CBMC: Bounded Model Checking for ANSI-C

CBMC

Version 1.0, 2010

Outline

Preliminaries

BMC Basics

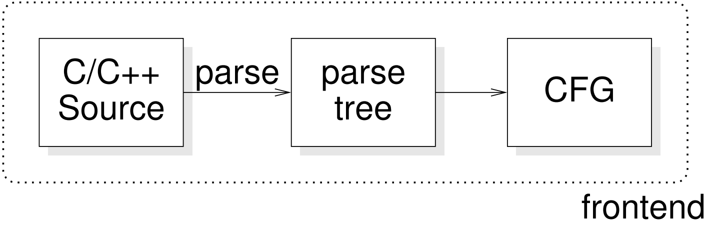
Completeness

Solving the Decision Problem

Preliminaries

- We aim at the analysis of programs given in a commodity programming language such as C, C++, or Java

- As the first step, we transform the program into a *control flow graph* (CFG)



Example: SHS

```
if ( (0 <= t) && (t <= 79) )
  switch ( t / 20 )
  {
  case 0:
     TEMP2 = ( (B AND C) OR (~B AND D) );
     TEMP3 = ( K_1 );
     break;

  case 1:
     TEMP2 = ( (B XOR C XOR D) );
     TEMP3 = ( K_2 );
     break;

  case 2:
     TEMP2 = ( (B AND C) OR (B AND D) OR (C AND D) );
     TEMP3 = ( K_3 );
     break;

  case 3:
     TEMP2 = ( B XOR C XOR D );
     TEMP3 = ( K_4 );
     break;

  default:
     assert(0);
  }
```

Example: SHS

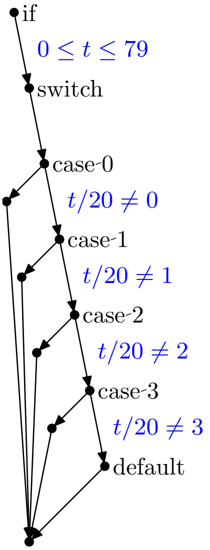
```
if ( (0 <= t) && (t <= 79) )
  switch ( t / 20 )
  {
  case 0:
     TEMP2 = ( (B AND C) OR (~B AND D) );
     TEMP3 = ( K_1 );
     break;

  case 1:
     TEMP2 = ( (B XOR C XOR D) );
     TEMP3 = ( K_2 );
     break;

  case 2:
     TEMP2 = ( (B AND C) OR (B AND D) OR (C AND D) );
     TEMP3 = ( K_3 );
     break;

  case 3:
     TEMP2 = ( B XOR C XOR D );
     TEMP3 = ( K_4 );
     break;

  default:
     assert(0);
  }
```

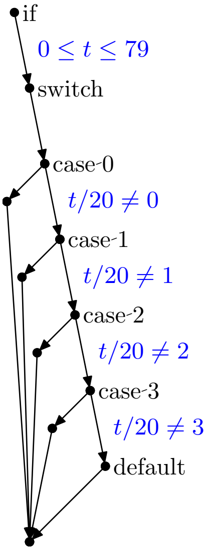


Bounded Program Analysis

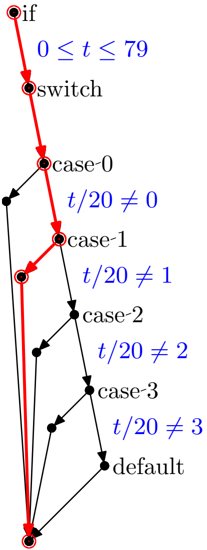
Goal: check properties of the form $\mathbf{AG}p$,
say assertions.

Idea: follow paths through the CFG to an assertion,
and build a formula that corresponds to the path

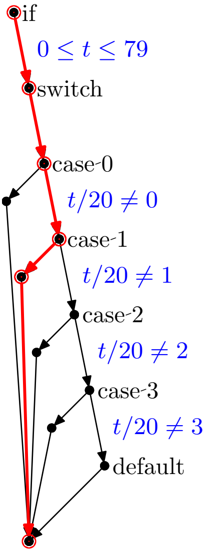
Example


if
$0 \leq t \leq 79$
switch
case-0
$t/20 \neq 0$
case-1
$t/20 \neq 1$
case-2
$t/20 \neq 2$
case-3
$t/20 \neq 3$
default

Example


if
$0 \leq t \leq 79$
switch
case-0
$t/20 \neq 0$
case-1
$t/20 \neq 1$
case-2
$t/20 \neq 2$
case-3
$t/20 \neq 3$
default

Example



$$
\begin{array}{ll}
 & 0 \le t \le 79 \\
\wedge & t/20 \neq 0 \\
\wedge & t/20 = 1 \\
\wedge & \mathit{TEMP2} = B \oplus C \oplus D \\
\wedge & \mathit{TEMP3} = K\_2
\end{array}
$$

Example

CBMC

We pass

$$
\begin{array}{ll}
 & 0 \leq t \leq 79 \\
\wedge & t/20 \neq 0 \\
\wedge & t/20 = 1 \\
\wedge & \mathit{TEMP2} = B \oplus C \oplus D \\
\wedge & \mathit{TEMP3} = K\_2
\end{array}
$$

to a decision procedure, and obtain a satisfying assignment, say:

$$
t \mapsto 21,\ B \mapsto 0,\ C \mapsto 0,\ D \mapsto 0,\ K\_2 \mapsto 10,\\
\mathit{TEMP2} \mapsto 0,\ \mathit{TEMP3} \mapsto 10
$$

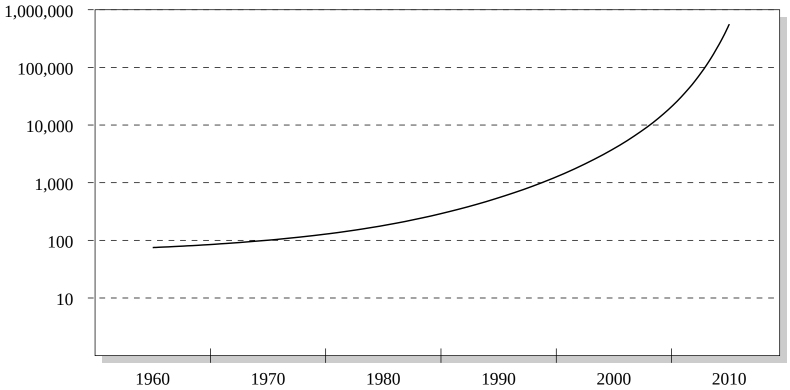
✔ It provides the values of any inputs on the path.

Which Decision Procedures?

- We need a decision procedure for an appropriate logic
  - Bit-vector logic (incl. non-linear arithmetic)
  - Arrays
  - Higher-level programming languages also feature lists, sets, and maps

- Examples
  - Z3 (Microsoft)
  - Yices (SRI)
  - Boolector

Enabling Technology: SAT



number of variables of a typical, practical SAT instance
that can be solved by the best solvers in that decade

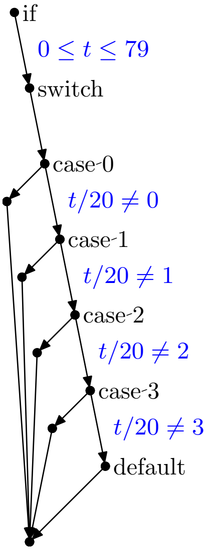
Enabling Technology: SAT

CBMC

- propositional SAT solvers have made enourmous progress in the last 10 years

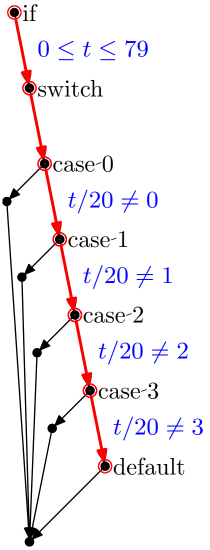
- Further scalability improvements in recent years because of efficient word-level reasoning and array decision procedures

Let's Look at Another Path

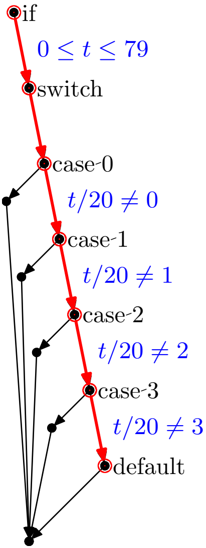


Let's Look at Another Path

CBMC

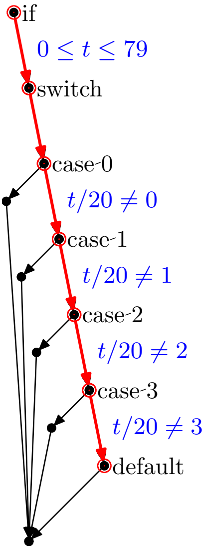


Let's Look at Another Path



$$
\begin{array}{ll}
 & 0 \leq t \leq 79 \\
\wedge & t/20 \neq 0 \\
\wedge & t/20 \neq 1 \\
\wedge & t/20 \neq 2 \\
\wedge & t/20 \neq 3
\end{array}
$$

Let's Look at Another Path



$$
\begin{array}{cl}
 & 0 \leq t \leq 79 \\
\wedge & t/20 \neq 0 \\
\wedge & t/20 \neq 1 \\
\wedge & t/20 \neq 2 \\
\wedge & t/20 \neq 3
\end{array}
$$

That is UNSAT, so the assertion is unreachable.

What If a Variable is Assigned Twice?

```
x=0;

if (y>=0)
  x++;
```

Rename appropriately:

$$
\begin{array}{ll}
 & x = 0 \\
\land & y \geq 0 \\
\land & x = x + 1
\end{array}
$$

What If a Variable is Assigned Twice?

```
x=0;

if (y>=0)
  x++;
```

Rename appropriately:

$$
\begin{array}{rl}
 & x_1 = 0 \\
\wedge & y_0 \geq 0 \\
\wedge & x_1 = x_0 + 1
\end{array}
$$

This is a special case of *SSA* (static single assignment)

Pointers

How do we handle dereferencing in the program?

Pointers

How do we handle dereferencing in the program?

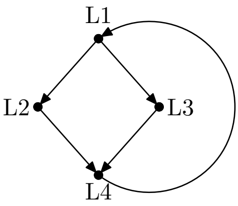
$$\textbf{int}\ *p;$$
$$p{=}malloc(\textbf{sizeof}(\textbf{int})*5);$$
$$\ldots$$
$$p[1]{=}100;$$

$$\begin{aligned} & p_1 = \&\mathit{DO1} \\ \wedge\ & \mathit{DO1}_1 = (\lambda i. \\ & i = 1 ? 100 : \mathit{DO1}_0[i]) \end{aligned}$$

Track a ‘may-point-to’ abstract state while simulating!

Scalability of Path Search

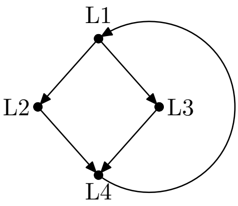
Let's consider the following CFG:



This is a loop with an `if` inside.

Scalability of Path Search

Let's consider the following CFG:



This is a loop with an `if` inside.

Q: how many paths for n iterations?

Bounded Model Checking

- Bounded Model Checking (BMC) is the most successful formal validation technique in the *hardware* industry
- Advantages:
  - ✔ Fully automatic
  - ✔ Robust
  - ✔ Lots of subtle bugs found
- Idea: only look for bugs up to specific depth
- Good for many applications, e.g., embedded systems

Transition Systems

CBMC

Definition: A transition system is a triple (S, S_0, T) with

- set of states S,
- a set of initial states $S_0 \subset S$, and
- a transition relation $T \subset (S \times S)$.

The set S_0 and the relation T can be written as their characteristic functions.

Unwinding a Transition System

Q: How do we avoid the exponential path explosion?

We just "concatenate" the transition relation T:

Unwinding a Transition System

Q: How do we avoid the exponential path explosion?

We just "concatenate" the transition relation T:

$$\bullet \xrightarrow{S_0 \wedge T} \bullet$$

Unwinding a Transition System

Q: How do we avoid the exponential path explosion?

We just "concatenate" the transition relation T:

$$\begin{tikzcd} \bullet \arrow[r, "S_0 \wedge T"] & \overset{\wedge}{\bullet} \arrow[r, "T"] & \bullet \end{tikzcd}$$

Unwinding a Transition System

Q: How do we avoid the exponential path explosion?

We just "concatenate" the transition relation T:

$$S_0 \wedge T \quad \wedge \quad T \quad \wedge \quad \cdots \quad \wedge \quad T$$

$$\bullet \xrightarrow{} \bullet \xrightarrow{} \bullet \quad \cdots \quad \bullet \xrightarrow{} \bullet$$

Unwinding a Transition System

Q: How do we avoid the exponential path explosion?

We just "concatenate" the transition relation T:

$$\begin{tikzcd} s_0 \arrow[r, "S_0 \wedge T"] & s_1 \arrow[r, "\wedge \quad T"] & s_2 \arrow[r, phantom, "\wedge \quad \cdots \quad \wedge"] & s_{k-1} \arrow[r, "T"] & s_k \end{tikzcd}$$

Unwinding a Transition System

As formula:

$$S_0(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1})$$

Satisfying assignments for this formula are traces through the transition system

Example

$$T \subseteq \mathbb{N}_0 \times \mathbb{N}_0$$

$$T(s, s') \iff s'.x = s.x + 1$$

...and let $S_0(s) \iff s.x = 0 \vee s.x = 1$

Example

$$T \subseteq \mathbb{N}_0 \times \mathbb{N}_0$$

$$T(s, s') \iff s'.x = s.x + 1$$

...and let $S_0(s) \iff s.x = 0 \vee s.x = 1$

An unwinding for depth 4:

$$\begin{array}{ll} & (s_0.x = 0 \vee s_0.x = 1) \\ \wedge & s_1.x = s_0.x + 1 \\ \wedge & s_2.x = s_1.x + 1 \\ \wedge & s_3.x = s_2.x + 1 \\ \wedge & s_4.x = s_3.x + 1 \end{array}$$

Checking Reachability Properties

Suppose we want to check a property of the form $\mathbf{AG}p$.

Checking Reachability Properties

Suppose we want to check a property of the form $\mathbf{AG}p$.

We then want at least one state s_i to satisfy $\neg p$:

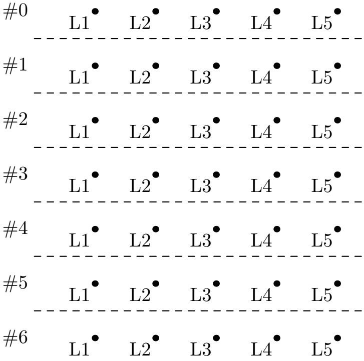
$$S_0(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \quad \wedge \quad \bigvee_{i=0}^{k} \neg p(s_i)$$

Satisfying assignments are counterexamples for the $\mathbf{AG}p$ property

Unwinding Software

We can do exactly that for our transition relation for software.

E.g., for a program with 5 locations, 6 unwindings:

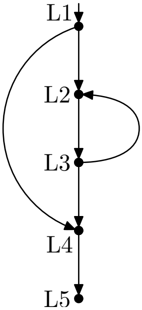


Unwinding Software

Problem: obviously, most of the formula is never ’used’, as only few sequences of PCs correspond to a path.

Unwinding Software

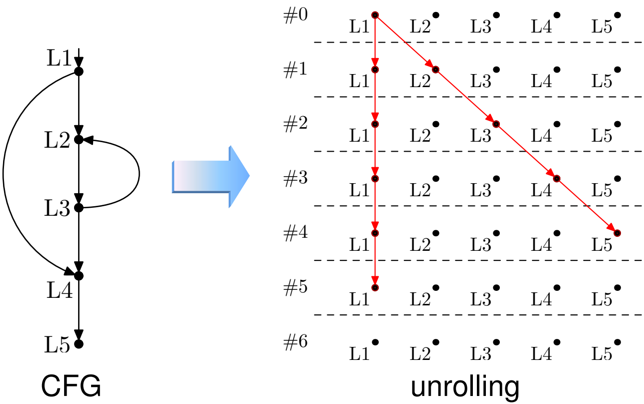
Example:



CFG

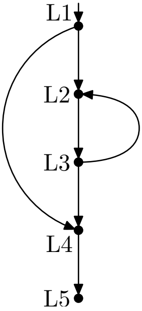
Unwinding Software

Example:



Unwinding Software

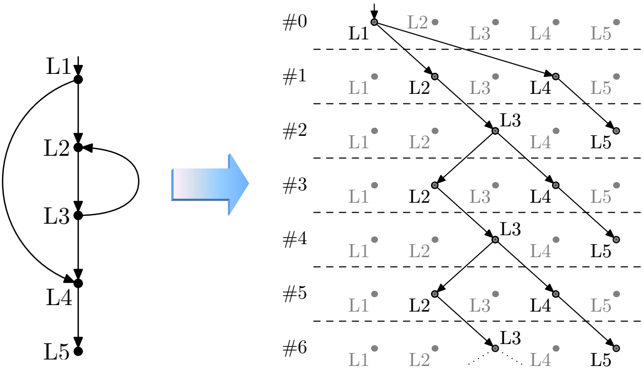
Optimization:
don't generate the parts of the formula that are not 'reachable'



CFG

Unwinding Software

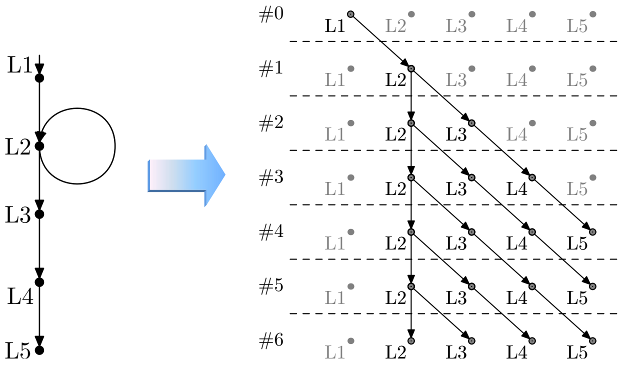
Optimization:
don't generate the parts of the formula that are not 'reachable'



CFG unrolling

Unwinding Software

Problem:



CFG

unrolling

Unwinding Software

- Unwinding T with bound k results in a formula of size

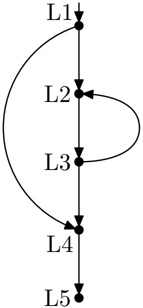
$$|T| \cdot k$$

- If we assume a k that is only linear in $|T|$,
  we get get a formula with size $O(|T|^2)$

- Can we do better?

Unrolling Loops

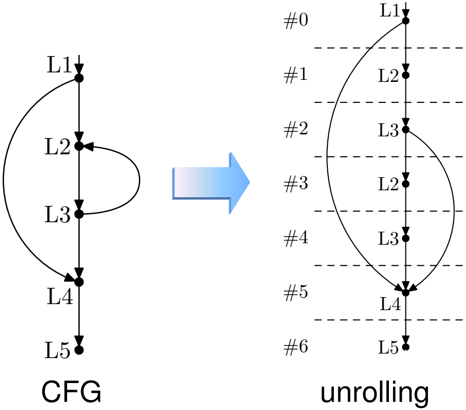
Idea: do exactly one location in each timeframe:



CFG

Unrolling Loops

Idea: do exactly one location in each timeframe:



Unrolling Loops

✔ More effective use of the formula size

✔ Graph has fewer merge nodes,
the formula is easier for the solvers

✘ Not all paths of length k are encoded
$\rightarrow$ the bound needs to be larger

Unrolling Loops

This essentially amounts to unwinding loops:

$\textbf{while}(\textsf{cond})$
    $\textsf{Body};$

Unrolling Loops

This essentially amounts to unwinding loops:

```
if(cond) {
  Body;
  while(cond)
    Body;
}
```

Unrolling Loops

This essentially amounts to unwinding loops:

```
if(cond) {
  Body;
  if(cond) {
    Body;
    while(cond)
      Body;
  }
}
```

Unrolling Loops

This essentially amounts to unwinding loops:

```
if(cond) {
  Body;
  if(cond) {
    Body;
    if(cond) {
      Body;
      while(cond)
        Body;
    }
  }
}
```

Unrolling Loops

This essentially amounts to unwinding loops:

```
if(cond) {
  Body;
  if(cond) {
    Body;
    if(cond) {
      Body;
      assume(!cond);

    }
  }
}
```

Completeness

BMC, as discussed so far, is incomplete.
It only refutes, and does not prove.

How can we fix this?

Unwinding Assertions

Let's revisit the loop unwinding idea:

$\mathbf{while}(\mathsf{cond})$
$\quad\mathsf{Body};$

Unwinding Assertions

Let's revisit the loop unwinding idea:

```
if(cond) {
  Body;
  while(cond)
    Body;
}
```

Unwinding Assertions

Let's revisit the loop unwinding idea:

```
if(cond) {
  Body;
  if(cond) {
    Body;
    while(cond)
      Body;
  }
}
```

Unwinding Assertions

Let's revisit the loop unwinding idea:

```
if(cond) {
  Body;
  if(cond) {
    Body;
    if(cond) {
      Body;
      while(cond)
        Body;
    }
  }
}
```

Unwinding Assertions

Let's revisit the loop unwinding idea:

```
if(cond) {
  Body;
  if(cond) {
    Body;
    if(cond) {
      Body;
      assert(!cond);

    }
  }
}
```

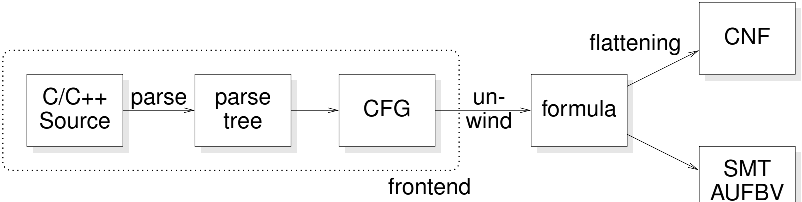
Unwinding Assertions

CBMC

- We replace the assumption we have used earlier to cut off paths by an assertion
- ✔ This allows us to prove that we have done enough unwinding
- This is a proof of a high-level worst-case execution time (WCET)
- Very appropriate for embedded software

CBMC Toolflow: Summary

1. Parse, build CFG
2. Unwind CFG, form formula
3. Formula is solved by SAT/SMT



Solving the Decision Problem

Suppose we have used some unwinding, and have built the formula.

For bit-vector arithmetic, the standard way of deciding satisfiability of the formula is *flattening*,
followed by a call to a propositional SAT solver.

In the SMT context: SMT-$\mathcal{BV}$

Bit-vector Flattening

- This is easy for the bit-wise operators.
- Denote the Boolean variable for bit i of term t by $\mu(t)_i$.
- Example for $a \,|_{[l]}\, b$:

$$\bigwedge_{i=0}^{l-1} (\mu(t)_i = (a_i \vee b_i))$$

(read $x = y$ over bits as $x \iff y$)

Bit-vector Flattening

- This is easy for the bit-wise operators.
- Denote the Boolean variable for bit i of term t by $\mu(t)_i$.
- Example for $a \,|_{[l]}\, b$:

$$\bigwedge_{i=0}^{l-1} (\mu(t)_i = (a_i \vee b_i))$$

(read $x = y$ over bits as $x \iff y$)

- We can transform this into CNF using Tseitin's method.

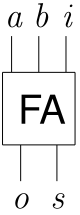
Flattening Bit-Vector Arithmetic

How to flatten $a + b$?

CBMC

How to flatten $a + b$?

$\longrightarrow$ we can build a *circuit* that adds them!



Full Adder

$$s \equiv (a + b + i\,) \bmod 2 \equiv a \oplus b \oplus i$$

$$o \equiv (a + b + i\,) \operatorname{div} 2 \equiv a \cdot b + a \cdot i + b \cdot i$$

The full adder in CNF:

$$(a \vee b \vee \neg o) \wedge (a \vee \neg b \vee i \vee \neg o) \wedge (a \vee \neg b \vee \neg i \vee o) \wedge$$
$$(\neg a \vee b \vee i \vee \neg o) \wedge (\neg a \vee b \vee \neg i \vee o) \wedge (\neg a \vee \neg b \vee o)$$

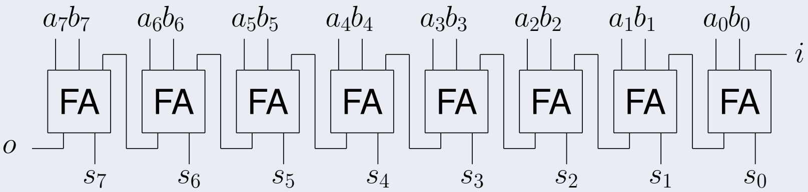
Flattening Bit-Vector Arithmetic

Ok, this is good for one bit! How about more?

Flattening Bit-Vector Arithmetic

Ok, this is good for one bit! How about more?

8-Bit ripple carry adder (RCA)



- Also called *carry chain adder*
- Adds l variables
- Adds $6 \cdot l$ clauses

Multipliers

- Multipliers result in very hard formulas
- Example:

$$a \cdot b = c \land b \cdot a \neq c \land x < y \land x > y$$

CNF: About 11000 variables,
unsolvable for current SAT solvers

- Similar problems with division, modulo
- Q: Why is this hard?

Multipliers

CBMC

- Multipliers result in very hard formulas
- Example:

$$a \cdot b = c \wedge b \cdot a \neq c \wedge x < y \wedge x > y$$

  CNF: About 11000 variables,
  unsolvable for current SAT solvers

- Similar problems with division, modulo
- Q: Why is this hard?
- Q: How do we fix this?

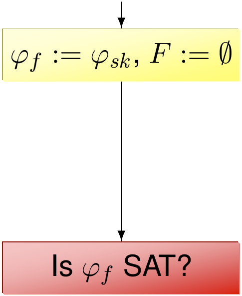
Incremental Flattening

$$\downarrow$$

$$\boxed{\varphi_f := \varphi_{sk},\ F := \emptyset}$$

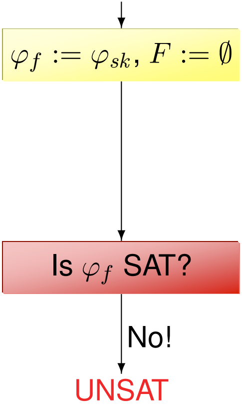
φ_{sk}: Boolean part of φ
F: set of terms that are in the encoding

Incremental Flattening



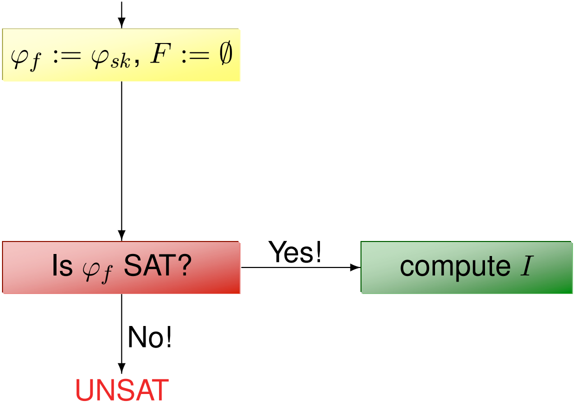
φ_{sk}: Boolean part of φ
F: set of terms that are in the encoding

Incremental Flattening



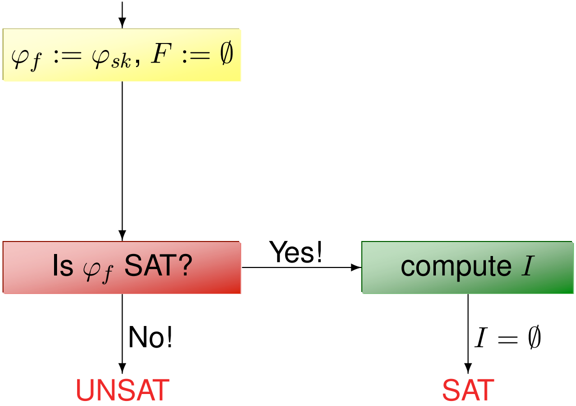
φ_{sk}: Boolean part of φ
F: set of terms that are in the encoding

Incremental Flattening



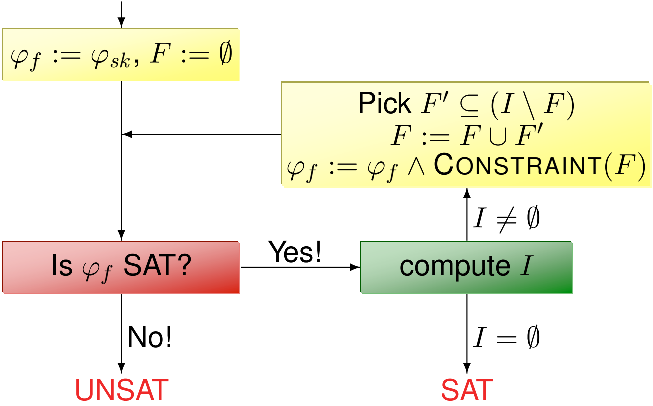
φ_{sk}: Boolean part of φ
F: set of terms that are in the encoding
I: set of terms that are inconsistent with the current assignment

Incremental Flattening



φ_{sk}: Boolean part of φ
F: set of terms that are in the encoding
I: set of terms that are inconsistent with the current assignment

Incremental Flattening



φ_{sk}: Boolean part of φ
F: set of terms that are in the encoding
I: set of terms that are inconsistent with the current assignment

Incremental Flattening

- Idea: add 'easy' parts of the formula first
- Only add hard parts when needed
- φ_f only gets stronger – use an incremental SAT solver