Serval

Integrating an existing symbolic verification framework with Rust

Filippo Costa

21.12.2022

The project

- Familiarize with Serval and the surrounding environment
- Try to verify simple programs in Rust
- Integrate it with Rust and simplify the verification process

What is Serval?

A quick overview on the framework

What does it mean to symbolically verify a program?

Serval

- Formal verification framework
- Written in Racket, syntax similar to Lisp
- Provides different front ends to verify:
  - Architectures: RISC-V, x86-32
  - LLVM-IR

Concrete and Symbolic values …

- Concrete values
  - They behave as usual program variables. They have a specific value that may change during the time
- Symbolic values
  - They don't have a unique value, it depends on the execution up to that point and on the values of the other variables

```
fn example(a: u32, b: u32) → u32 {
  let mut c = a;
  if a%2==0 {
      c = 1;
  }

  if b<10 {
      c = b+1;
  }

  return c
}
```

```
assume(b==10)
```

```
c = if a%2 == 0
        then 1
        else a
```

```
assert(c==a || c==1) ✓
```

… lead to different kinds of verification

- With concrete values
    - The evaluation of a program is like a unit test. The program receives an input, it returns an output
- With symbolic values
    - All possible execution paths are evaluated at once
    - During evaluation, conditions are added to the values, to represent their content
    - At the end, all the collected information is used to construct the result
    - Example:
        1. The developer writes assumptions on input values and assertions on output values
        2. The code is executed symbolically
        3. The assertions are checked to hold with respect to the given assumptions and to the execution

Verification infrastructure

My Work

Integrate Serval with Rust

Serval

Framework to map code to Rosette and test/prove it.

Rosette

Extends Racket with constructs useful for verification, symbolic variables and the logic around them

Z3Prover

Theorem Prover

Constraint system solver

Widely used in academy and industry

Issues

- Serval is not documented to be concretely used
  - Substantial part of the coming presentation is based on «reverse engineer» the framework, the examples and the tests.
- It focuses on verifying small security monitors written in C for RISC-V
  - It is relatively complex to build a complete x86-64 verifier upon it
    - It is necessary to write an interpreter for all the needed instructions and modelize correctly the entire low level architecture

First Phase

- Understand how simple programs can be verified
- Identify what kind of program is possible to verify

Idea

- Rust uses LLVM as compiler
- Serval can verify LLVM-IR language
- Solution: compile Rust to LLVM-IR and verify this
  - Advantage: verify code independently from underlying architecture
  - Drawback: no support for architecture-specific instructions

Example: Verification Workflow

Write the code

Obtain LLVM IR: “language independent” representation of code

Serval parses obtained IR to translate it to a “pseudo assembly” representation as a sequence of Racket instructions

```rust
fn test(x: u32, y: u32) -> u32{
  let r: u32 = x + y;
  return r;
}
```

File: source.rs

Example: Verification Workflow

Write the code

Obtain LLVM IR: “language independent” representation of code

Serval parses obtained IR to translate it to a “pseudo assembly” representation as a sequence of Racket instructions

```
define i32 @test(i32 %x, i32 %y)
unnamed_addr #0 {
start:
  %r = add i32 %x, %y
  ret i32 %r
}
```

File: source.ll

Example: Verification Workflow

Write the code

Obtain LLVM IR: “language independent” representation of code

Serval parses obtained IR to translate it to a “pseudo assembly” representation as a sequence of Racket instructions

```
(define (@test %x %y)
; %start
  (define-label (%start) #:merge #f
(set! %r (add %x %y))
(ret %r))

  (define-value %r)
  (enter! %start))
```

File: generated.source.rkt

Example: Verification Workflow

Write the lemma to verify, as a Racket function

Integrate it in the testing framework

Run the test and wait for a response

```
(define (call-sum)
  (define-symbolic a (bv 32))
  (define-symbolic b (bv 32))
  (define r (@test a b))
  (assert (bveq r (bvadd b a))))
```

File: sumrust.rkt

Example: Verification Workflow

Write the lemma to verify, as a Racket function

Integrate it in the testing framework

Run the test and wait for a response

```
(define rust-tests
  (test-suite+
   "Tests for rust functions"
   (parameterize ([llvm:current-machine
(llvm:make-machine)])
  (test-case+ "sum" (check-function0 call-
sum))
)))
```

File: sumrust.rkt

Example: Verification Workflow

Write the lemma to verify, as a Racket function

Integrate it in the testing framework

Run the test and wait for a response

```
$ raco test --table sumrust.rkt
raco test: 1 (submod "sumrust.rkt" test)
Tests for rust functions
[ RUN   ] "sum"
[  OK   ] "sum" (3ms cpu) (3ms real) (6 terms)
1 success(es) 0 failure(s) 0 error(s) 1 test(s) run
cpu time: 3 real time: 3 gc time: 0
0
  1 sumrust.rkt
1 test passed
```

In a terminal

Issues with the base approach

- Source code and verification code are disjoint in their structure
  - 2 files should be manually edited, and a bunch of commands has to be executed in a terminal
  - Requires the programmer to be able to write code in two completely different languages
  - Requires specialized IDE to support the lemma development
- Even if Serval declares to be «automated», all the automation has to come from the programmer
- Syntax is verbose, in particular regarding access to memory

Second Phase

- **Integrate the workflow in a single command**
- **Express lemma directly in Rust source code**

Lemmas in Rust - Focus

- At the end, lemmas must be a Racket program to be provable
- They should be as similar as possible to Rust code
- They should be near to proved code, for convenience

Lemmas in Rust

- Expressed as macros
- Parsed using Syn, a library built to parse Rust programs
- Only a subset of Rust is available to the programmer in lemmas
  - The rest of the code can be arbitrary

What can be expressed?

- Variables and values
- Arithmetic and boolean expressions
- Function calls
- Memory access (arrays and structs)
- `if-else`, ranged `loops`

Example: The new syntax

Before

```
fn test(x: u32, y: u32) -> u32{
  let r: u32 = x + y;
  return r;
}
```

File: source.rs

```
(define (call-sum)
  (define-symbolic a (bv 32))
  (define-symbolic b (bv 32))   (define r (@test a b))
  (assert (bveq r (bvadd b a))))

(define rust-tests
  (test-suite+
   "Tests for rust functions"
   (parameterize ([llvm:current-machine (llvm:make-machine)])
  (test-case+ "sum" (check-function0 call-sum))
)))
```

File: sumrust.rkt

After

```
#[verify({
  let a: i32;
  let b: i32;

  *assert(test(a, b) == (a+b));
})]
fn test(x: u32, y: u32) -> u32{
  let r: u32 = x + y;
  return r;
}
```

File: source.rs

Example: The new syntax

```
#[verify({
  let x: u32;
  let y: u32;

  let result = equality(x, y);
  if x == y {
    *assert(result==1u8);
  } else {
    *assert(result==0u8);
  }

})]
fn equality(x: u32, y: u32) -> u8{
  if x==y || (x==0&&y==1) {
    return 1;
  }
  return 0;
}
```

```
#[verify({
    for i in 0..4 {
        *println(array_to_print[*bv(i, 64l)]);
    }
})]
```

```
raco test: (submod "gen.equality.rkt" test)
Tests equality
[ RUN      ] "equality"
Failed assertions:
--------------------
Tests equality > equality
FAILURE
name:       check-unsat?
location:   /home/filippo/serval/serval/lib/unittest.rkt:46:13
params:
  '((model
 [x (bv #x00000000 32)]
 [y (bv #x00000001 32)]))
--------------------
0 success(es) 1 failure(s) 0 error(s) 1 test(s) run
cpu time: 67 real time: 503 gc time: 5
```

Limitations

- Size of numeric literals must be explicitly written
    - Almost all variables in Serval/Rosette are bit vectors, of predefined and fixed size
- Rosette provides different operators for signed and unsigned numbers
    - But there is no elegant way to express it in a Rust-like syntax
- No semantic and type checking at translation time

Problems arose

- Serval is a research artifact for a paper
  - Documentation is lacking, even if there are some examples and documents
  - It is not actively maintained anymore
- Not every LLVM-IR is supported by the interpreter
  - Rust code needs to be carefully written, avoiding particular constructs
    - Using range loops inside the code generates boilerplate code with exception handling («`landingpad`», «`resume`» instructions) that fails to be verified
  - Compatibility issues start to show: LLVM-IR bytecode generated by Rust Nightly contains a new type of pointer that always breaks Serval

Thank you!

There is a lot of work that must be done to have a complete verification environment, but it is already possible to work with pretty complex systems.

Some of the issues come from Serval, while others are added by the integration, but the final capabilities simplify a lot the developer experience.