# Proving Safety

**with**

## Acceleration and Bounded Model Checking

Daniel Kröning, Matt Lewis,
**Georg Weissenbacher**

TU WIEN

# Challenges in Bounded Software Model Checking

1. High unwinding depth [FMSD'15]
2. Safety proofs [FM'15]

# Challenges in Bounded Software Model Checking

1. High unwinding depth [FMSD'15]
2. Safety proofs [FM'15]

# Challenges in Bounded Software Model Checking

```
memset(buf, 0, len);
```

# Challenges in Bounded Software Model Checking

```
void* memset(void *buf, int c, size_t len){
    for(size_t i=0; i<len; i++)
        ((char*)buf)[i]=c;
}
```

# Challenges in Bounded Software Model Checking

```
void* memset(void *buf, int c, size_t len){
    for(size_t i=0; i<len; i++)
        ((char*)buf)[i]=c;
}
```

# Acceleration

```
i++
```

# Acceleration

$$i' = i + 1$$

# Acceleration

$$\exists n \in \mathbb{N}.\quad \mathtt{i}' = \mathtt{i} + n$$

## Integers vs. Bit-Vectors

- Unsigned integers: $0 \leq \texttt{i} < \infty$
- Unsigned bit-vectors: $0 \leq \texttt{i} \leq \texttt{INT\_MAX}$

# Integers vs. Bit-Vectors

- Unsigned integers: $0 \leq \texttt{i} < \infty$
- Unsigned bit-vectors: $0 \leq \texttt{i} \leq \texttt{INT\_MAX}$

$\texttt{i = i + } n$ for $n > (\texttt{INT\_MAX} - \texttt{i})$:

(arithmetic overflow)

## Does it really matter in practice?

- Last week on `sv-comp@googlegroups.com`:

> *We use the property*
>
> ```
> CHECK(init(main()),
>       LTL(G ! signed_integer_overflow) )
> ```
>
> *[...]*
>
> *The results are quite unpleasant, it seems there are lots of overflow bugs in our benchmarks.*
> *– Matthias Heizmann*

# Bounding Acceleration

- Solution: impose bound $\beta$ on $n$:

$$n < \beta$$

# Bounding Acceleration

- Solution: impose bound $\beta$ on *n*:

$$n < \beta$$

- Example:

$$\exists n \leq \underbrace{(\texttt{INT\_MAX} - \texttt{i})}_{\beta} \;.\; \texttt{i}' = \texttt{i} + n$$

## Bounding Acceleration

- Solution: impose bound $\beta$ on *n*:

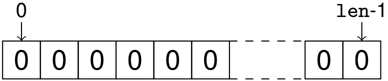
$$n < \beta(\mathtt{i})$$

  ($\beta$ depends on state)
- Example:

$$\exists n \leq \underbrace{(\mathtt{INT\_MAX} - \mathtt{i})}_{\beta(\mathtt{i})} \;.\; \mathtt{i}' = \mathtt{i} + n$$

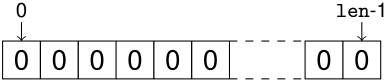
- $\beta$ can also be *implicit*

# Accelerating Arrays: Content Matters




$$\exists n \le \overbrace{(\texttt{INT\_MAX} - \texttt{i})}^{\beta(\texttt{i})} .\quad \texttt{i}' = \texttt{i} + n \wedge \texttt{i} = 0 \wedge \texttt{i}' < \texttt{len} \quad \wedge \begin{pmatrix} \forall j \le n \,.\, \texttt{buf}'[\texttt{i} + j] = \texttt{c} \quad \wedge \\ \forall j > n \,.\, \texttt{buf}'[\texttt{i} + j] = \texttt{buf}[\texttt{i} + j] \end{pmatrix}$$

# Accelerating Arrays: Content Matters

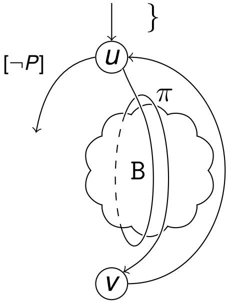



$$\exists n \le \overbrace{(\texttt{INT\_MAX} - \texttt{i})}^{\beta(\texttt{i})} \, . \quad \texttt{i}' = \texttt{i} + n \wedge \texttt{i} = 0 \wedge \texttt{i}' < \texttt{len} \quad \wedge$$
$$\begin{pmatrix} \forall j \le n \, . \, \texttt{buf}'[\texttt{i} + j] = \texttt{c} \quad \wedge \\ \forall j > n \, . \, \texttt{buf}'[\texttt{i} + j] = \texttt{buf}[\texttt{i} + j] \end{pmatrix}$$

- Bound can be conservative ⇒ *Under-Approximation*
- Details in our previous papers [CAV’13; FMSD’15]

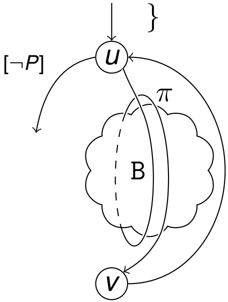
# Instrumenting Programs

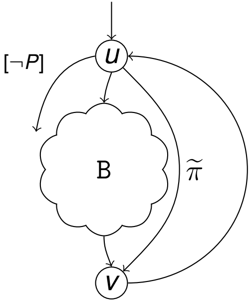
```
while (P) {
  B;
}
```



# Instrumenting Programs

```
while (P) {
  B;
}
```




```
while (P) {
  if(*) {
    π̃;
  } else {
    B;
  }
}
```

## Reachability Diameter

$$
\begin{array}{l}
\texttt{unsigned}\ \texttt{N} = \texttt{10}^6; \\
\texttt{unsigned}\ \texttt{x} = \texttt{N}, \texttt{y} = \texttt{0}; \\
\texttt{while}\ (\texttt{x} > \texttt{0})\ \{ \\
\qquad \texttt{x} = \texttt{x} - \texttt{1}; \\
\qquad \texttt{y} = \texttt{y} + \texttt{1}; \\
\} \\
\texttt{assert}\ (\texttt{y} \neq \texttt{N});
\end{array}
$$

## Reachability Diameter

$$
\begin{array}{l}
\texttt{unsigned}\ \mathtt{N} = \mathtt{10}^6; \\
\texttt{unsigned}\ \mathtt{x} = \mathtt{N}, \mathtt{y} = \mathtt{0}; \\
\texttt{while}\ (\mathtt{x} > \mathtt{0})\ \{ \\
\quad\quad \mathtt{x} = \mathtt{x} - \mathtt{1}; \\
\quad\quad \mathtt{y} = \mathtt{y} + \mathtt{1}; \\
\} \\
\texttt{assert}\ (\mathtt{y} \neq \mathtt{N});
\end{array}
$$

Reachability diameter:

- Longest shortest path between two states
- From $\mathtt{x} = \mathtt{10}^6, \mathtt{y} = \mathtt{0}$ to $\mathtt{x} = \mathtt{0}, \mathtt{y} = \mathtt{10}^6$: $10^6$ iterations

## Reducing the Reachability Diameter using Acceleration

```
unsigned N = 10^6;
unsigned x = N, y = 0;
while (x > 0) {
    x = x - 1;
    y = y + 1;
}
assert (y ≠ N);
```

## Reducing the Reachability Diameter using Acceleration

$$
\begin{array}{l}
\texttt{unsigned}\;\texttt{N} = 10^6;\\
\texttt{unsigned}\;\texttt{x} = \texttt{N}, \texttt{y} = 0;\\
\texttt{while}\;(\texttt{x} > 0)\;\{\\
\quad \texttt{x} = \texttt{x} - 1;\\
\quad \texttt{y} = \texttt{y} + 1;\\
\}\\
\texttt{assert}\;(\texttt{y} \neq \texttt{N});
\end{array}
$$

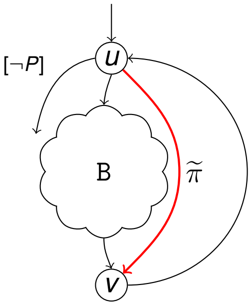
$$
\begin{array}{ll}
\left.\begin{array}{l}\texttt{unsigned}\; n = *;\\ \texttt{assume}\;(n > 0)\end{array}\right\} & \text{iteration counter}\\
\left.\texttt{assume}(\texttt{x} > 0);\right\} & \text{feasibility check}\\
\left.\begin{array}{l}\texttt{x} = \texttt{x} - n;\\ \texttt{y} = \texttt{y} + n;\end{array}\right\} & \text{acceleration}\\
\left.\texttt{assume}(\neg \text{underflow}\;(\texttt{x}));\right\} & \text{iteration bound}
\end{array}
$$

## Fail Fast, Fail Early...

$$
\begin{array}{l}
\texttt{unsigned}\ \texttt{N} = 10^6, \texttt{x} = \texttt{N}, \texttt{y} = 0; \\
\texttt{while}\ (\texttt{x} > 0)\ \{ \\
\quad \texttt{if}\ (*)\ \{ \\
\quad\quad n = *;\ \texttt{assume}\ (n > 0); \\
\quad\quad \texttt{x} = \texttt{x} - n;\ \texttt{y} = \texttt{y} + n; \\
\quad\quad \texttt{assume}\ (\neg \text{underflow}\ (\texttt{x})); \\
\quad \}\ \texttt{else}\ \{ \\
\quad\quad \texttt{x} = \texttt{x} - 1;\ \texttt{y} = \texttt{y} + 1; \\
\quad \} \\
\} \\
\texttt{assert}\ (\texttt{y} \neq \texttt{N});
\end{array}
$$

- From $\texttt{x} = 10^6, \texttt{y} = 0$ to $\texttt{x} = 0, \texttt{y} = 10^6$: 1 iteration

# Fail Fast, Fail Early...




- Reduced reachability diameter
- Shorter paths to bugs!

# Challenges in Bounded Software Model Checking

1. High unwinding depth [FMSD'15]
2. Safety proofs [FM'15]

## Proving Safety

- Why not use interpolation-based model checking?

## Proving Safety

- Why not use interpolation-based model checking?
  - Accelerated transition relation contains quantifiers
  - Insurmountable challenge for current interpolation systems

# Proving Safety

- Why not use interpolation-based model checking?
  - Accelerated transition relation contains quantifiers
  - Insurmountable challenge for current interpolation systems
- In some cases, BMC can actually prove safety!

# Unwinding-Assertions

while ($C$) { $B$; }

```
if (C) {
  B;
    if (C) {
      B;
      if (C) {
        B;

      }
    }
  }
}
```

## Unwinding-Assertions

$$\text{while } (C)\ \{\ B;\ \}$$

**if** ($C$) $\{$
  B;
    **if** ($C$) $\{$
      B;
      **if** ($C$) $\{$
        B;
        **assert** ($\neg C$);
      $\}$
    $\}$
  $\}$
$\}$

- Assertion holds if loop cannot be unwound further!

# Unwinding-Assertions

while $(C)$ $\{$ $B$; $\}$

```
if (C) {
  B;
    if (C) {
      B;
      if (C) {
        B;
        assert (¬C);
      }
    }
  }
}
```

- Assertion holds if loop cannot be unwound further!
  - more generally: no more feasible paths to extend

## Unwinding-Assertions

$$\text{while } (C)\ \{\ B;\ \}$$

$$
\begin{array}{l}
\textbf{if } (C)\ \{ \\
\quad \text{B;} \\
\quad\quad \textbf{if } (C)\ \{ \\
\quad\quad\quad \text{B;} \\
\quad\quad\quad \textbf{if } (C)\ \{ \\
\quad\quad\quad\quad \text{B;} \\
\quad\quad\quad\quad \textbf{assert } (\neg C); \\
\quad\quad\quad \} \\
\quad\quad \} \\
\quad \} \\
\}
\end{array}
$$

- Assertion holds if loop cannot be unwound further!
  - more generally: no more feasible paths to extend
- Otherwise, there is a path exceeding the bound $k$!

# Unwinding-Assertions and Accelerated Transitions

while $(\texttt{i} \leq \texttt{googol})$ $\{$ `i++;` $\}$

```
if (i ≤ googol) {
  i++;
    if (i ≤ googol) {
      i++;
      if (i ≤ googol) {
        i++;
        assert (i > googol);
      }
    }
  }
}
```

## Unwinding-Assertions and Accelerated Transitions

while $(\texttt{i} \le \texttt{googol})$ { `i++;` }

```
if (i ≤ googol) {
  i=i+n1 ;
    if (i ≤ googol) {
      i=i+n2 ;
      if (i ≤ googol) {
        i=i+n3 ;
        assert (i > googol);
      }
    }
  }
}
```

- `i=i+`$n$ subsumes `i++`
- Allows repeated and redundant execution of accelerated statement

# Example: A Safe Program

```
unsigned N = *;
unsigned x = N, y = 0;
while (x > 0) {
    x = x - 1;
    y = y + 1;
}
assert (y == N);
```

# Unwinding Safe Program with Unwinding Assertion

```
N = *;
x = N, y = 0;
if (x > 0) {
    x = x − 1; y = y + 1;
    if (x > 0) {
        x = x − 1; y = y + 1;
        if (x > 0) {
            x = x − 1;
            y = y + 1;
            assert (x ≤ 0);
        }
    }
}
assert (y == N);
```

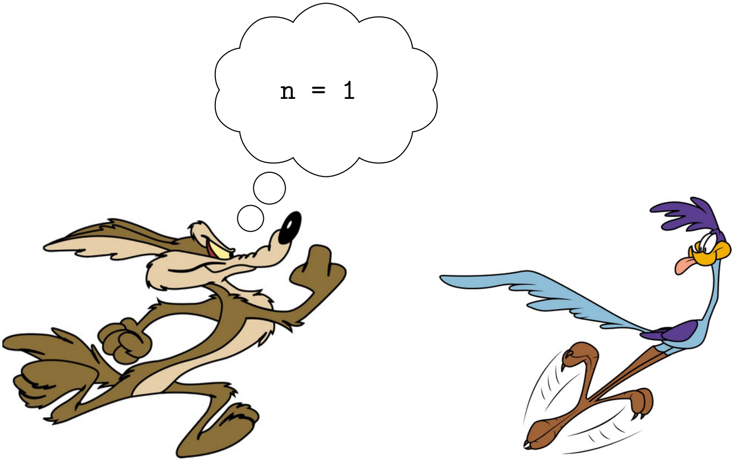
## Accelerated Safe Program

```
unsigned N = *, x = N, y = 0;
while (x > 0) {
    if (*) {
        n = *; assume (n > 0);
        x = x−n; y = y+n;
        assume (¬underflow (x));
    } else {
        x = x − 1; y = y + 1;
    }
}
assert (y == N);
```

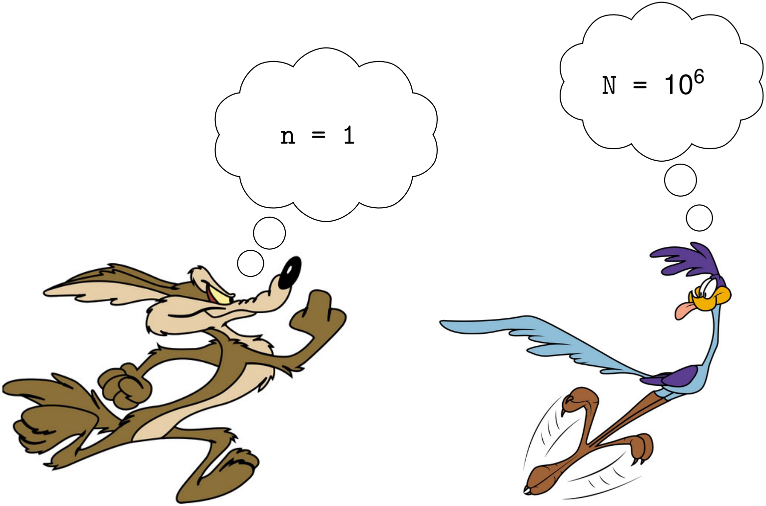
# Unwinding-Assertions and Accelerated Transitions

# Unwinding-Assertions and Accelerated Transitions

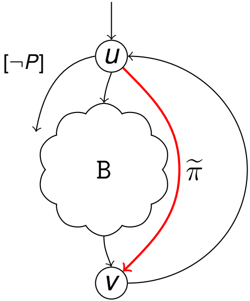
# Unwinding-Assertions and Accelerated Transitions



# Unwinding-Assertions and Accelerated Transitions

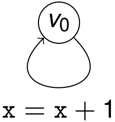


# Solution: Disallow Redundant Executions



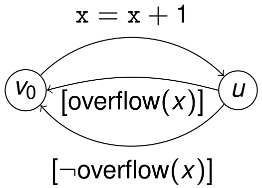

- Never take $\widetilde{\pi}$ twice in a row!

# Instrumenting Programs: Revisited



# Instrumenting Programs: Revisited

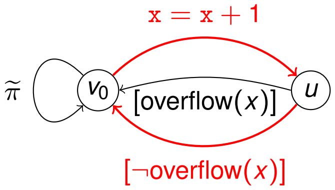
- Consider paths with and without overflow



# Instrumenting Programs: Revisited

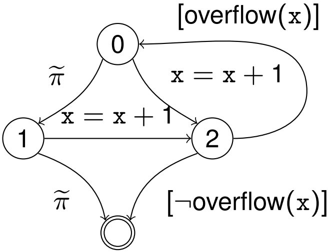
- Accelerate overflow-free path only

$$\pi \stackrel{\text{def}}{=} \mathtt{x} = \mathtt{x} + 1;\ [\neg\text{overflow}(\mathtt{x})]$$
$$\widetilde{\pi} \stackrel{\text{def}}{=} \mathtt{x} = \mathtt{x} + *;\ [\neg\text{overflow}(\mathtt{x})]$$



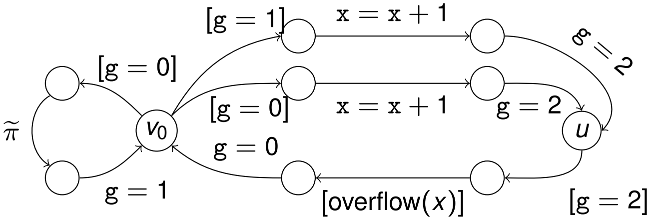
# Instrumenting Programs: Revisited

- "Trace automaton" disallows paths
  - that execute $\widetilde{\pi}$ twice in a row
  - that execute `x = x + 1` without subsequent overflow



# Instrumenting Programs: Revisited

- Instrument program:
  - $g \in \{0, 1, 2\}$ represents non-final states of trace automaton
  - edges reaching final state are suppressed



## Instrumented Example (Simplified)

```
    unsigned N = *, x = N, y = 0;
    bool g = *;
1:  while (x > 0) {
        if (*) {
            assume (¬g);
2:          n = *; x = x−n; y = y+n;
            assume (¬underflow (x));
3:          g = true;
        } else {
            x = x − 1; y = y + 1;
            assume (underflow (x));
            g = false;
        }
    }
4:  assert (y == N);
```

## Experimental Results

- CBMC with Z3 as backend (required for quantifiers)
- SVCOMP'14 (loop category) safe benchmarks:
  - 21/35 accelerated (current limitation: no nested loops)
  - 14 proven correct, including unbounded loops
- SVCOMP'14 unsafe benchmarks:
  - 18/32 accelerated
  - 12 bugs found
- Significant speedup for unsafe crafted benchmarks (factor 6)

| | | CBMC | | CBMC + Acceleration | CBMC + Acceleration + Trace Automata |
|---|---|---|---|---|---|
| | #Benchmarks | #Correct | #Benchmarks accelerated | #Correct | #Correct |
| SVCOMP safe | 35 | 14 | 21 | 2 | 14 |
| SVCOMP unsafe | 32 | 20 | 18 | 11 | 12 |
| Crafted safe | 15 | 0 | 15 | 0 | 15 |
| Crafted unsafe | 14 | 0 | 14 | 14 | 14 |

## The SUM_ARRAYS SV-COMP Benchmark

$$
\begin{array}{l}
\texttt{unsigned}\ \texttt{M} = *, \texttt{i};\\
\texttt{int}\ \texttt{a}[\texttt{M}], \texttt{b}[\texttt{M}], \texttt{c}[\texttt{M}];\\
\texttt{for}\ (\texttt{i} = \texttt{0};\ \texttt{i} < \texttt{M};\ \texttt{i} = \texttt{i} + \texttt{1})\ \{\\
\qquad \texttt{c}[\texttt{i}] = \texttt{a}[\texttt{i}] + \texttt{b}[\texttt{i}];\\
\}\\
\texttt{for}\ (\texttt{i} = \texttt{0};\ \texttt{i} < \texttt{M};\ \texttt{i} = \texttt{i} + \texttt{1})\ \{\\
\qquad \texttt{assert}\ (\texttt{c}[\texttt{i}] == \texttt{a}[\texttt{i}] + \texttt{b}[\texttt{i}]);\\
\}
\end{array}
$$

- Contains *unbounded* loops!
- Proven safe using CBMC in less than 2 seconds

## Take Home Message

- (Under-approximating) Acceleration helps finding deeper bugs
- No fix-points for safety proofs (in some cases ;-))