Proving Safety
with
Acceleration and Bounded Model Checking

Daniel Kröning, Matt Lewis, Georg Weissenbacher
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]
memset(buf, 0, len);
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

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

\[ i++ \]
Acceleration

\[ i' = i + 1 \]
\[ \exists n \in \mathbb{N}. \quad i' = i + n \]
Integers vs. Bit-Vectors

- Unsigned integers: $0 \leq i < \infty$
- Unsigned bit-vectors: $0 \leq i \leq \text{INT\_MAX}$
Integers vs. Bit-Vectors

- Unsigned integers: \(0 \leq i < \infty\)
- Unsigned bit-vectors: \(0 \leq i \leq \text{INT_MAX}\)

\[i = i + n \text{ for } n > (\text{INT_MAX} - i)\]:

(arithmetic overflow)
Does it really matter in practice?

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

  “We use the property
  \[
  \text{CHECK}(\text{init}(\text{main}()),
  \quad \text{LTL}(G \land \neg \text{signed_integer_overflow})
  \]
  
  \[
  [\ldots]
  
  \text{The results are quite unpleasant, it seems there are lots of overflow bugs in our benchmarks.}
  
  – Matthias Heizmann"
Solution: impose bound $\beta$ on $n$:

$$n < \beta$$
Bounding Acceleration

- Solution: impose bound $\beta$ on $n$:
  \[ n < \beta \]

- Example:
  \[ \exists n \leq (\text{INT_MAX} - i). \underbrace{i'}_{\beta} = i + n \]
Bounding Acceleration

- Solution: impose bound $\beta$ on $n$:
  \[ n < \beta(i) \]
  
  ($\beta$ depends on state)

- Example:
  \[ \exists n \leq (\text{INT MAX} - i). i' = i + n \]
  \[ \underbrace{\beta(i)} \]

- $\beta$ can also be *implicit*
Accelerating Arrays: Content Matters

\[
\exists n \leq (\text{INT_MAX} - i).
\beta(i) \cdot i' = i + n \land i = 0 \land i' < \text{len} \land
\begin{aligned}
\forall j \leq n. & \quad \text{buf}'[i + j] = c \land \\
\forall j > n. & \quad \text{buf}'[i + j] = \text{buf}[i + j]
\end{aligned}
\]
Accelerating Arrays: Content Matters

\[ \exists n \leq (\text{INT\_MAX} - i). \quad i' = i + n \land i = 0 \land i' < \text{len} \land \]

\[
(\forall j \leq n. \text{buf}'[i + j] = c \land \quad
(\forall j > n. \text{buf}'[i + j] = \text{buf}[i + j])
\]

- Bound can be conservative \(\Rightarrow\) Under-Approximation
- Details in our previous papers [CAV’13; FMSD’15]
Instrumenting Programs

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

\([-P]\)

\(U\)

\(\pi\)

\(\beta\)

\(V\)
Instrumenting Programs

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

while \( (P) \) {
    if(\texttt{*}) {
        \( \neg \pi \); \\
    } else {
        B;
    }
}
Reachability Diameter

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

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

Reachability diameter:
- Longest shortest path between two states
- From $x = 10^6, y = 0$ to $x = 0, y = 10^6$: $10^6$ 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

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

unsigned n = *;
assume (n > 0) \{ \text{iteration counter} \}
assume (x > 0); \{ \text{feasibility check} \}
x = x - n;
\{ \text{acceleration} \}
y = y + n;
assume (\neg \text{underflow (x)}); \{ \text{iteration bound} \}
unsigned N = 10^6, 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);

▶ From x = 10^6, y = 0 to x = 0, 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]
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
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;
      assert (¬C);
    }
  }
}

 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

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

\[\textbf{if} \ (C) \ \{\]
\[\quad B;\]
\[\quad \textbf{if} \ (C) \ \{\]
\[\quad \\quad B;\]
\[\quad \quad \textbf{if} \ (C) \ \{\]
\[\quad \\quad B;\]
\[\quad \\quad \textbf{assert} \ (\neg C);\]
\[\quad \}\]
\[\}\]
\[\]\]
\[\}\]

\[\]

\[\] ▶ Assertion holds if loop cannot be unwound further!\]
Unwinding-Assertions

\[
\text{while } (C) \{ \; B; \; \}
\]

\[
\text{if } (C) \{ \\
\;
\text{B;} \;
\text{if } (C) \{ \\
\;
\text{B;} \;
\text{if } (C) \{ \\
\;
\text{B;} \\
\text{assert } (\neg C); \;
\}
\}
\}
\}
\]

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

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

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

- 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 (i ≤ googol) { i++; }

if (i ≤ googol) {
    i++;
    if (i ≤ googol) {
        i++;
        if (i ≤ googol) {
            i++;
            assert (i > googol);
        }
    }
}
Unwinding-Assertions and Accelerated Transitions

while (i ≤ 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

```c
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

\[ \begin{align*}
N &= \ast; \\
x &= N, y = 0; \\
\text{if} \ (x > 0) \{ \\
    &\quad x = x - 1; \ y = y + 1; \\
    &\quad \text{if} \ (x > 0) \{ \\
    &\quad \quad x = x - 1; \ y = y + 1; \\
    &\quad \quad \text{if} \ (x > 0) \{ \\
    &\quad \quad \quad x = x - 1; \\
    &\quad \quad \quad y = y + 1; \\
    &\quad \quad \quad \text{assert} \ (x \leq 0); \\
    &\quad \quad \} \\
    &\quad \} \\
&\} \\
\text{assert} \ (y == N); 
\end{align*} \]
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

\[ n = 1 \]
Unwinding-Assertions and Accelerated Transitions

\[ n = 1 \]

\[ N = 10^6 \]
Solution: Disallow Redundant Executions

- Never take $\tilde{\pi}$ twice in a row!
Instrumenting Programs: Revisited

\[ v_0 \]

\[ x = x + 1 \]
Consider paths with and without overflow

\[ x = x + 1 \]

[overflow(x)]

[\neg\text{overflow}(x)]
Instrumenting Programs: Revisited

- Accelerate overflow-free path only

\[
\pi \overset{\text{def}}{=} x = x + 1; \ [\neg \text{overflow}(x)] \\
\overset{\pi}{\sim} \overset{\text{def}}{=} x = x + \ast; \ [\neg \text{overflow}(x)]
\]

\[
\begin{tikzcd}
\pi \\
\overset{\pi}{\sim} \\
\overset{\pi}{\sim}
\end{tikzcd}
\]

\[
x = x + 1
\]
Instrumenting Programs: Revisited

- “Trace automaton” disallows paths
  - that execute \( \bar{\pi} \) twice in a row
  - that execute \( x = x + 1 \) without subsequent overflow

```
x = x + 1

\[ \bar{\pi} \]
```

[overflow(x)]

```
x = x + 1

\[ \bar{\pi} \]
```

[\(\neg\)overflow(x)]

```
0 -> 1
```

```
1 -> 2
```

```
2 -> 0
```

```
0 -> 2
```

```
1 -> 0
```

```
1 -> 1
```

```
2 -> 1
```

```
2 -> 2
```

```
0 -> 1
```

```
1 -> 0
```

```
2 -> 0
```

```
0 -> 0
```

```
1 -> 1
```

```
2 -> 1
```

```
0 -> 1
```

```
1 -> 0
```

```
2 -> 0
```

```
0 -> 1
```

```
1 -> 0
```

```
2 -> 0
```

```
0 -> 1
```

```
1 -> 0
```

```
2 -> 0
```

```
0 -> 1
```

```
1 -> 0
```

```
2 -> 0
```

```
0 -> 1
```

```
1 -> 0
```

```
2 -> 0
```

```
0 -> 1
```

```
1 -> 0
```

```
2 -> 0
```

```
0 -> 1
```

```
1 -> 0
```

```
2 -> 0
```

```
0 -> 1
```

```
1 -> 0
```

```
2 -> 0
```

```
0 -> 1
```

```
1 -> 0
```

```
2 -> 0
```

```
0 -> 1
```

```
1 -> 0
```

```
2 -> 0
```

```
0 -> 1
```

```
1 -> 0
```

```
2 -> 0
```

```
0 -> 1
```

```
1 -> 0
```

```
2 -> 0
```

```
0 -> 1
```

```
1 -> 0
```

```
2 -> 0
```

```
0 -> 1
```

```
1 -> 0
```

```
2 -> 0
```

```
0 -> 1
```

```
1 -> 0
```

```
2 -> 0
```

```
0 -> 1
```

```
1 -> 0
```

```
2 -> 0
```

```
0 -> 1
```

```
1 -> 0
```

```
2 -> 0
```

```
0 -> 1
```

```
1 -> 0
```

```
2 -> 0
```

```
0 -> 1
```

```
1 -> 0
```

```
2 -> 0
```
Instrumenting Programs: Revisited

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

\[
\begin{align*}
\pi &\quad [g = 0] \\
g &\quad [g = 1] \\
v_0 &\quad x = x + 1 \\
g &\quad [g = 0] \\
x &\quad x = x + 1 \\
g &\quad g = 2 \\
u &\quad g = 2 \\
\text{overflow}(x) &\quad [g = 2]
\end{align*}
\]
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)

<table>
<thead>
<tr>
<th></th>
<th>#Benchmarks</th>
<th>#Correct</th>
<th>#Benchmarks accelerated</th>
<th>#Correct</th>
</tr>
</thead>
<tbody>
<tr>
<td>SVCOMP safe</td>
<td>35</td>
<td>14</td>
<td>21</td>
<td>2</td>
</tr>
<tr>
<td>SVCOMP unsafe</td>
<td>32</td>
<td>20</td>
<td>18</td>
<td>11</td>
</tr>
<tr>
<td>Crafted safe</td>
<td>15</td>
<td>0</td>
<td>15</td>
<td>0</td>
</tr>
<tr>
<td>Crafted unsafe</td>
<td>14</td>
<td>0</td>
<td>14</td>
<td>14</td>
</tr>
<tr>
<td>CBMC + Acceleration</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CBMC + Acceleration + Trace Automata</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The SUM ARRAYS SV-COMP Benchmark

```c
unsigned M = *, i;
int a[M], b[M], c[M];

for (i = 0; i < M; i = i + 1) {
    c[i] = a[i] + b[i];
}

for (i = 0; i < M; i = i + 1) {
    assert (c[i] == a[i] + b[i]);
}
```

- 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 ;-))