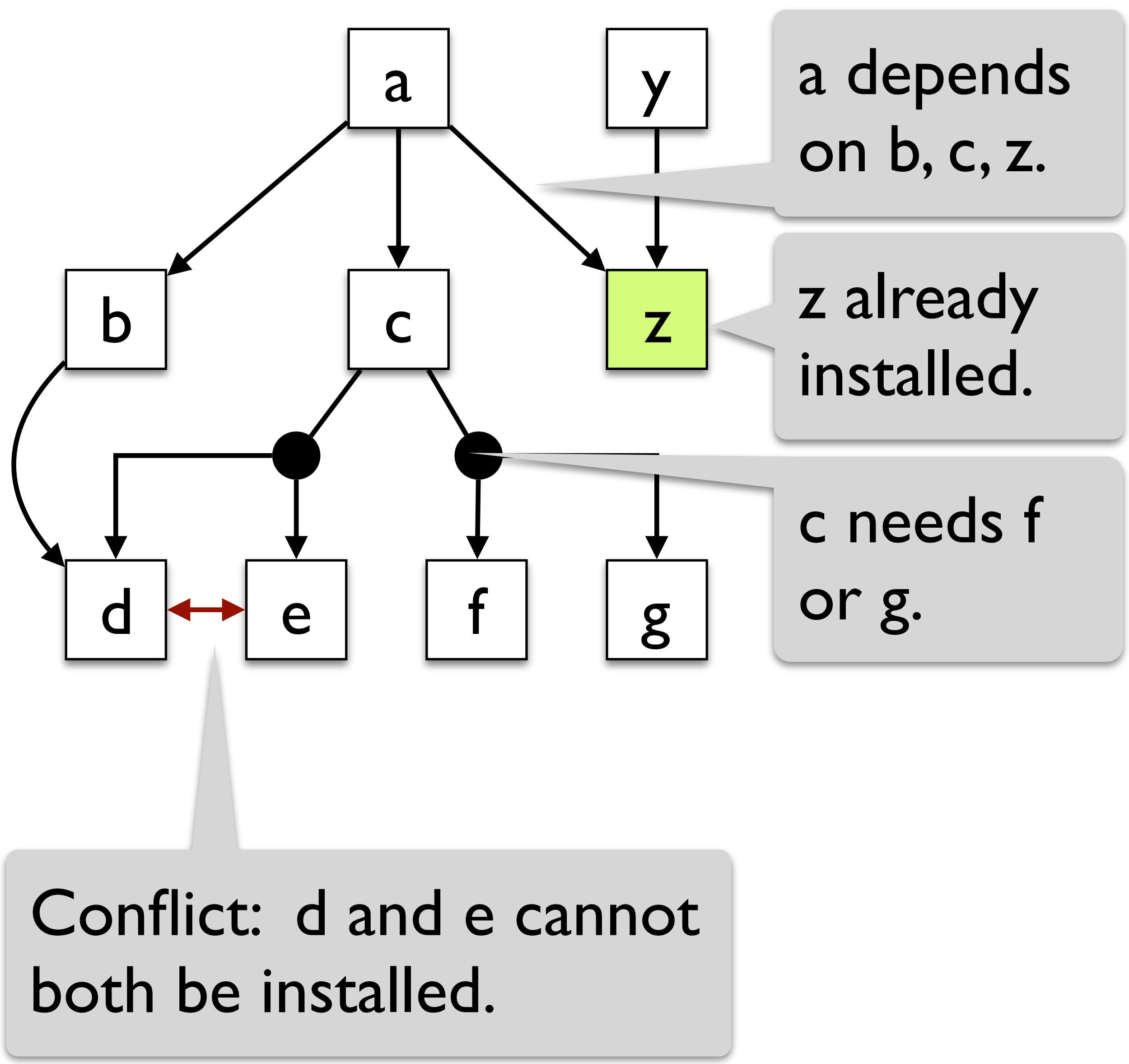
Deciding if a component can be installed



To install a, CNF constraints are:

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

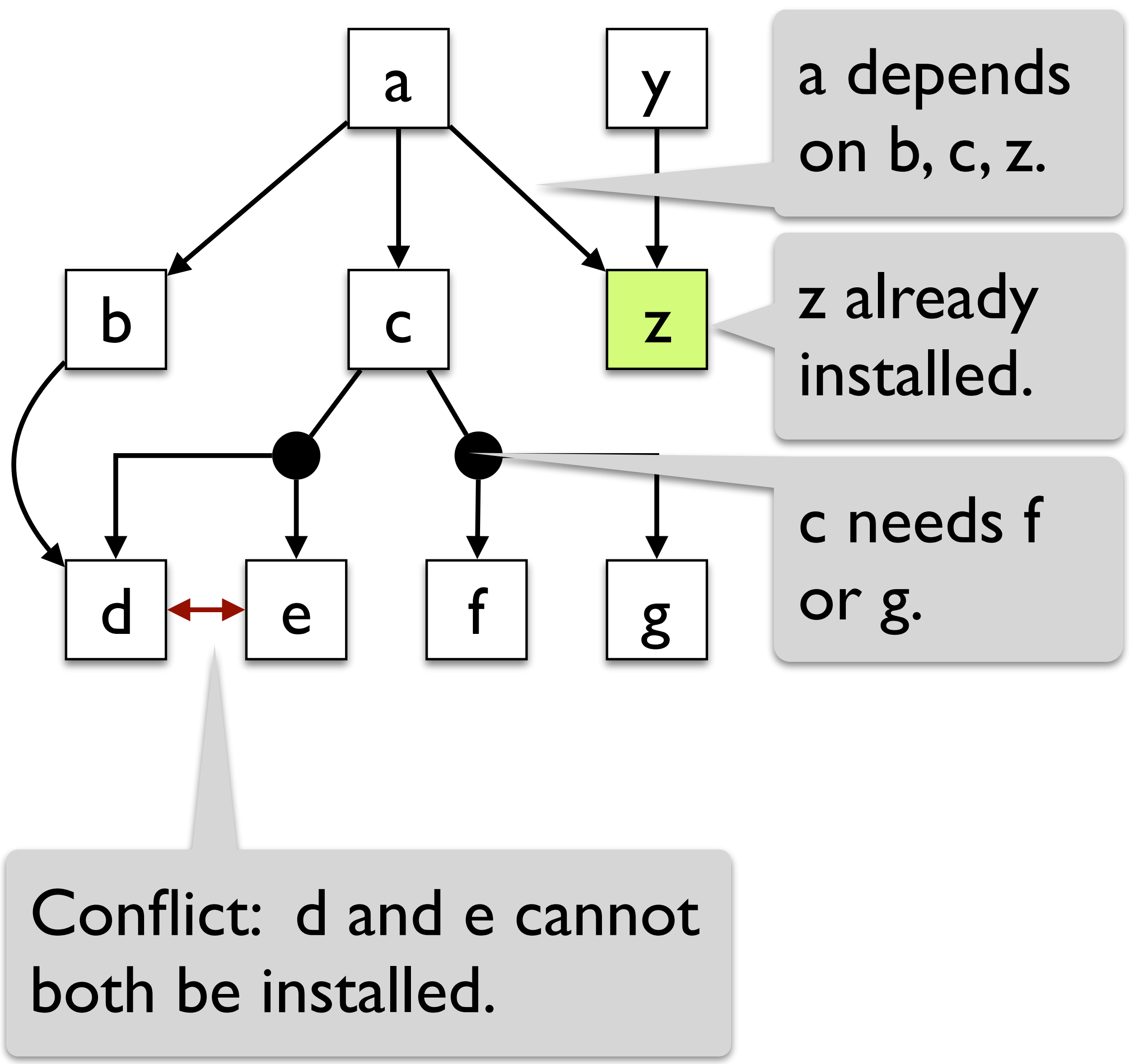
Deciding if a component can be installed



To install a, CNF constraints are:

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

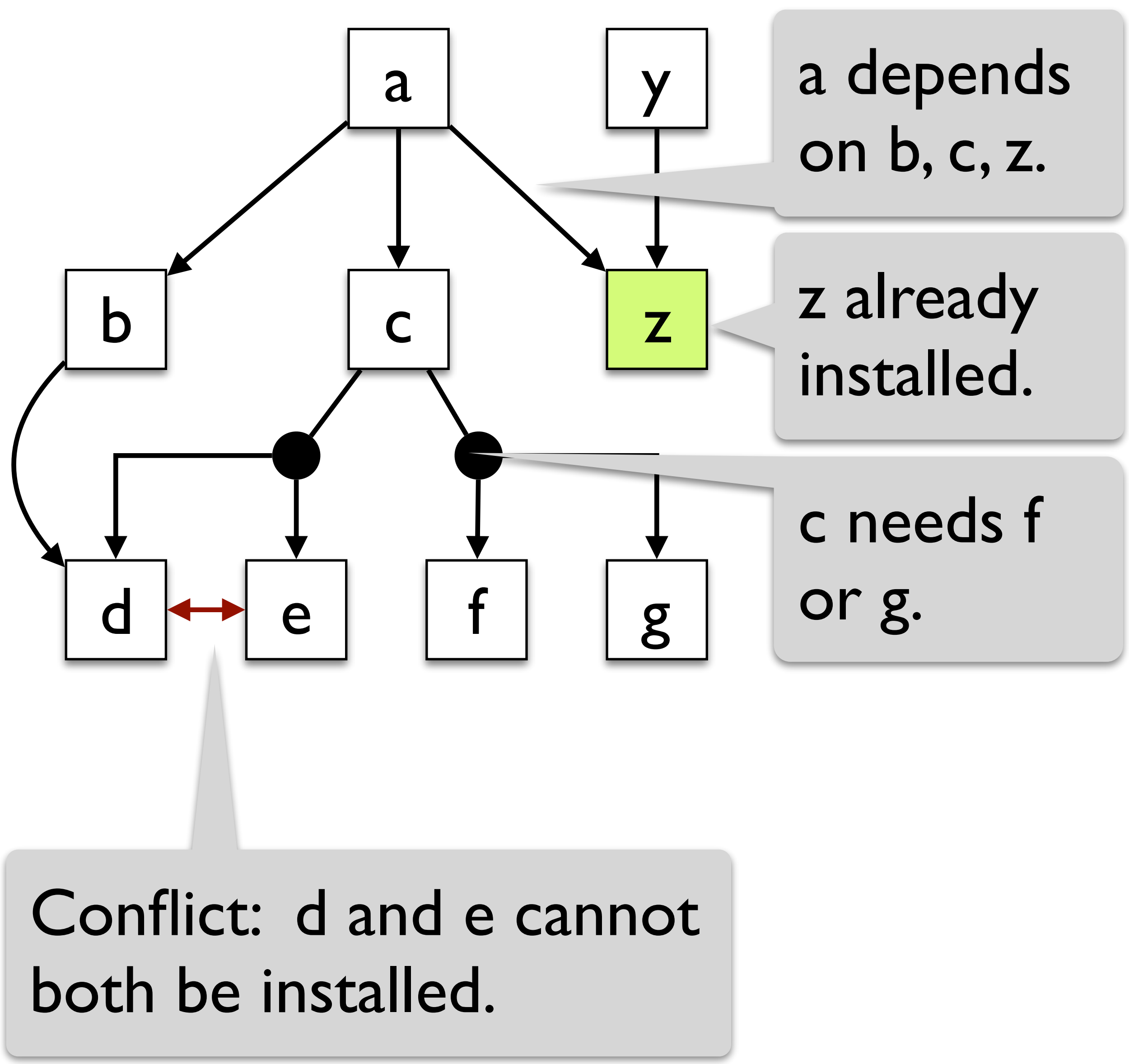
Deciding if a component can be installed



To install a, CNF constraints are:

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

Deciding if a component can be installed



To install a, CNF constraints are:

$(\neg a \vee b) \wedge (\neg a \vee c) \wedge (\neg a \vee z) \wedge$

$(\neg b \vee d) \wedge$

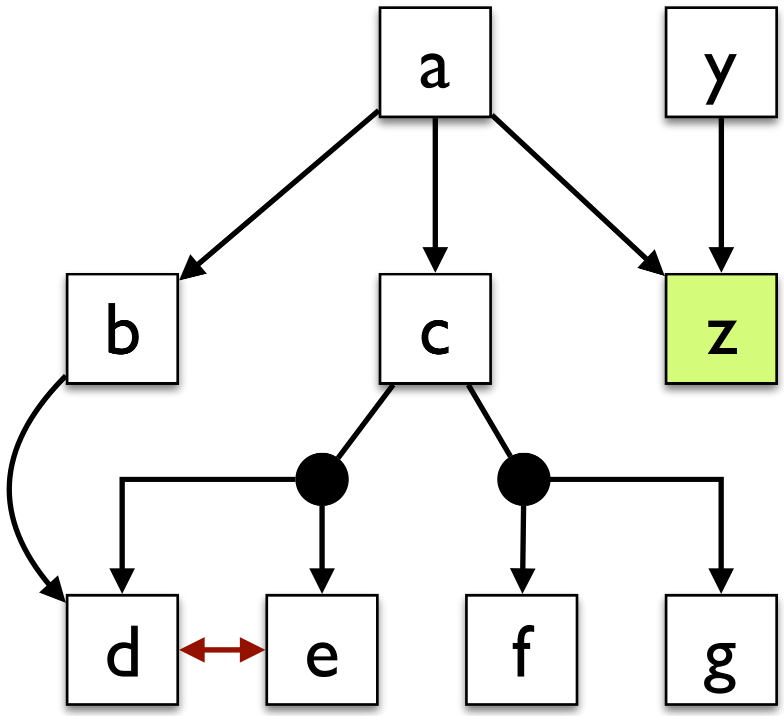
$(\neg c \vee d \vee e) \wedge (\neg c \vee f \vee g) \wedge$

$(\neg d \vee \neg e) \wedge$

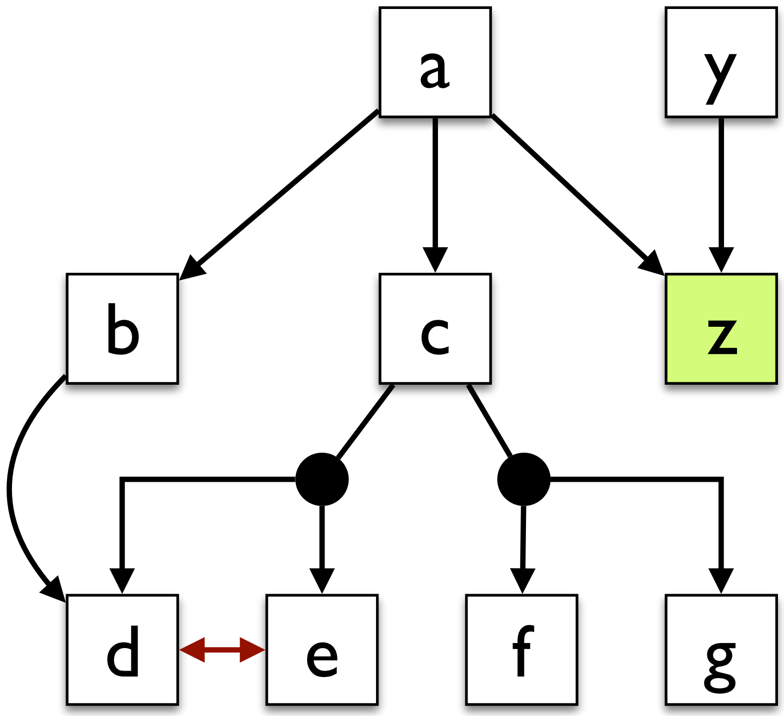
$(\neg y \vee z) \wedge$

$a \wedge z$

Optimal installation



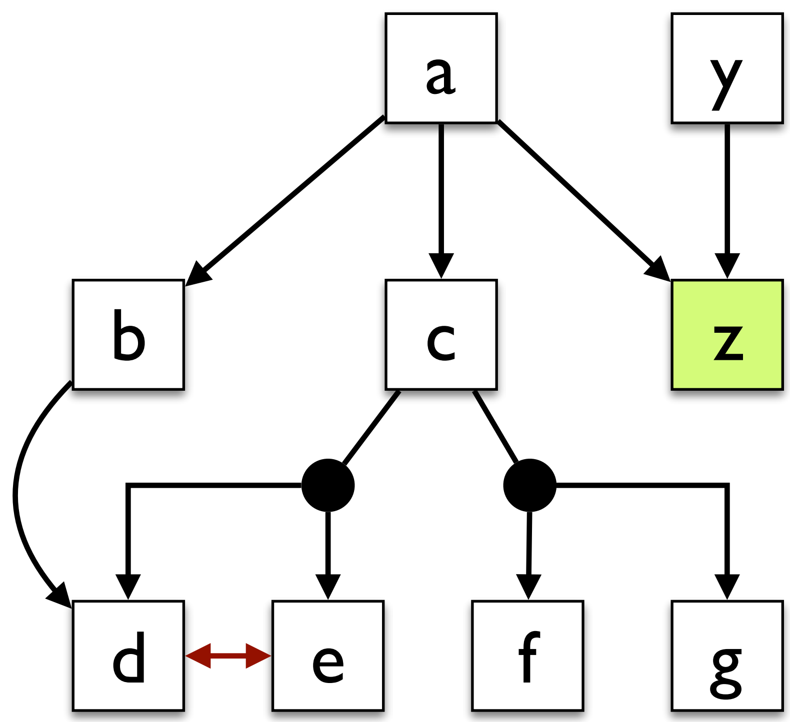
Optimal installation



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.

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:

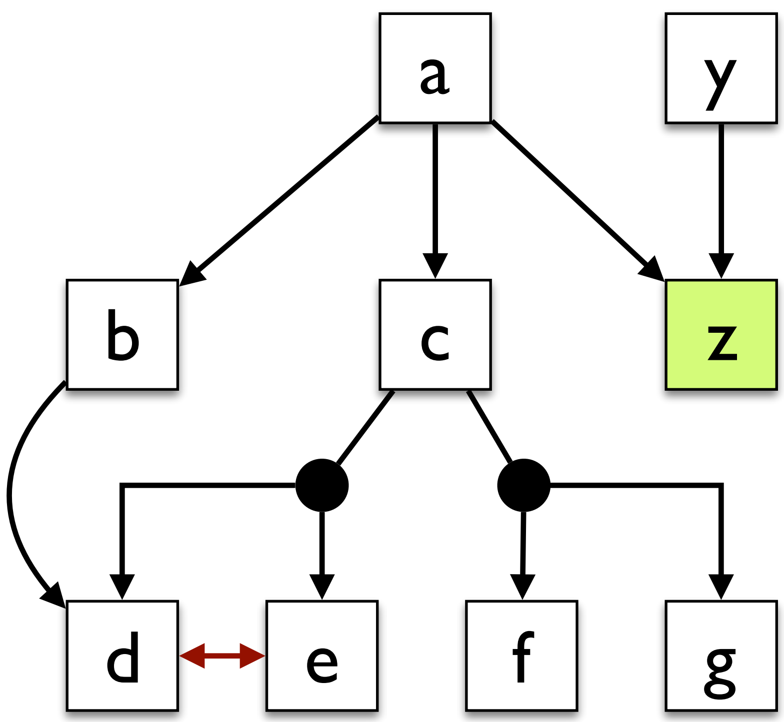
min $c_1x_1 + \ldots + c_nx_n$

$a_{11}x_1 + \ldots + a_{1n}x_n \geq b_1$

$\ldots$

$a_{k1}x_1 + \ldots + a_{kn}x_n \geq b_k$

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:

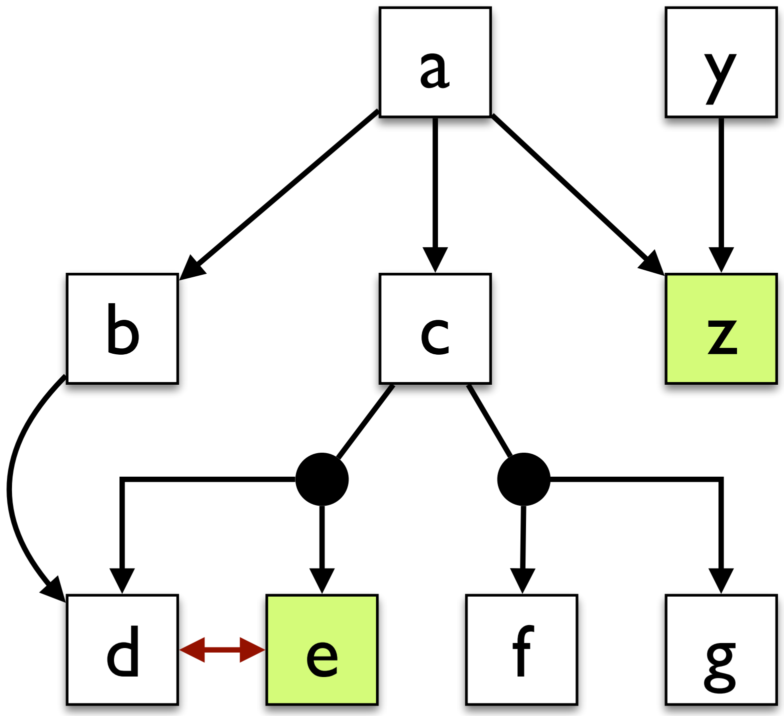
min $a + b + c + d + e + 5f + 2g + y + 0z$

$(-a + b \geq 0) \wedge (-a + c \geq 0) \wedge (-a + z \geq 0) \wedge$
$(-b + d \geq 0) \wedge$
$(-c + d + e \geq 0) \wedge (-c + f + g \geq 0) \wedge$
$(-d + -e \geq -1) \wedge$
$(-y + z \geq 0) \wedge$
$(a \geq 1) \wedge (z \geq 1)$

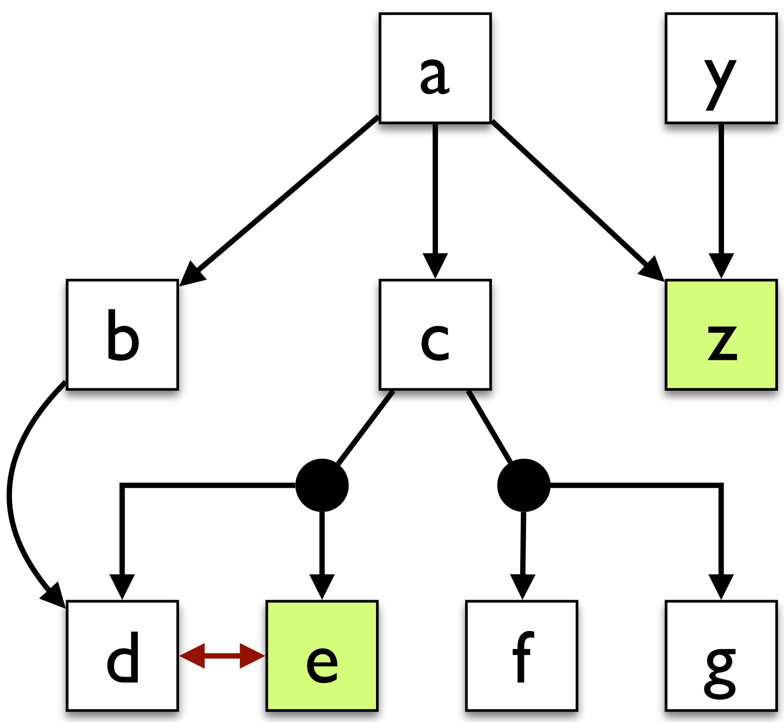
min $c_1x_1 + \ldots + c_nx_n$

$a_{11}x_1 + \ldots + a_{1n}x_n \geq b_1$
$\ldots$
$a_{k1}x_1 + \ldots + a_{kn}x_n \geq b_k$

Installation in the presence of conflicts

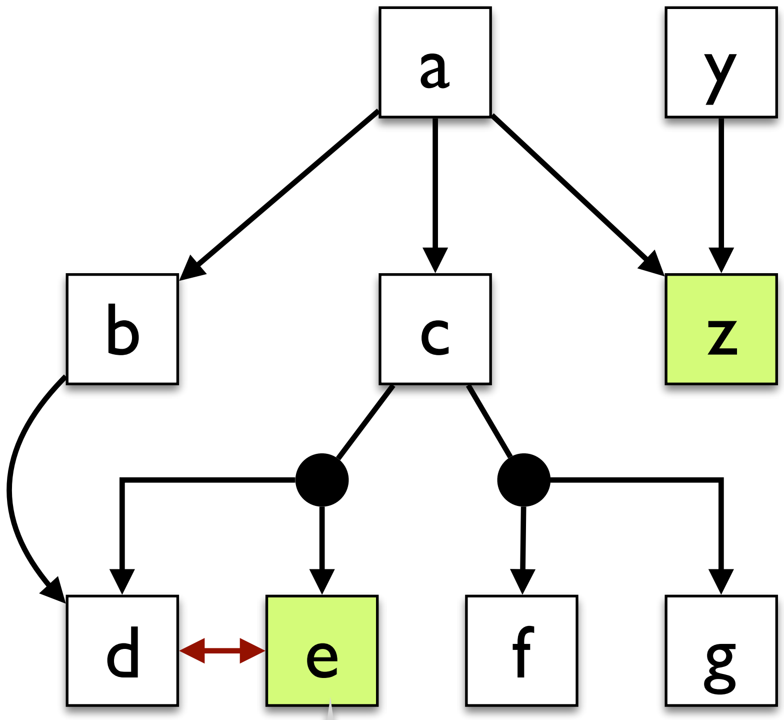


Installation in the presence of conflicts



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

Installation in the presence of conflicts



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

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

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)