Bounded Model Checking Higher-Order Programs

Yu-Yang Lin

Nikos Tzevelekos

Queen Mary University of London

Verifying Software: Model Checking

Given a program M:

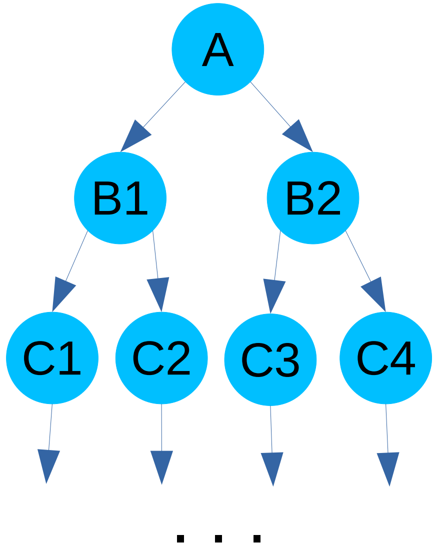
For model ϕ of M and a property α, prove $\phi \models \alpha$

- α safety defined by assertions
- ϕ is exhaustively explored
- State-space explosion limits feasibility

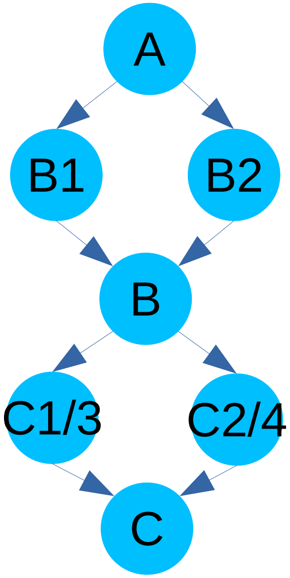
Bounded Model Checking

- Industrially successful on C-like languages
- Exchanges completeness for feasibility
- **BMC:** Bounded unwindings of a system
  - Symbolic values (free variables) instead of concrete
  - Performs state-merging after branching

Unbounded computation paths



Bounded model with state-merging



The CBMC Approach

Reduces verification problem to SAT:

1) Unwind loops k times

Loop:

```
while(B) A ;
```

Loop unwound to k=3:

```
if(B) { A;
  if(B) { A;
    if(B) { A; }
  }
}
```

2) Convert program to Static-Single Assignment (SSA) form

Normal Assignment:

```
x=x+1;
x=x+2;
y=x+3;
y=y+x;
```

Static-Single Assignment:

```
x1=x0+1;
x2=x1+2;
y1=x2+3;
y2=y1+x2;
```

3) Produce program constraints φ and assertions α

The CBMC Procedure

```
if (x > z) {
  y := x;
} else {
  y := z+1;
}
w := 2*y;

assert(w > 2*z);
```

```
if (x0 > z0) {
  y1 := x0;
} else {
  y2 := z0+1;
}
y3 := (x0 > z0)? y1 : y2;
w1 := 2*y3;

assert(w1 > 2*z0);
```

$$
\begin{aligned}
\phi := {} & (y_1 = x_0) \land \\
& (y_2 = z_0 + 1) \land \\
& (y_3 := (x_0 > z_0) ? y_1 : y_2) \land \\
& (w_1 := 2 * y_3)
\end{aligned}
$$

$$\alpha := (w1 > 2 * z0)$$

To prove $\phi \implies \alpha$ we show that $\phi \land \neg\alpha$ is not SAT

i.e. if $\exists A. A \models \phi \land \neg\alpha$ then A is a counterexample

Higher-Order Programs

We examine a higher-order language with references

- higher-order methods
- higher-order lambda abstractions
- higher-order (global) store

$$
\begin{aligned}
M ::= & \, \mathtt{assert}(M) \mid x \mid m \mid i \mid () \mid r := M \mid !r \\
& \mid \lambda x.M \mid M M \mid M \oplus M \mid \langle M, M \rangle \mid \pi_1 M \mid \pi_2 M \\
& \mid \mathtt{if}\ M\ \mathtt{then}\ M\ \mathtt{else}\ M \mid \mathtt{let}\ x = M\ \mathtt{in}\ M \\
& \mid \mathtt{letrec}\ x = \lambda x.M\ \mathtt{in}\ M \mid (\!| M |\!)
\end{aligned}
$$

$$
\frac{M : \mathtt{int}}{\mathtt{assert}(M) : \mathtt{unit}} \qquad \frac{}{() : \mathtt{unit}} \qquad \frac{}{i : \mathtt{int}} \qquad \frac{x \in \mathtt{Vars}_\theta}{x : \theta} \qquad \frac{m \in \mathtt{Meths}_{\theta,\theta'}}{m : \theta \to \theta'}
$$

$$
\frac{M : \mathtt{int} \qquad M_0, M_1 : \theta}{\mathtt{if}\ M\ \mathtt{then}\ M_1\ \mathtt{else}\ M_0 : \theta} \qquad \frac{r \in \mathtt{Refs}_\theta}{!r : \theta} \qquad \frac{r \in \mathtt{Refs}_\theta \qquad M : \theta}{r := M : \mathtt{unit}} \qquad \frac{M' : \theta \to \theta' \qquad M : \theta}{M' M : \theta'}
$$

Operational Semantics

Configurations of the form (M,R,S,k)

M: term to evaluate
R: method repository
S: store
k: call counter

- **Counter** k for nested method application

Example transition rules:

$$(E[\texttt{assert}\ (i)], R, S, k) \to (E[()], R, S, k)$$
$$(E[!r], R, S, k) \to (E[S(r)], R, S, k)$$

$$(E[\texttt{if}\ 0\ \texttt{then}\ M_1\ \texttt{else}\ M_0], R, S, k) \to (E[M_0], R, S, k)$$
$$(E[\texttt{if}\ i\ \texttt{then}\ M_1\ \texttt{else}\ M_0], R, S, k) \to (E[M_1], R, S, k) \quad (i \neq 0)$$

$$(E[mv], R, S, k) \to (E[(\!|M|\!)], R, S, k+1) \text{where } R(m) = \lambda x.M$$
$$(E[(\!|v|\!)], R, S, k) \to (E[v], R, S, k-1)$$

$$E ::= \bullet \mid \texttt{assert}(E) \mid r := E \mid E \oplus M \mid v \oplus E \mid \langle E, M \rangle \mid \langle v, E \rangle \mid \pi_j E \mid mE$$
$$\mid \texttt{let}\ x = E\ \texttt{in}\ M \mid \texttt{if}\ E\ \texttt{then}\ M\ \texttt{else}\ M \mid (\!|E|\!)$$

Nominal Defunctionalization

- **Repository** $R : \texttt{Meths} \to \texttt{Terms}$ of all method names created so far
- Higher-order methods become first-order values
- Allows reasoning about control-flow of methods passed
- Replace all arrow-type terms with names m

$$(\lambda x.M, R, S, k) \to (m, R[m \mapsto \lambda x.M], S, k)$$

$$(\texttt{letrec}\ f = \lambda x.M\ \texttt{in}\ M', R, S, k) \to (M'\{m/f\}, R[m \mapsto \lambda x.M\{m/f\}, S, k)$$

Our Approach

Bounded symbolic-state syntactical translation based on:

- *defunctionalization* using nominal techniques
- adaptation of *SSA* to higher-order values
- *points-to analysis* to deal with symbolic methods

Bound: nested method application

Returns: ground-type counterexamples

The translation:

$$[\![M, R, S]\!]_{k_0} = (\phi, \alpha, pc)$$

M,R,S as before. We add program constraints ϕ, assertions α, path condition pc, and a call bound k_0

Symbolic Method Application

```
r := if (n <= 0) then (λ x. x−1) else (λ x. x+1);
assert(!r n >= n)
```

```
let ret = if n then m1 else m2
r := ret
assert(!r n >= n)
```

```
let ret' = m1 n in
assert(ret' >= n)
```

```
let ret' = m2 n in
assert(ret' >= n)
```

which name to use when dereferencing *r*?

Points-to Analysis avoids combinatorial blow-up in names

$$(ret' < n) \land (r = m1 \Rightarrow ret' = n - 1) \land (r = m2 \Rightarrow ret' = n + 1) \land (r = ret)$$
$$\land (n <= 0 \Rightarrow ret = m1) \land (n > 0 \Rightarrow ret = m2)$$

(SAT with n=0)

Our BMC Procedure

- Keep track of bound with flag variable inil
- When bound is breached, add assertion $(pc \implies \texttt{inil})$

BMC Procedure:

1) $[\![M, R, S]\!]_{k_0} = (\phi, \alpha, pc)$
2) Check for assertion violation: $\phi \wedge \texttt{inil} \wedge \neg\alpha$

    if SAT, error found. Otherwise:

3) Check for breached bound: $\phi \wedge \neg\texttt{inil} \wedge \neg\alpha$

    if UNSAT, program is fully verified.
    Otherwise, increase bound and repeat

Soundness and Correctness

- Soundness (no false positives):
  - $\phi \land \texttt{inil} \land \neg\alpha$ is SAT iff an assertion violation is reachable

    Otherwise:
  - $\phi \land \neg\texttt{inil} \land \neg\alpha$ is SAT iff the bound is reachable

- Correctness:

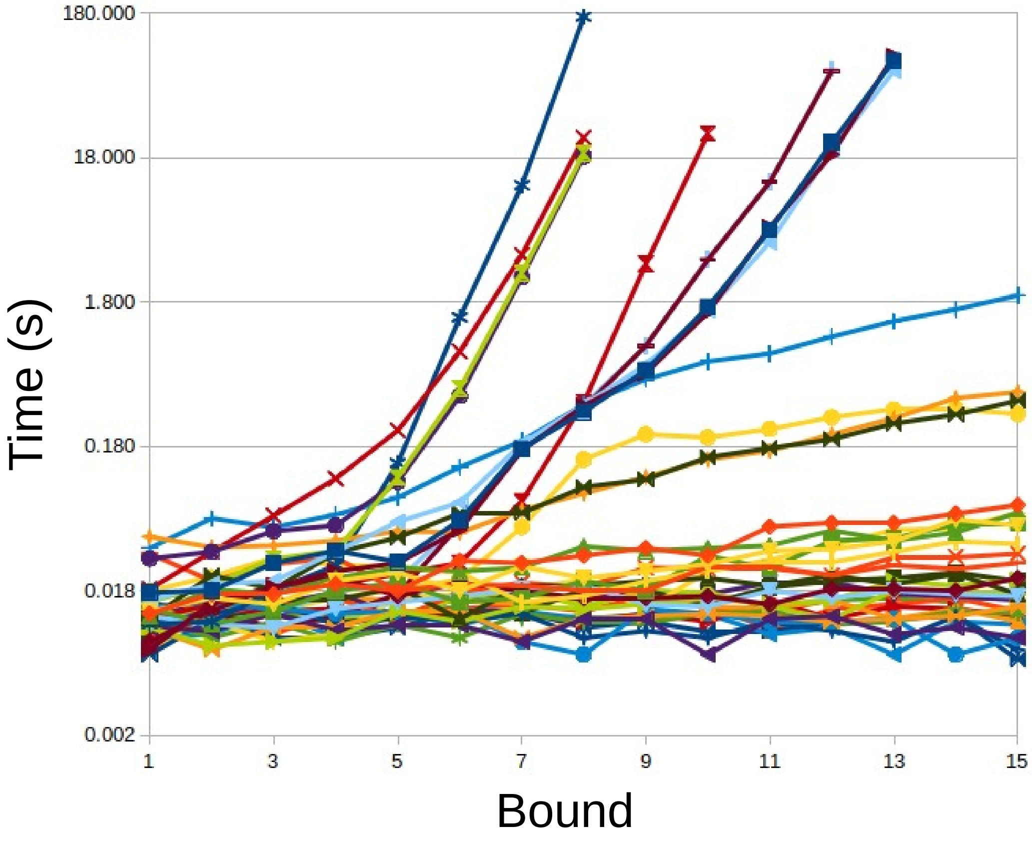
  The translation captures the operational semantics

$$(M, R, S, k) \to (\chi, R', S', k') \implies \exists\sigma \models \phi \land (ret = \chi)$$

Implementation: BMC-2

Tests on 40 programs, including ones from MoCHi benchmark

MoCHi (<50 LoC) + larger programs (100 to 400 LoC)



https://github.com/LaifsV1/BMC-2

Comparison: MoCHi

Kobayashi, Sato, and Unno. PLDI 2011

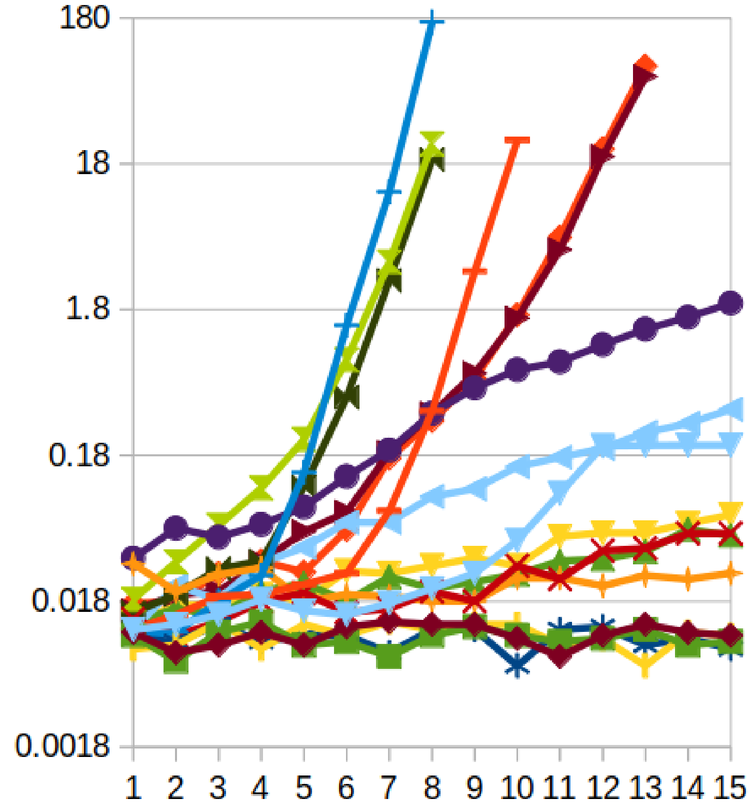
- Model Checking tool for HO Programs via CEGAR and HORS

MoCHi:

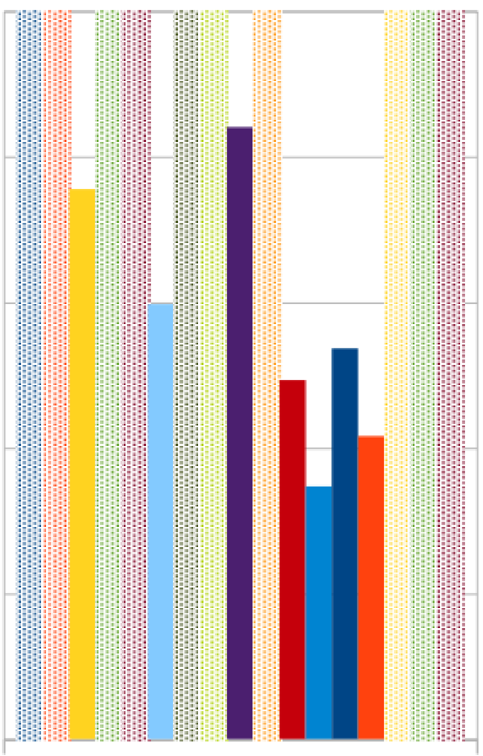
- Able to prove full correctness

BMC-2:

- Less affected by program size
- Less affected by specific program features
- Supports general references
- Found all errors soundly
- Inconclusive if no errors found



Time (s) vs Bound for **BMC-2**



Time (s) for **MoCHi**

Dotted Area: Timeout or Crash

Comparison: Rosette

Torlak and Bodik. PLDI 2014

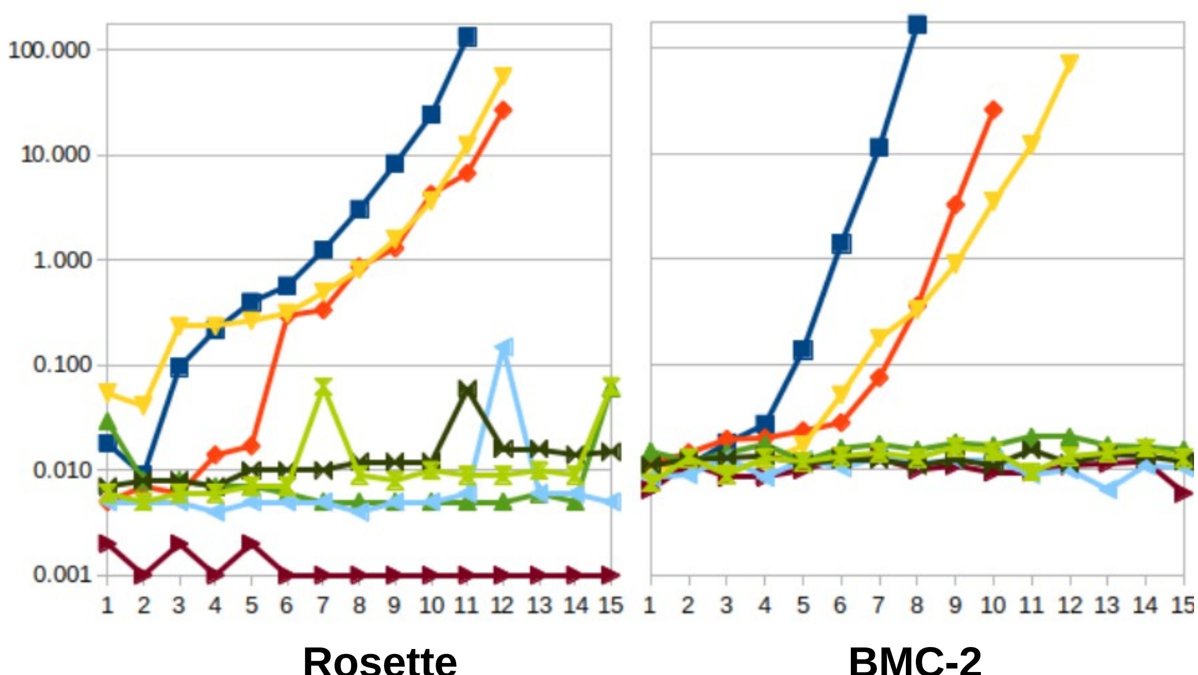
- Solver-aided language for Racket
- Built-in efficient Symbolic Execution (SE) with type-driven state merging
- We implemented our bounding semantics in Rosette to compare BSE to BMC

Rosette:

- Better optimised for diverging paths

BMC-2:

- Alternative to SE-based tools
- Faster for low bounds
- Faster compilation than SE
- Similar behaviour, but BMC-2 is not optimised
  - Inefficient state merging
  - No concretisation



Time (s) vs. Bound

Demo: Bug-Finding

Consider program mc91-e from the MoCHi benchmark:

```
let rec mc91 x =
  if x > 100 then
      x - 10
  else
      mc91 (mc91 (x + 11))
let main n = assert (mc91 n = 91)
```

It is supposed to always return 91, but does it?

Demo: Total Correctness

Consider the following program:

```
let r = ref 0
let f x = (r := (x-1); !r)
let g x = (r := (x+1); !r)
let main n = r:=n; assert (f(g n) == n)
```

The program has a finite-depth computation tree. Can it be totally verified?

Conclusions and Future Work

- Alternative to Bounded Symbolic Execution for HO programs
- BMC has theoretical advantages to SE (compilation, memory)
- Results are preliminary: technology is not fully optimised
- Approach could be implemented into existing tools like CBMC