Verified ucore: not a final report

Zhilei Han, Xingyu Xie

CST, Tsinghua University

May 2020

Contents

Contents

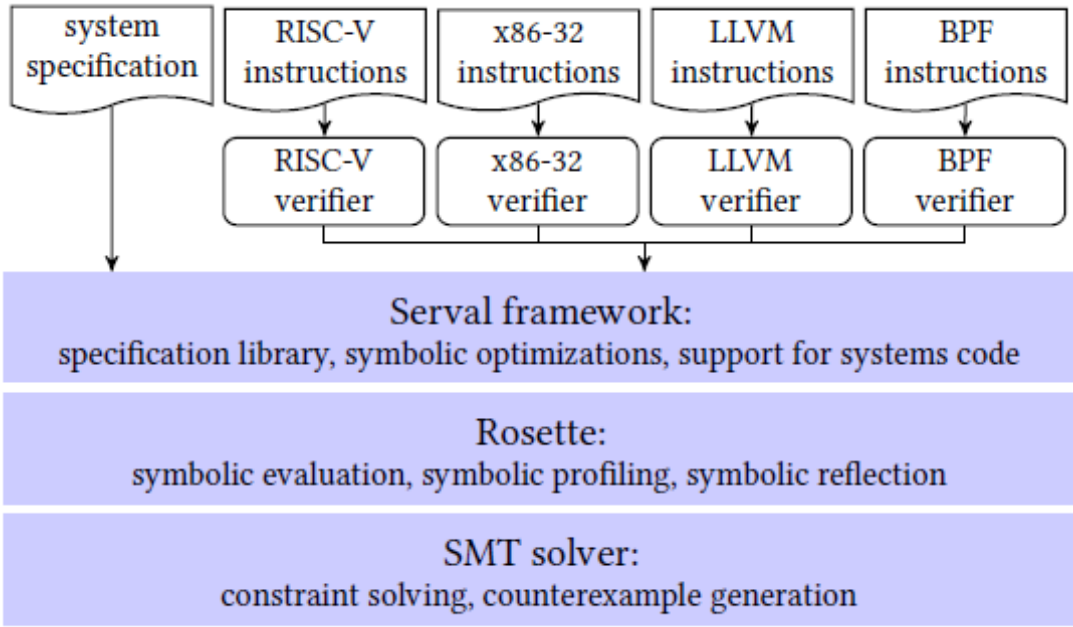
Formal Verification

Formal verification is to formally check the code is *correctness*, that is, implementation satisfies the specification, under some semantics.

```
// An example of Dafny
method Sort(a: int, b: int, c: int)
        returns (x: int, y: int, z: int)
    ensures x <= y <= z
{
    x, y, z := a, b, c;
    if z < y { y, z := z, y; }
    if y < x { x, y := y, x; }
    if z < y { y, z := z, y; }
}
```

Serval

A lightweight operating system verification framework, proposed in SOSP'19[1].



[1] Scaling symbolic evaluation for automated verification of systems code with Serval. Luke Nelson et al.

SMT Solver

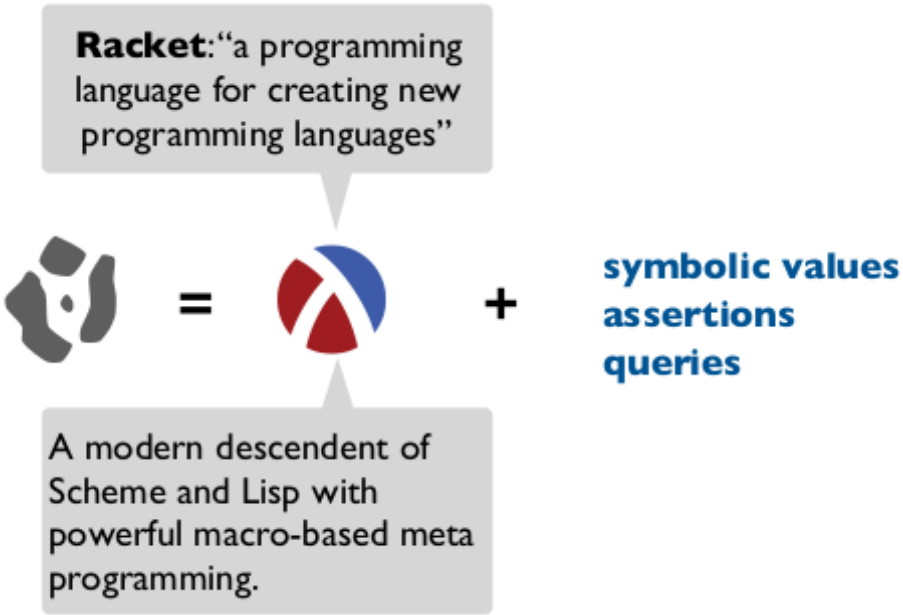
Satisfiability Modulo Theories Solver (e.g. Z3): a solver to solve some combination of first-order background theories.

```
; An example of Z3
(declare-const a Int)
(declare-const b Int)
(declare-const c Int)
(declare-const d Real)
(declare-const e Real)
(assert (> a (+ b 2)))
(assert (= a (+ (* 2 c) 10)))
(assert (<= (+ c b) 1000))
(assert (>= d e))
(check-sat)
(get-model)
```

Rosette

Rosette: a solver-aided programming system with two components

- A programming language that extends a sub core of Racket
- A symbolic virtual machine



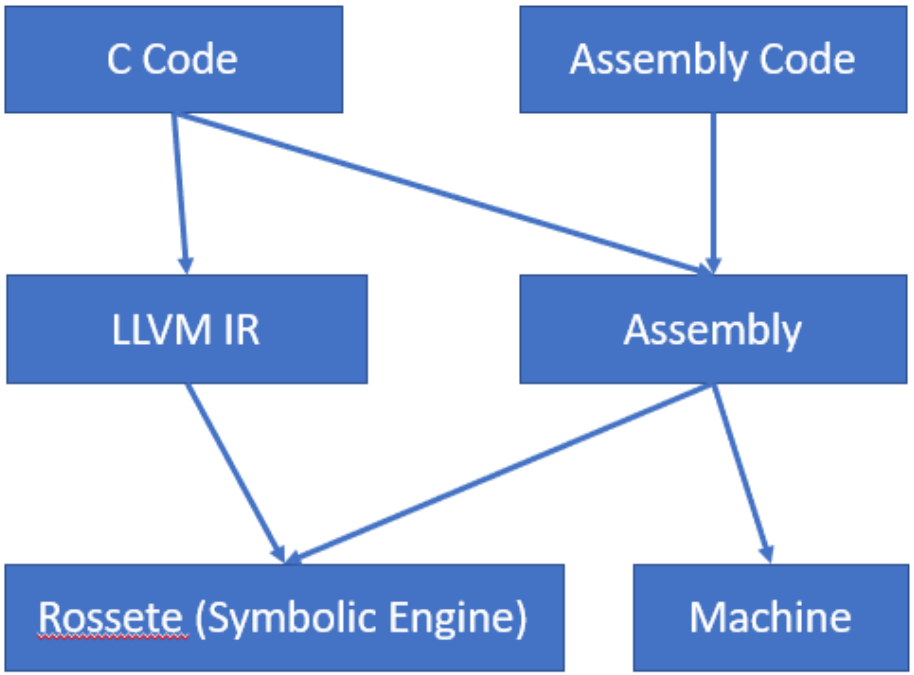
Symbolic Execution and Rosette

Symbolic execution: concrete value => symbolic value

```
(define (poly x)
    (+ (* x x x x) (* 6 x x x) (* 11 x x) (* 6 x)))
(define (factored x)
    (* x (+ x 1) (+ x 2) (+ x 2)))
(define (same p f x)
    (assert (= (p x) (f x))))
(define-symbolic i integer?)
(define cex (verify (same poly factored i)))
```

Serval framework

Serval ships a RISC-V verifier, x86-32 verifier, LLVM verifier, BPF verifier. Each is a symbolic interpreter.
Serval also provides a verification library for common verification work and specification.



Verification Paradigm

State machine refinement: verify the following two sentences:

$$\forall c, f. RI(c) \rightarrow RI(f_{impl}(c))$$

$$\forall c, s, f. \mathrm{RI}(c) \wedge AF(c) = s \rightarrow \mathrm{AF}(f_{\mathrm{impl}}(c)) = f_{\mathrm{spec}}(s)$$

where c is concrete state, s is specification state, f is transition function, RI stands for assistant representaion invariants, AF stands for abstraction function.
On the specification state, we can further more prove intended safety property: undefined behavior, overflow checking, noninterference...

Contents

Timer

The verification of lab1 involves validation of timer interrupt handler.
What it looks like:

```
case IRQ_S_TIMER:
    clock_set_next_event();
    if (++ticks % TICK_NUM == 0) {
        print_ticks();
    }
    break;
```

We don't care about console output or intermission length, which is meaningless for functional correctness of OS. So what we need to track is the update of 'ticks'.

Timer

The spec we wrote:

```
(struct state (tick)
 #:transparent
 #:mutable)

(define (intrp-timer st)
  (set-state-tick! st
    (bvadd (state-tick st) (bv 1 64))))
```

Since it's an interrupt, we use the riscv assembly as input. By emulating a riscv hardware and perform symbolic execution on the codes, we verify the correctness of implementation against this spec.

PMM:Programmer's View

Now consider how to verify physical memory management. For a C programmer, physical pages are an array of struct 'Page' residing in memory. Thus, he would use the following fragment to allocate n consecutive free pages:

```
for (size_t p = 0; p < NPAGE; p ++)
    if (PageReserved(p) || PageAllocated(p)) {
        first_usable = p + 1;
    }
    else {
        if (p - first_usable + 1 == n) {
            page = first_usable;
            break; // found! 'page' is allocated!
        }
    }
```

PMM:Programmer's View

This algorithm is efficient and elegant, but it does not necessarily entail correctness.
To prove this algorithm is actually first-fit algorithm by hand is somewhat complicated.

PMM:Formalization

Suppose there are n pages, and page can be 'reserved', 'allocated' or 'free', which is indicated by a 64bit flag. Let's call the function over B to B a 'pagedb', where B is the sort of 64bit vector.
Given pagedb f, the updated database g for indice i and value v is defined as $g = \lambda x.ite(x = i)v(fx)$. The updated database h for predicate p and update function u is defined as $h = \lambda x.ite(px)(u(fx))(fx)$
Then, the update of a database f with function u in range [a,b) is the updated databse h for predicate $\lambda x.(a <= x)\&\&(x < b)$ and update function u.

PMM:Formalization

Given pagedb f, a first-fit allocation algorithm finds the first block with at least m consecutive free memory. The routine is encoded as a recursive function in Rosette:

```
(define (find-free-pages s num)
  (define (find-free-accumulate lst acc ans)
    (cond
      [(bveq num acc) ans]
      [(null? lst) #f]
      [(page-available? s (car lst))
       (find-free-accumulate
         (cdr lst) (bvadd1 acc)
         (if (bveq acc (bv 0 64)) (car lst) ans))]
      [else (find-free-accumulate (cdr lst) (bv 0 64)
  (define indexl (map bv64 (range constant:NPAGE)))
  (find-free-accumulate indexl (bv 0 64) (bv 0 64)))
```

PMM:Formalization

Now, if the routine above returns an index, we just update the database in range [index, index + m) with function : $\lambda x.'allocated'$.
The formalization and Rosette specification looks quite different from the C code, but they turn out to be the same, given the following invariant: The number of free pages in the database is in the range [0, n].