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

\[
\text{if } (0 \leq t \leq 79) \land \text{switch } (t/20) \begin{cases} 
  0: & \text{TEMP2} = (B \land C) \lor (\neg B \land D), \text{TEMP3} = k_1^2, \text{break;} \\
  1: & \text{TEMP2} = (B \oplus C \oplus D), \text{TEMP3} = k_2^1, \text{break;} \\
  2: & \text{TEMP2} = (B \land C) \lor (B \land D) \lor (C \land D), \text{TEMP3} = k_3^1, \text{break;} \\
  3: & \text{TEMP2} = B \oplus C \oplus D, \text{TEMP3} = k_4^1, \text{break;} \\
  \text{default:} & \text{assert(0)} \end{cases}
\]

Bounded Program Analysis

Goal: check properties of the form AGp, say assertions.

Idea: follow paths through the CFG to an assertion, and build a formula that corresponds to the path
We pass
\[ 0 \leq t \leq 79 \]
\[ t/20 \neq 0 \]
\[ t/20 = 1 \]
\[ TEMP2 = B \oplus C \oplus D \]
\[ TEMP3 = K \cdot 2 \]
to a decision procedure, and obtain a satisfying assignment, say:
\[ t \mapsto 21, \ B \mapsto 0, \ C \mapsto 0, \ D \mapsto 0, \ K \cdot 2 \mapsto 10, \]
\[ TEMP2 \mapsto 0, \ TEMP3 \mapsto 10 \]

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

Enabling Technology: SAT

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

Let’s Look at Another Path

if
switch
case/suppress0
case/suppress1
case/suppress2
case/suppress3
default
0 ≤ t ≤ 79
\[ t/20 \neq 0 \]
\[ t/20 = 1 \]
\[ t/20 \neq 2 \]
\[ t/20 \neq 3 \]

That is UNSAT, so the assertion is unreachable.

What If a Variable is Assigned Twice?

\[ x=0; \]
\[ \text{if}(y>0) \]
\[ x++; \]

Rename appropriately:
\[ x1 = 0 \]
\[ y0 \geq 0 \]
\[ x1 = x0 + 1 \]

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

How do we handle dereferencing in the program?

```c
int *p;
p = malloc(sizeof(int) * 5);
...
p[1] = 100;
```

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.

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

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 \(T\):

```
S_0 \land T \land T \land \ldots \land T
```

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 \( S_0(s) \iff s.x = 0 \land s.x = 1 \)

An unwinding for depth 4:

\[
\begin{align*}
(s_0.x = 0 \lor s_0.x &= 1) \\
\land s_1.x &= s_0.x + 1 \\
\land s_2.x &= s_1.x + 1 \\
\land s_3.x &= s_2.x + 1 \\
\land s_4.x &= s_3.x + 1 \\
\end{align*}
\]

Checking Reachability Properties

Suppose we want to check a property of the form \( \text{AG} p \).

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

\[
S_0(s_0) \land \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \land \bigvee_{i=0}^{k} \neg p(s_i)
\]

Satisfying assignments are counterexamples for the \( \text{AG} p \) property.

Unwinding Software

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

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

\[
\begin{align*}
\theta^0: & L_1^* L_2^* L_3^* L_4^* L_5^* \\
\theta^1: & L_1^* L_2^* L_3^* L_4^* L_5^* \\
\theta^2: & L_1^* L_2^* L_3^* L_4^* L_5^* \\
\theta^3: & L_1^* L_2^* L_3^* L_4^* L_5^* \\
\theta^4: & L_1^* L_2^* L_3^* L_4^* L_5^* \\
\theta^5: & L_1^* L_2^* L_3^* L_4^* L_5^* \\
\theta^6: & L_1^* L_2^* L_3^* L_4^* L_5^* \\
\end{align*}
\]

Optimization:

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

Unwinding Software

Example:

Unwinding Software

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

**Problem:**

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 a formula with size $O(|T|^2)$
- Can we do better?

### Unrolling Loops

**Idea:** do exactly one location in each timeframe:

- 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

### 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:

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

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

Bit-vector Flattening

- This is easy for the bit-wise operators.
- Denote the Boolean variable for bit i of term t by µ(t)ᵢ.
- Example for a ⊕ b:
  \[
  \bigwedge_{i=0}^{t-1} (µ(t)ᵢ \equiv (aᵢ \lor bᵢ))
  \]
  (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 \(a + b\)?

→ we can build a circuit that adds them!

\[
\begin{align*}
    a + b & \equiv (a + b + i) \mod 2 \equiv (a \oplus b) \oplus i \\
    o & \equiv (a + b + i) \div 2 \equiv a \cdot b + a \cdot i + b \cdot i
\end{align*}
\]

The full adder in CNF:

\[
(a \lor b \lor \neg o) \land (a \lor \neg b \lor i \lor \neg o) \land (a \lor \neg b \lor i \lor \neg o) \land \\
(\neg a \lor b \lor i \lor \neg o) \land (\neg a \lor b \lor i \lor \neg o) \land (\neg a \lor \neg b \lor o)
\]
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?
▶ Q: How do we fix this?

Incremental Flattening

\( \varphi_f := \varphi_{sk}, F := \emptyset \)

\( F' \subseteq (I \setminus F) \)
\( F' := F \cup F' \)
\( \varphi_f := \varphi_f \land \text{CONSTRAINT}(F) \)

Is \( \varphi_f \) SAT?
Yes!

\( F \subseteq \emptyset \)
\( I = \emptyset \)
SAT

\( \varphi_{sk} \): Boolean part of \( \varphi \)
\( F \): set of terms that are in the encoding
\( I \): set of terms that are inconsistent with the current assignment

\( \varphi_f \) only gets stronger – use an incremental SAT solver