Language-Independent Semantics-Based Compilation and Bounded Model Checking

FORMAL SYSTEMS LABORATORY

(LED BY PROF. GRIGORE ROSU)

EVERETT HILDENBRANDT, DAEJUN PARK, LUCAS PEÑA, COSMIN RADOI, MANASVI SAXENA

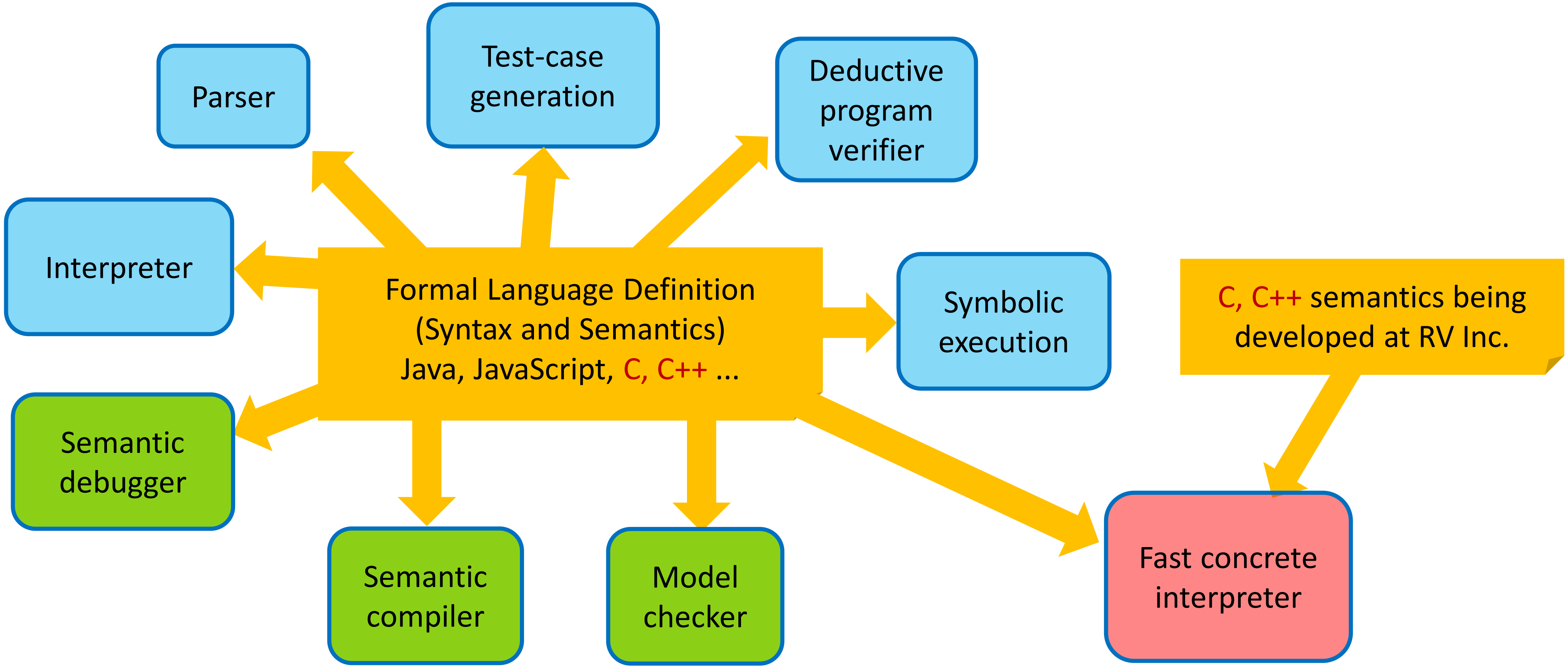
Project Overview

Objective: implement two formal analysis tools for the K semantic framework

- Semantics-based compiler – to achieve much faster formal execution of programs, by statically summarizing the behavior of the program, using the semantics of the programming language
- Bounded model checker – to execute a program systematically, on all paths, up to a given depth/bound, in order to detect errors or “unexpected” behaviors

Extends previous project, which focused mainly on semantics-based execution in order to detect undefined behaviors – that technology is now being commercialized by RV Inc.

𝕂 Framework



Parser

Test-case generation

Deductive program verifier

Interpreter

Formal Language Definition
(Syntax and Semantics)
Java, JavaScript, C, C++ ...

Symbolic execution

C, C++ semantics being developed at RV Inc.

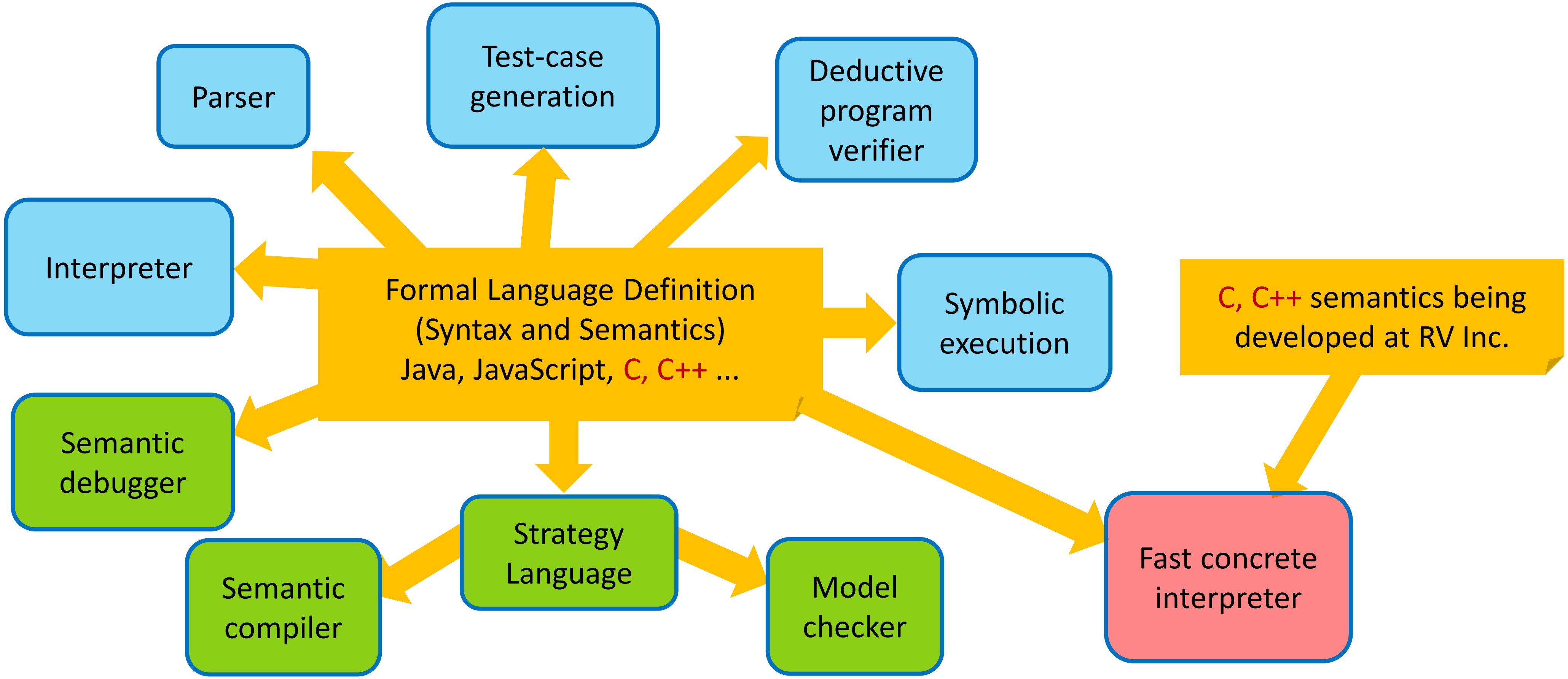
Semantic debugger

Semantic compiler

Model checker

Fast concrete interpreter

𝕂 Framework + Strategy Language



Formal Language Definition
(Syntax and Semantics)
Java, JavaScript, C, C++ ...

Parser

Test-case generation

Deductive program verifier

Interpreter

Symbolic execution

Semantic debugger

Strategy Language

Semantic compiler

Model checker

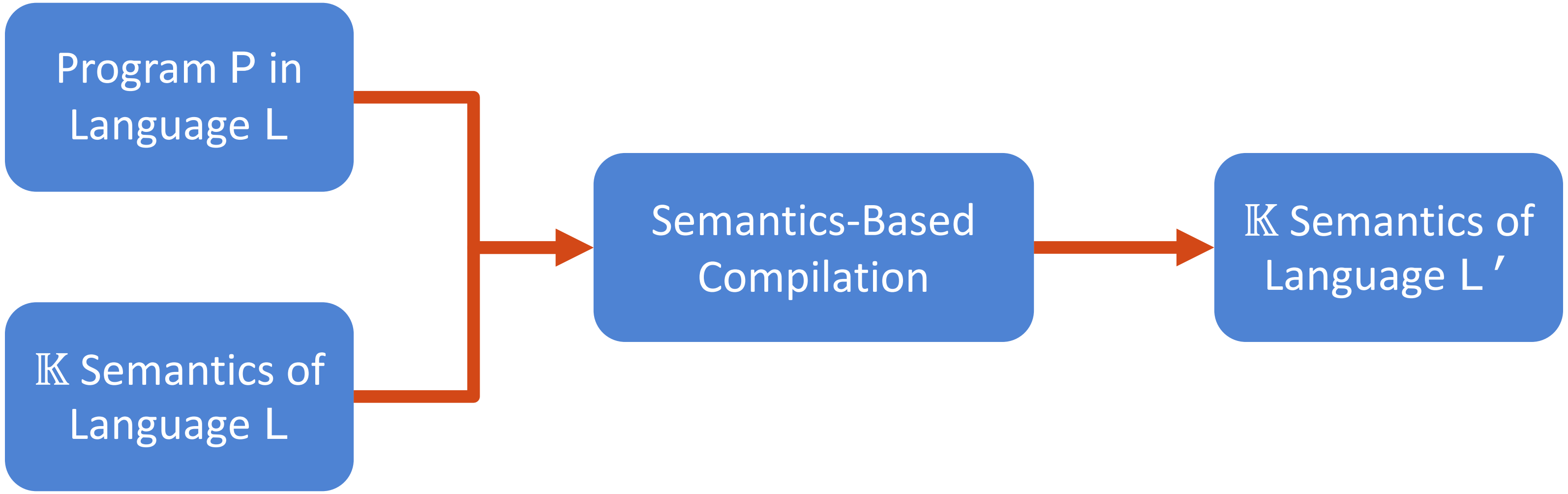
Fast concrete interpreter

C, C++ semantics being developed at RV Inc.

Semantics-Based Compilation (SBC)

Problem: Given a program P in language L, generate language L′ such that execution of program P in languages L and L′ is equivalent.

Goal: Language L′ should be “as simple as possible”, only capturing exactly the dynamics of L necessary to execute program P.



Program P in Language L

𝕂 Semantics of Language L

Semantics-Based Compilation

𝕂 Semantics of Language L′

SBC Algorithm

One step of compilation:

- Check if current state has already been compiled or execution is finished (`subsumed? or stuck?`)
- Symbolically execute program until abstraction point is reached (`step until cut-point?`)
- Summarize previous statements into one rewrite rule (`begin-rule ... end-rule`)
- Abstract state to ensure completeness

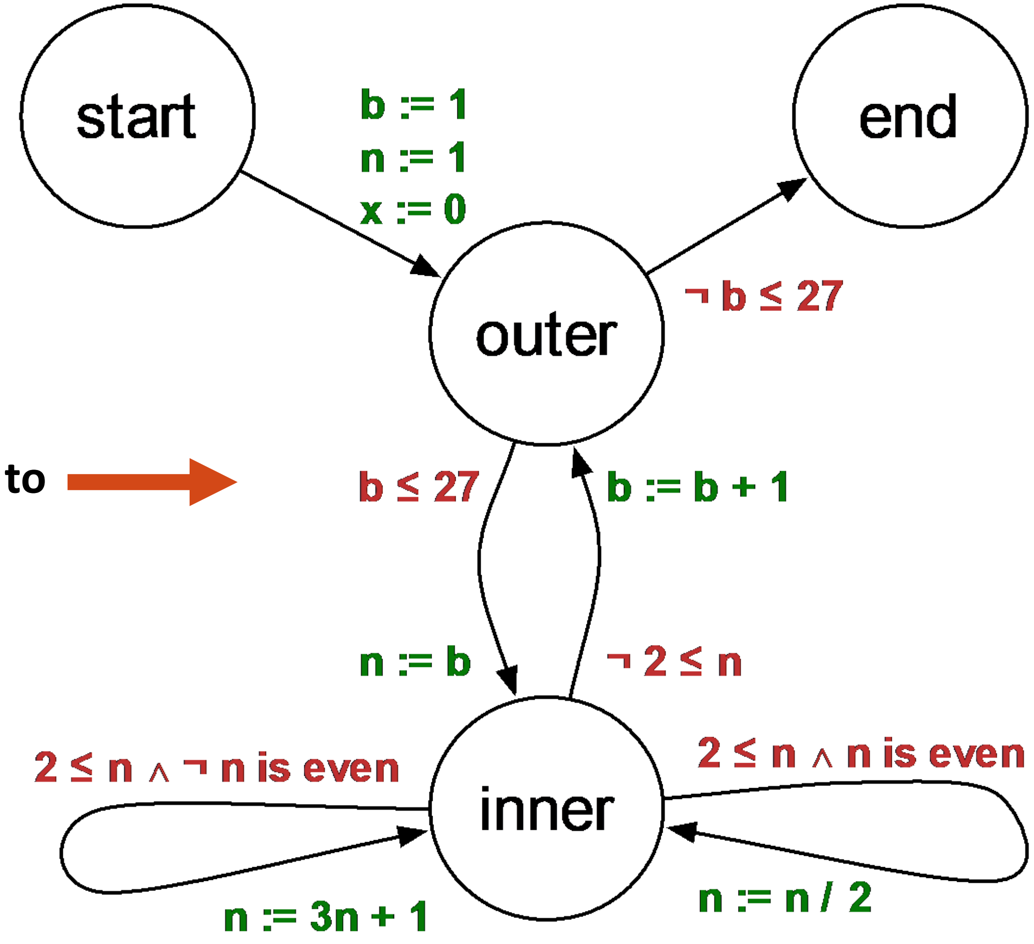
```
if (subsumed? or stuck?)
then skip ;
else begin-rule ;
     step ;
     step until cut-point? ;
     end-rule ;
     abstract ;
```

Each line of code is executed **once** (often much faster than full execution of program).

Output of the algorithm is a semantics-preserving $\mathbb{K}$ definition (can be used in further analysis).

```
// start
int b , n , x ;
b = 1 ; n = 1 ; x = 0 ;

// outer
while (b <= 27) {
  n = b ;

  // inner
  while (2 <= n) {
    if (n <= ((n / 2) * 2)) {
      n = n / 2 ;
    } else {
      n = (3 * n) + 1 ;
    }
    x = x + 1 ;
  }
  b = b + 1 ;
}
// end
```

compiles to



SBC Benchmarking

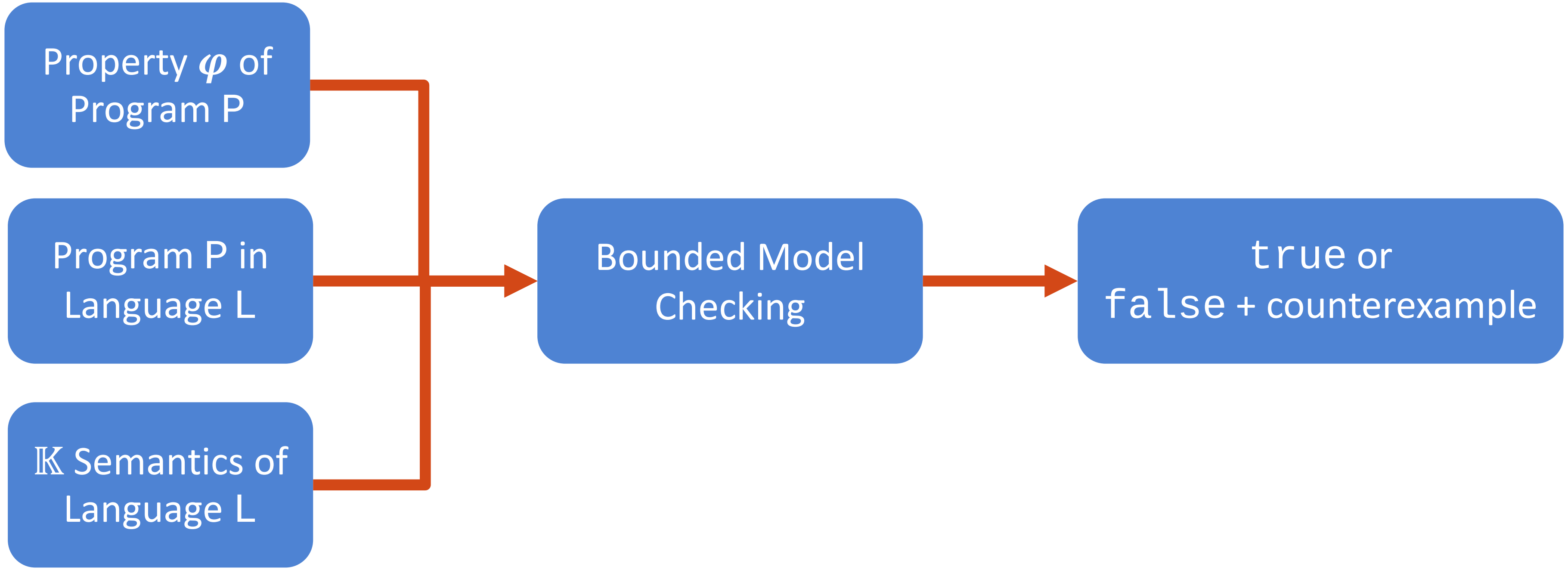
Program	Original Time (s)	Compiled Time (s)	Speedup
sum.imp	70.6	7.3	9.7
collatz.imp	34.5	2.7	12.8
collatz-all.imp	77.4	5.7	13.6
krazy-loop.imp	67.6	3.3	20.5

These numbers were gathered using concrete execution in the strategyharness

Normal $\mathbb{K}$ execution is uniformly ~10x faster

Bounded Model Checking (BMC)

Problem: Given a program P, a language L, a property φ, and a bound N, output `true` if φ is not violated within N steps, and output `false` and the violating trace otherwise.



Property φ of Program P

Program P in Language L

$\mathbb{K}$ Semantics of Language L

Bounded Model Checking

`true` or `false` + counterexample

BMC Algorithm

Bounded Model Checking:

- `record` appends current state to the accumulated trace
- `step` takes one (user-defined) step of symbolic execution
- `bmc-result` checks whether the property P holds at the current state, and outputs the current trace if it does not

```
record ;
while N P (step ; record) ;
bmc-result P ;
```

This algorithm is useful for undefinedness checking, checking common programming errors (division by zero, off by one), etc.

```
// div-good.imp
int x , y ;
x = 3 ; y = 100 ;

while (0 < x) {
  y = y / x ;
  x = x - 1 ;
}

// div-bad.imp
int x , y ;
x = 3 ; y = 100 ;

while (0 <= x) {
  y = y / x ;
  x = x - 1 ;
}
```

```
$> bmc 100 "not div-zero-error?" div-good.imp
#bmc-result #true in 32 steps ...
```

`div-good.imp` does not exhibit a division by zero error.

`div-bad.imp` does exhibit a division by zero error at 35 steps. When the bound is decreased to 34, the property is not violated but the program is about to divide by zero.

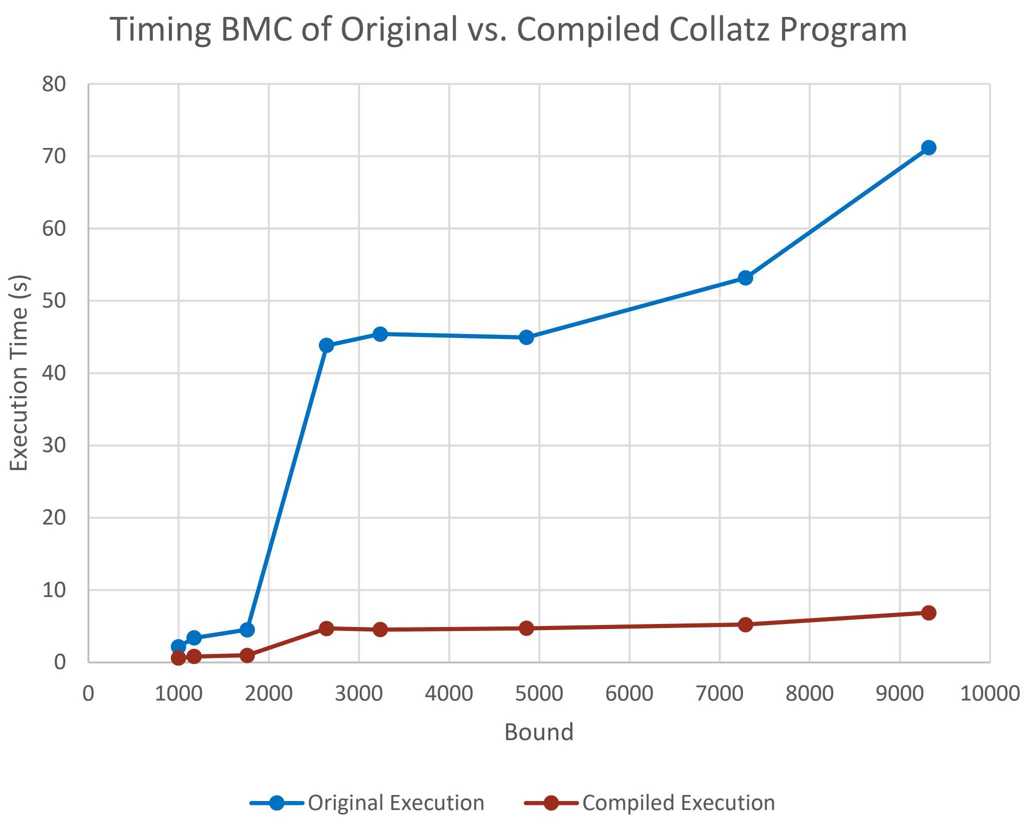
```
$> bmc 100 "not div-zero-error?" div-bad.imp
#bimc-result #false in 35 steps ...

$> bmc 34 "not div-zero-error?" div-bad.imp
<s> #bmc-result #true in 34 steps ... </s>
<k> ( 16 / 0 ) ... </k>
```

Accelerating BMC with SBC

Perform BMC on output of SBC instead of original program

- Avoid many steps of symbolic execution
- Roughly 10x speedup on Collatz program
- Higher density of straight-line code will yield more speedup

Since SBC produces a correct 𝕂 definition, it can be used to speedup many other processes such as concrete execution and program verification.



Future Extensions

Run analysis tools using faster OCaml backend developed at RV Inc

- Ocaml backend about 10,000x faster than the current Java/Scala backend, but does not support symbolic execution yet

Develop more language-independent analysis tools using this strategy-based technique

- Runtime verification of programs, combined with SBC and BMC
- Static analysis of programs, using symbolic execution
- Full program verification using deductive reasoning and invariant inference, plus SBC and BMC

Demonstrate scalability on larger languages

- C, C++, Java, JavaScript: we have K semantics for them, but in v3.6; not yet updated to work with c4.0
- Add optimizations to BMC
- Transfer formal analysis technology in commercial-grade tools at RV Inc.