Programm- & Systemverifikation

Bounded Model Checking

Georg Weissenbacher

184.741

What happened so far

- How bugs come into being:
  - Fault – cause of an error (e.g., mistake in coding)
  - Error – *incorrect* state that may lead to failure
  - Failure – deviation from *desired* behaviour
- We specified *intended* behaviour using assertions
- We proved our programs correct (inductive invariants).
- We learned how to test programs.
- We heard about logical formalisms:
  - Propositional Logic
  - First Order Logic
- Last time we learned about Hoare Logic.

Hoare's Axioms

$$\frac{}{\{P[E/\mathtt{x}]\}\ \mathtt{x:=}E\ \{P\}}$$

$$\frac{\{P\}\ C_1\ \{Q\}\ ,\ \{Q\}\ C_2\ \{R\}}{\{P\}\ C_1\ ;\ C_2\ \{R\}}$$

$$\frac{\{B \wedge P\}\ C_1\ \{Q\}\quad \{\neg B \wedge P\}\ C_2\ \{Q\}}{\{P\}\ \text{if } B \text{ then } C_1 \text{ else } C_2\ \{Q\}}$$

$$\frac{P' \rightarrow P \quad \{P\}\ C\ \{Q\} \quad Q \rightarrow Q'}{\{P'\}\ C\ \{Q'\}}$$

$$\frac{\{P \wedge B\}\ C\ \{P\}}{\{P\}\ \text{while } B \text{ do } C\ \{\neg B \wedge P\}}$$

Can We Automate Hoare Reasoning?

- Extremely hard to "synthesize" invariants!
- Automating loop rule is impossible.

Can We Automate Hoare Reasoning?

- Extremely hard to “synthesize” invariants!
- Automating loop rule is impossible.
- What if we restrict ourselves to the other rules?

Can We Automate Hoare Reasoning?

- Extremely hard to “synthesize” invariants!
- Automating loop rule is impossible.
- What if we restrict ourselves to the other rules?
  - . . . and unwind loops only *n* times.
  - Sufficient for bug finding.

Predicate Transformers

$$\{P\}\ \texttt{stmt}\ \{Q\}$$

"Forwards with Hoare": Given P, how can we compute Q?

Predicate Transformers

$$\{P\}\ \texttt{stmt}\ \{Q\}$$

"Forwards with Hoare": Given P, how can we compute Q?

Definition (Strongest Postcondition)

The *strongest post-condition* $sp(\texttt{stmt}, P)$

- for a statement `stmt`
- with respect to a pre-condition P

is the strongest predicate Q such that $\{P\}\ \texttt{stmt}\ \{Q\}$ holds.

Predicate Transformers

$$\{P\}\ \texttt{stmt}\ \{Q\}$$

"Forwards with Hoare": Given P, how can we compute Q?

Definition (Strongest Postcondition)

The *strongest post-condition* $sp(\texttt{stmt}, P)$

- for a statement $\texttt{stmt}$
- with respect to a pre-condition P

is the strongest predicate Q such that $\{P\}\ \texttt{stmt}\ \{Q\}$ holds.

I.e., $\{P\}\ \texttt{stmt}\ \{Q'\}$ is equivalent to $sp(\texttt{stmt}, P) \Rightarrow Q'$.

Predicate Transformers: Strongest Post-condition

Strongest Post-condition for assignment `x:=`e

$$sp(\mathtt{x} := e, P) \stackrel{\text{def}}{=} \exists \mathtt{x}' \,.\, (\mathtt{x} = e[\mathtt{x}/\mathtt{x}']) \wedge P[\mathtt{x}/\mathtt{x}']$$

Predicate Transformers: Strongest Post-condition

Strongest Post-condition for assignment $\mathtt{x}{:=}e$

$$sp(\mathtt{x} := e, P) \stackrel{\text{def}}{=} \exists \mathtt{x}' \,.\, (\mathtt{x} = e[\mathtt{x}/\mathtt{x}']) \wedge P[\mathtt{x}/\mathtt{x}']$$

- Example:

$$sp(\mathtt{x} := \mathtt{x} + 1, (\mathtt{x} \leq 10)) \;=$$

Predicate Transformers: Strongest Post-condition

Strongest Post-condition for assignment `x:=`*e*

$$sp(\mathtt{x} := e, P) \stackrel{\text{def}}{=} \exists \mathtt{x}' \,.\, (\mathtt{x} = e[\mathtt{x}/\mathtt{x}']) \wedge P[\mathtt{x}/\mathtt{x}']$$

- Example:

$$sp(\mathtt{x} := \mathtt{x} + 1, (\mathtt{x} \leq 10)) \;=\; \exists \mathtt{x}' \,.\, (\mathtt{x} = \mathtt{x}' + 1) \wedge (\mathtt{x}' \leq 10)$$

Predicate Transformers: Strongest Post-condition

Strongest Post-condition for assignment `x:=`*e*

$$sp(\mathtt{x} := e, P) \stackrel{\text{def}}{=}$$
$$\exists \mathtt{x}' \,.\, (\mathtt{x} = e[\mathtt{x}/\mathtt{x}']) \wedge P[\mathtt{x}/\mathtt{x}']$$

- Example:

$$sp(\mathtt{x} := \mathtt{x} + 1, (\mathtt{x} \leq 10)) \;=\; \exists \mathtt{x}' \,.\, (\mathtt{x} = \mathtt{x}' + 1) \wedge (\mathtt{x}' \leq 10)$$

- Can we get rid of the quantifier?

Predicate Transformers: Strongest Post-condition

Strongest Post-condition for assignment `x:=`*e*

$$sp(\mathtt{x} := e, P) \stackrel{\text{def}}{=} \exists \mathtt{x}' \,.\, (\mathtt{x} = e[\mathtt{x}/\mathtt{x}']) \land P[\mathtt{x}/\mathtt{x}']$$

- Example:

$$sp(\mathtt{x} := \mathtt{x} + 1, (\mathtt{x} \leq 10)) \;=\; \exists \mathtt{x}' \,.\, (\mathtt{x} = \mathtt{x}' + 1) \land (\mathtt{x}' \leq 10)$$

- Can we get rid of the quantifier?

$$\mathtt{x}' = (\mathtt{x} - 1)$$

Predicate Transformers: Strongest Post-condition

Strongest Post-condition for assignment `x:=`*e*

$$sp(\mathtt{x} := e, P) \stackrel{\text{def}}{=}$$
$$\exists \mathtt{x}' \,.\, (\mathtt{x} = e[\mathtt{x}/\mathtt{x}']) \wedge P[\mathtt{x}/\mathtt{x}']$$

- Example:

$$sp(\mathtt{x} := \mathtt{x} + 1, (\mathtt{x} \leq 10)) \;=\; \exists \mathtt{x}' \,.\, (\mathtt{x} = \mathtt{x}' + 1) \wedge (\mathtt{x}' \leq 10)$$

- Can we get rid of the quantifier?

$$\mathtt{x}' = (\mathtt{x} - 1)$$
$$\exists \mathtt{x}' \,.\, (\mathtt{x} = (\mathtt{x} - 1) + 1) \wedge ((\mathtt{x} - 1) \leq 10)$$

Predicate Transformers: Strongest Post-condition

Strongest Post-condition for assignment $\texttt{x:=}e$

$$\mathit{sp}(\mathtt{x} := e, P) \stackrel{\text{def}}{=}$$
$$\exists \mathtt{x}' \,.\, (\mathtt{x} = e[\mathtt{x}/\mathtt{x}']) \wedge P[\mathtt{x}/\mathtt{x}']$$

- Example:

$$\mathit{sp}(\mathtt{x} := \mathtt{x} + 1, (\mathtt{x} \leq 10)) \;=\; \exists \mathtt{x}' \,.\, (\mathtt{x} = \mathtt{x}' + 1) \wedge (\mathtt{x}' \leq 10)$$

- Can we get rid of the quantifier?

$$\mathtt{x}' = (\mathtt{x} - 1)$$
$$(\mathtt{x} = (\mathtt{x} - 1) + 1) \wedge ((\mathtt{x} - 1) \leq 10)$$

- *Only* if underlying logic allows quantifier elimination!

Predicate Transformers: Strongest Post-condition

Strongest Post-condition for assertion $\texttt{assert}(R)$

$$\mathit{sp}(\texttt{assert}(R), P) \stackrel{\text{def}}{=} P \wedge R$$

Predicate Transformers: Strongest Post-condition

Strongest Post-condition for assertion $\texttt{assert}(R)$

$$sp(\texttt{assert}(R), P) \stackrel{\text{def}}{=} P \land R$$

- Note: If P and R are inconsistent, then $sp(\texttt{assert}(R), P) = \text{false}$

Strongest Post-condition for sequential execution

$$sp(\mathtt{stmt}_1;\ \mathtt{stmt}_2, P) \stackrel{\text{def}}{=} sp(\mathtt{stmt}_2, sp(\mathtt{stmt}_1, P))$$

Strongest Post-condition for sequential execution

$$\mathit{sp}(\mathtt{stmt}_1;\ \mathtt{stmt}_2, P) \stackrel{\text{def}}{=}$$

$$\mathit{sp}(\mathtt{stmt}_2, \mathit{sp}(\mathtt{stmt}_1, P))$$

Strongest Post-condition for sequential execution

$$sp(\mathtt{stmt}_1;\ \mathtt{stmt}_2, P) \stackrel{\text{def}}{=}$$

$$sp(\mathtt{stmt}_2, sp(\mathtt{stmt}_1, P))$$

Predicate Transformers: Strongest Post-condition

Strongest Post-condition for sequential execution

$$sp(\texttt{stmt}_1;\ \texttt{stmt}_2, P) \stackrel{\text{def}}{=}$$

$$sp(\texttt{stmt}_2, sp(\texttt{stmt}_1, P))$$

Example:

$$\{\mathtt{x} \leq \mathtt{y}\}\ \mathtt{x} := \mathtt{x} + 1;\ \{\qquad\qquad\qquad\qquad\qquad\qquad\}$$

$$\texttt{assert}(x > 0)$$

$$\{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\}$$

Predicate Transformers: Strongest Post-condition

Strongest Post-condition for sequential execution

$$sp(\mathtt{stmt}_1;\ \mathtt{stmt}_2, P) \stackrel{\text{def}}{=}$$

$$sp(\mathtt{stmt}_2, sp(\mathtt{stmt}_1, P))$$

Example:

$$\{\mathtt{x} \le \mathtt{y}\}\ \mathtt{x} := \mathtt{x} + 1;\ \{\exists \mathtt{x}' \,.\, (\mathtt{x} = \mathtt{x}' + 1) \wedge (\mathtt{x}' \le \mathtt{y})\}$$

$$\mathtt{assert}(x > 0)$$

$$\{ \qquad\qquad\qquad\qquad \}$$

Strongest Post-condition for sequential execution

$$sp(\mathtt{stmt}_1;\ \mathtt{stmt}_2, P) \stackrel{\text{def}}{=}$$

$$sp(\mathtt{stmt}_2, sp(\mathtt{stmt}_1, P))$$

Example:

$$\{\mathtt{x} \le \mathtt{y}\}\ \mathtt{x} := \mathtt{x} + 1;\ \{\exists \mathtt{x}' \,.\, (\mathtt{x} = \mathtt{x}' + 1) \wedge (\mathtt{x}' \le \mathtt{y})\}$$

$$\mathtt{assert}(x > 0)$$

$$\{\ (\exists \mathtt{x}' \,.\, (\mathtt{x} = \mathtt{x}' + 1) \wedge (\mathtt{x}' \le \mathtt{y})) \wedge (\mathtt{x} > 0)\ \}$$

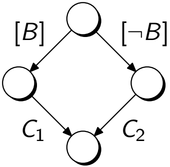
Strongest Post-condition for conditionals

$$sp(\texttt{if } B \texttt{ then } C_1 \texttt{ else } C_2, P) \stackrel{\text{def}}{=} sp(C_1, B \wedge P) \vee sp(C_2, \neg B \wedge P)$$

Strongest Post-condition for conditionals

$$sp(\texttt{if}\ B\ \texttt{then}\ C_1\ \texttt{else}\ C_2, P) \stackrel{\text{def}}{=} sp(C_1, B \wedge P) \vee sp(C_2, \neg B \wedge P)$$

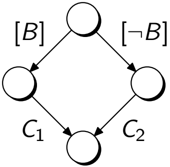
- What does this mean in terms of program paths?



Strongest Post-condition for conditionals

$$sp(\texttt{if } B \texttt{ then } C_1 \texttt{ else } C_2, P) \stackrel{\text{def}}{=} sp(C_1, B \wedge P) \vee sp(C_2, \neg B \wedge P)$$

- What does this mean in terms of program paths?

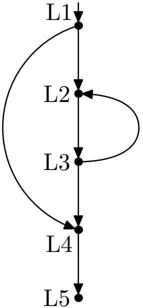


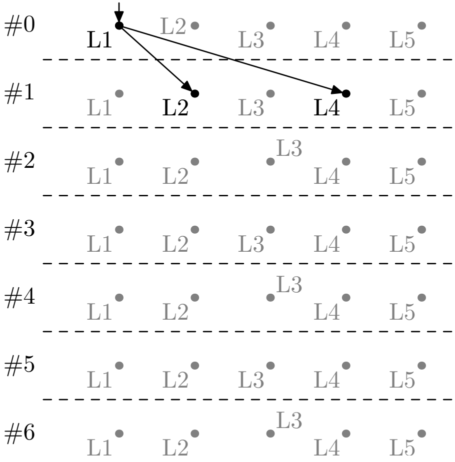
- Merging two paths!

Unwinding Paths

Remember our Test-Case-Generation technique?

- Unwinds paths/loops *without* merging!

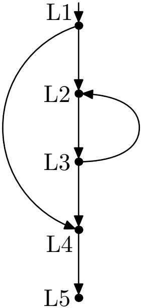


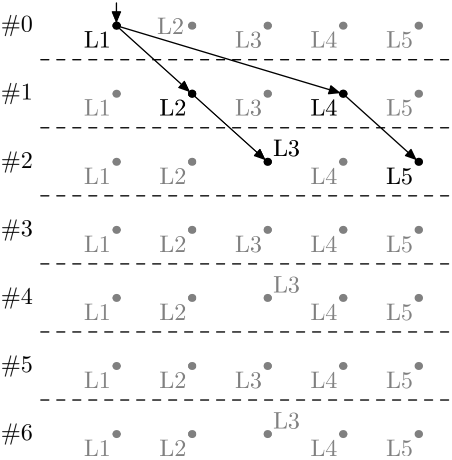


Unwinding Paths

Remember our Test-Case-Generation technique?

- Unwinds paths/loops *without* merging!

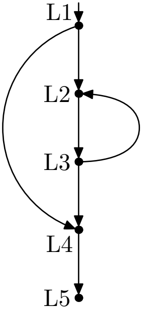


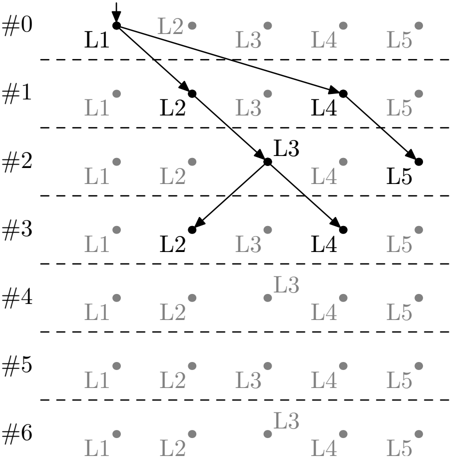


Unwinding Paths

Remember our Test-Case-Generation technique?

- Unwinds paths/loops *without* merging!

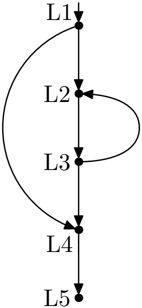


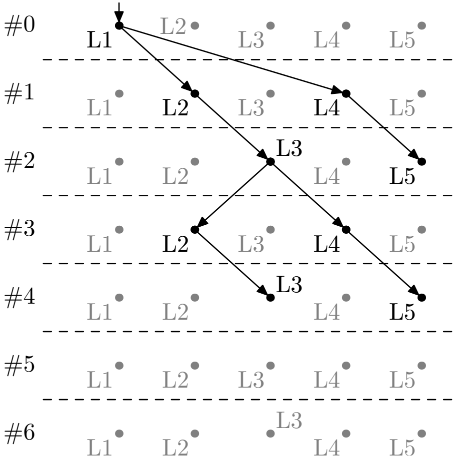


Unwinding Paths

Remember our Test-Case-Generation technique?

- Unwinds paths/loops *without* merging!

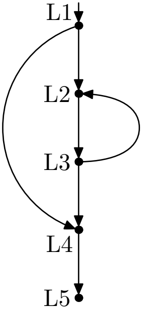


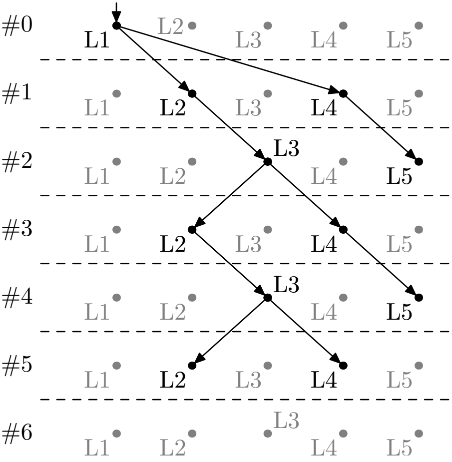


Unwinding Paths

Remember our Test-Case-Generation technique?

- Unwinds paths/loops *without* merging!

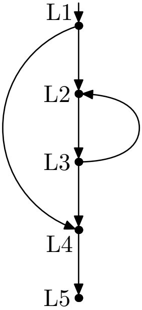


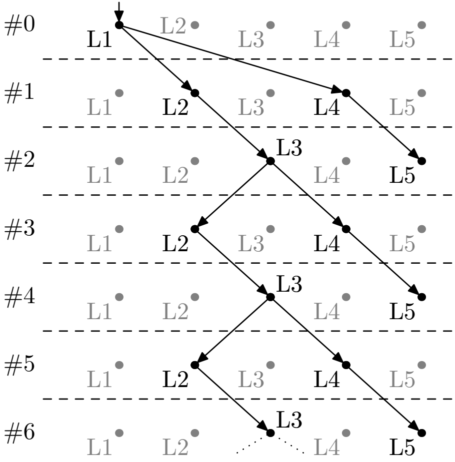


Unwinding Paths

Remember our Test-Case-Generation technique?

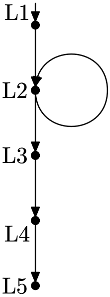
- Unwinds paths/loops *without* merging!



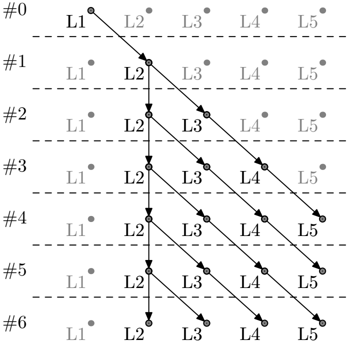


Unwinding Transition Relations

- Is path-wise unwinding a good strategy?
  - Previous unwinding contains 3 copies of L4 and L5!
  - Path enumeration → exponential blowup!



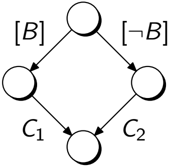
CFG

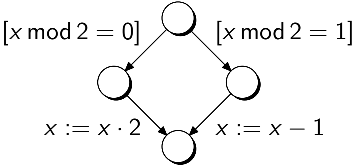


unrolling

Unwinding Loops

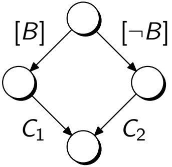
- Recall $sp(\texttt{if}\ (B)\ \texttt{then}\ S\ \texttt{else}\ T, P)$

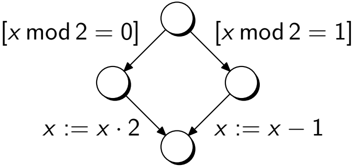




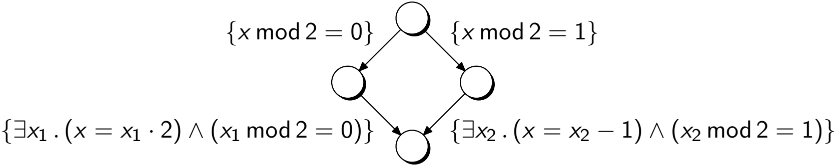
Unwinding Loops

- Recall $sp(\texttt{if}\ (B)\ \texttt{then}\ S\ \texttt{else}\ T, P)$



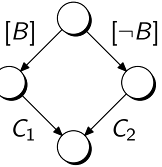


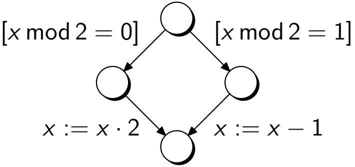
We get:



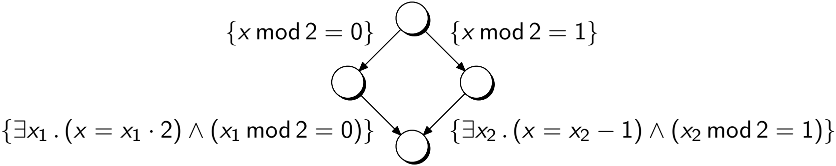
Unwinding Loops

- Recall $sp(\texttt{if}\ (B)\ \texttt{then}\ S\ \texttt{else}\ T, P)$





We get:

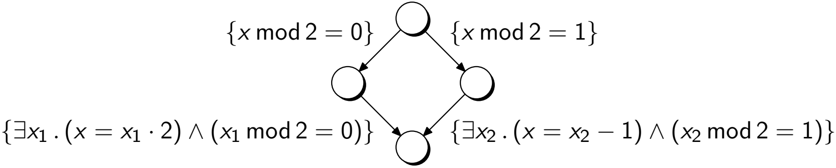


Merge:

$$(\exists x_1 \,.\, (x = x_1 \cdot 2) \wedge (x_1 \bmod 2 = 0)) \vee$$
$$(\exists x_2 \,.\, (x = x_2 - 1) \wedge (x_2 \bmod 2 = 1))$$

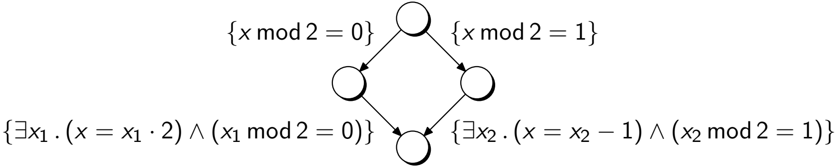
Unwinding Transition Relations

"Choice" of $\mathtt{x}_1, \mathtt{x}_2$ depends on which condition holds!

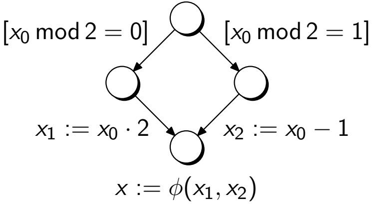


Unwinding Transition Relations

"Choice" of $\mathtt{x}_1, \mathtt{x}_2$ depends on which condition holds!



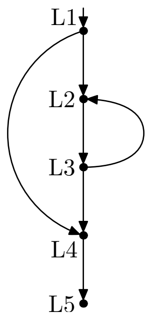
Should look familiar to compiler engineers:

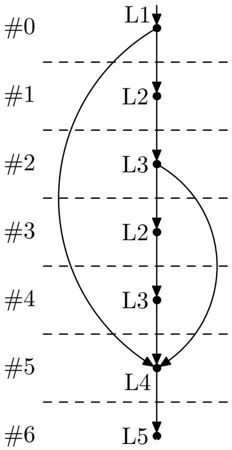


(static single assignment form [Cytron, Ferrante, Rosen, Wegman, Zadeck 1991])

Unwinding Loops

- Idea:
- Unwind loop bodies individually and *merge* on exit





Unwinding Loops

while *B* do *BODY* done

if (*B*) {
    BODY

Unwinding Loops

while *B* do *BODY* done

if (*B*) {
  BODY
    if (*B*) {
      BODY

Unwinding Loops

while *B* do *BODY* done

if (*B*) {
  BODY
    if (*B*) {
      BODY
      if (*B*) {
        BODY

Unwinding Loops

while *B* do *BODY* done

```
if (B) {
  BODY
    if (B) {
      BODY
      if (B) {
        BODY
        if (B) {
          exit();
      }
    }
  }
}
```

Unwinding Loops

while *B* do *BODY* done

```
if (B) {                    if (B) {
  BODY                        BODY
    if (B) {                    if (B) {
      BODY                        BODY
      if (B) {                    if (B) {
        BODY                        BODY
        if (B) {                    if (B) {
          exit();                     assert(false);
      }                           }
    }                           }
  }                           }
}                           }
```

What happens if we replace **exit** with **assert**(false)?

Unwinding Loops: Unwinding Assertions

Assertion fails if loop can be unwound further.

- This is known as “unwinding assertion”.

Bounded Model Checking

① "Unwind" all loops in program *n* times.

② Compute strongest post-condition for *loop-free* program.

- Start with $\{\text{true}\}$ at beginning of program
- Iteratively compute post-condition of each statement
- Merge paths whenever possible

Bounded Model Checking

① "Unwind" all loops in program *n* times.

② Compute strongest post-condition for *loop-free* program.

- Start with $\{\text{true}\}$ at beginning of program
- Iteratively compute post-condition of each statement
- Merge paths whenever possible

- For each program construct `stmt`, we obtain $\{P\}$ `stmt` $\{Q\}$.

Bounded Model Checking

① "Unwind" all loops in program *n* times.

② Compute strongest post-condition for *loop-free* program.

- Start with $\{\text{true}\}$ at beginning of program
- Iteratively compute post-condition of each statement
- Merge paths whenever possible

- For each program construct `stmt`, we obtain $\{P\}$ `stmt` $\{Q\}$.
  - P and Q are *existentially quantified* FOL formulas
  - If we encounter $\{P\}$ `assert(B)` $\{Q\}$:
    report error if $P \wedge \neg B$ is satisfiable

CBMC Example

Example.C:

```
unsigned nondet();

unsigned a[100];

int main(int argc, char** argv) {
  unsigned i;
  for (i=0; i<100; i++) {
    a[i]=nondet();
    __CPROVER_assume(a[i] <= i);
  }
  i=nondet();
  __CPROVER_assume(i<100);
  __CPROVER_assert(a[i]<100, "Not too large");
  return 0;
}
```

CBMC **Command Line Parameters**

- `cbmc --show-claims Example.C`

```
Claim main.assertion.1:
  file Example.C line 14 function main
    Not too large
    a[i] < 100
```

- `cbmc --claim main.assertion.1 --unwinding-assertions --unwind 10 Example.C`

```
Violated property:
  file Example.C line 8 function main
    unwinding assertion loop 0
```

- `cbmc --claim main.assertion.1 Example.C`

```
VERIFICATION SUCCESSFUL
```

Wegner.C:

```
unsigned nondet();

unsigned count(unsigned x) {
  unsigned y, c=0;
  y=x;
  while (y!=0) {
    y=y&(y-1);
    c++;
    __CPROVER_assert(x!=y, "Not equal");
  }
}

int main(int argc, char** argv) {
  unsigned i=nondet();
  return count(i);
}
```

CBMC **Command Line Parameters**

- `cbmc Wegner.C`

```
Unwinding loop 0 iteration 1 file wegner.c line 7
    function count
...
Unwinding loop 0 iteration 3227 file wegner.c line 7
    function count
...
```

- `cbmc --32 --unwind 33 --unwinding-assertions Wegner.C`

```
VERIFICATION SUCCESSFUL
```

CBMC – Modeling

CBMC provides three mechanisms for modeling:

- Assertions: If `assert(c)` is reachable and *c* evaluates to false, CBMC reports a counterexample.
- Non-determinism: If the implementation of a function is not provided, CBMC assumes that the return value is arbitrary.
- Assumptions: If `__CPROVER_assume(c)` reachable, CBMC assumes that *c* is true and silently discards all execution paths for which this doesn't hold.

CBMC – Example

```c
int nondet_int();

int main() {
  int x,y;
  x = nondet_int();
  y = nondet_int();
  __CPROVER_assume(x >= 0 && x<10);
  __CPROVER_assume(y >= 0);
  int r = x+y;
  assert(r>=y);
  return 0;
}
```

Checks whether $\forall x, y. 0 \leq x < 10 \land y \geq 0 \Rightarrow x + y \geq y$ holds.

Non-determinism vs. Randomness

- Randomized Testing
  - Fixed distribution
  - Each path has a certain probability
- Model Checking with non-determinism:
  - *All* paths are checked
  - no path is “more likely”

CBMC – "Test Harness" and "Function Stubs"

Test Harness

- Code that calls the functions under test
- can be highly non-deterministic
  - e.g. order of function calls:

```
switch (nondet()) {
  case 0: foo ();
  break;
  case 1: bar ();
  break;
}
```

  - or non-deterministically initialized parameters

CBMC – "Test Harness" and "Function Stubs"

Function Stubs

- e.g. for modeling functions of an operating system
- clear demarcation of code that needs not be tested
- Can over-approximate behavior:
  - e.g. `int getchar()` with non-deterministic return values
  - or `fread` non-deterministically initializing an array:

```
size_t fread
  (char *ptr, size_t sz, size_t ni, FILE *s)
{
  for (unsigned i = 0; i < (ni * sz); i++)
    ptr[i] = nondet();
}
```

The “C Bounded Model Checker”

`http://www.cprover.org/cbmc`

- A bounded model checking tool for ANSI-C programs
- Checks and detects:
  - User-provided assertions
  - Array access violations (upper and lower bound)
  - Division by zero
  - Arithmetic overflow
  - NaN floating point values
  - Invalid pointers
- `cbmc --unwind 10 program.c` unwinds all loops 10 times

Recap

- How bugs come into being:
  - Fault – cause of an error (e.g., mistake in coding)
  - Error – *incorrect* state that may lead to failure
  - Failure – deviation from *desired* behaviour
- We specified *intended* behaviour using assertions
- We proved our programs correct (inductive invariants).
- We learned how to test programs.
- We heard about logical formalisms:
  - Propositional Logic
  - First Order Logic
- Formal correctness proofs with Hoare Logic.
- Automated software verification with BMC.