Owi: cross-language, multi-core, multi-solver symbolic execution

Léo …, Pierre Chambart, Arthur Carcano @ OCamlPro

Filipe Marques, José Fragoso Santos @ University of Lisbon

… Andrès, Jean-Christophe Filliâtre @ Université Paris-Saclay

With contributions from Dario Pinto, Eric Patrizio, Frederico Ramos, Olivier Pierre, Zhicheng Hui, Vasu Singh, Simon Ser, Neha Chriss, Hichem Rami Ait El Hara, Basile Clément, Saïd Zuhair, Émilien Lemaire, Félix Loyau-Kahn, Nathanaëlle Courant, Gabriel Scherer

12th of June 2025 @ Dagstuhl

Dagstuhl Mars 2023

Presented Wasocaml, an OCaml to Wasm compiler, and Owi was briefly mentionned.

Dagstuhl Mars 2023

What about making a symbolic interpreter with Owi?

— José Fragoso Santos

Riddle

C:

```c
float dot_product(float x[2], float y[2]) {
    return (x[0]*y[0] + x[1]*y[1]);
}
```

(idiomatic) Rust:

```rust
fn dot_product_rust(x: &[f32; 2], y: &[f32; 2]) -> f32 {
    x.iter().zip(y).map(|(xi, yi)| xi * yi).sum()
}
```

Are they the same for any input?

Outline

1. Symbolic Execution 101
2. Bug-finding with Owi
3. Test-Case Generation with Owi
3. Proof of Programs with Owi
4. Solver-Aided Programming with Owi

Symbolic Execution 101

What it can do

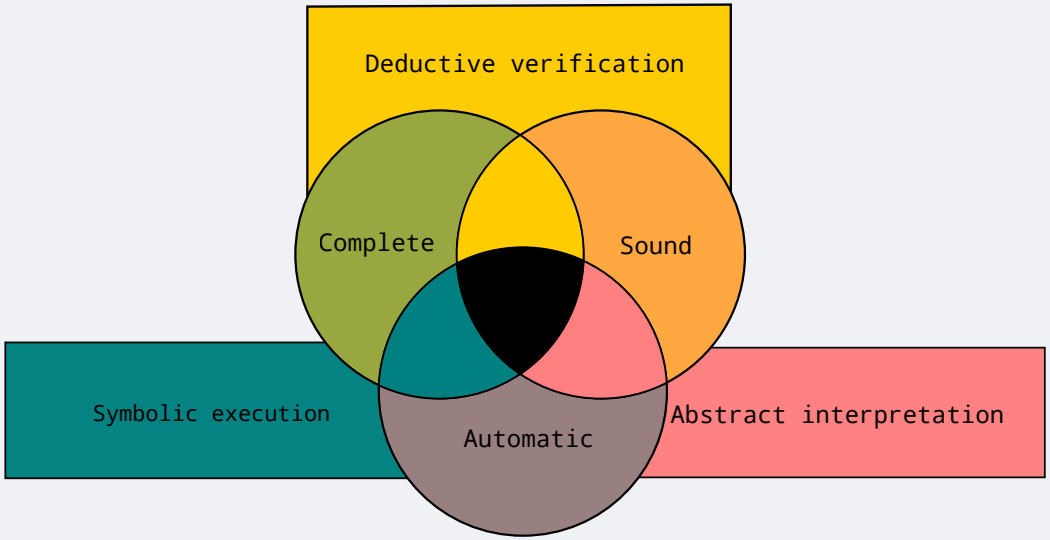
A “static” analysis that:

- explores all paths of a programs;
- at any point of the execution, can give input values leading to the current state.

Used for:

- bug-finding;
- solver-aided programming;
- test-case generation;
- proofs.

Comparison with other methods



- deductive verification: takes time and smart people to do the proof;
- abstract interpretation: takes time to remove false positives;
- symbolic execution: automatically find bugs and only bugs.

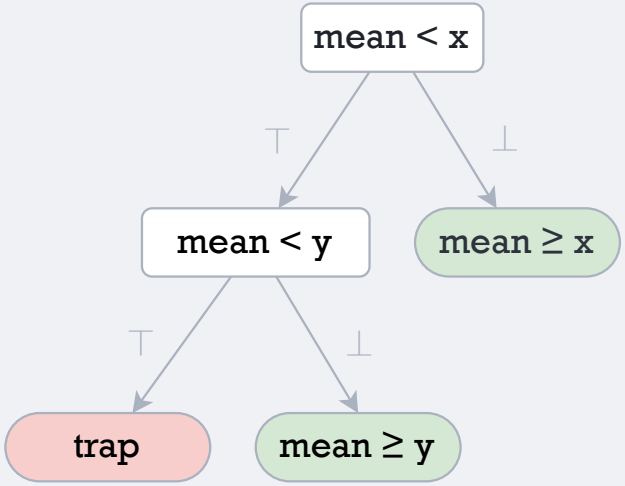
The price is that we may miss some bugs, but for bug-finding it does not matter.

Wasm Example

```
(func $mean (param $x i32) (param $y i32)
  (local $mean i32)

  i32.const 2                          ;; [ 2 ]
  (i32.add
    (local.get $x) (local.get $y)) ;; [ (x+y) ; 2 ]
  i32.div_u                            ;; [ (x+y) / 2 ]
  local.set $mean                      ;; [ ]

  (i32.lt_u (local.get $mean) (local.get $x))
  (if (then
    (i32.lt_u (local.get $mean) (local.get $y))
    (if (then unreachable )))))
```



Wasm Example

```
(func $mean (param $x i32) (param $y i32)
  (local $mean i32)

  i32.const 2                          ;; [ 2 ]
  (i32.add
    (local.get $x) (local.get $y)) ;; [ (x+y) ; 2 ]
  i32.div_u                            ;; [ (x+y) / 2 ]
  local.set $mean                      ;; [ ]

  (i32.lt_u (local.get $mean) (local.get $x))
  (if (then
    (i32.lt_u (local.get $mean) (local.get $y))
    (if (then unreachable )))))
```

```
Unreachable
model {
  symbol x 2147483650
  symbol y 2147483655
}
```

Indeed:

$$\frac{(x \oplus y)}{2} = \frac{2147483650 \oplus 2147483655}{2} = \frac{9}{2} = 4$$

From Concrete To Symbolic

How to get a symbolic interpreter from a concrete one?

1. Abstract Over the Type of Values (use expressions)
2. Abstract Over the Execution Strategy (use a choice monad)

Described in length in our journal article: Owi: Performant Parallel Symbolic Execution Made Easy, an Application to WebAssembly.

We have a Wasm symbolic interpreter that is parallel (thanks to OCaml 5).

Bug-Finding With Owi

C Symbolic Execution

```c
#include <owi.h>

unsigned int mean1(unsigned int x,
                   unsigned int y) {
  return (x & y) + ((x ^ y) >> 1); }

unsigned int mean2(unsigned int x,
                   unsigned int y) {
  return (x + y) / 2; }

void check(unsigned int x, unsigned int y) {
  owi_assert(mean1(x, y) == mean2(x, y)); }

void main(void) {
  unsigned int x = owi_i32();
  unsigned int y = owi_i32();
  check(x, y); }
```

The subcommand `owi c` takes care of compiling and linking:

```
$ owi c ./function_equiv.c
Assert failure
model {
  symbol_0 i32 -922221680
  symbol_1 i32 1834730321
}
Reached problem!
```

Standard library based on `dietlibc`. Special handling of `malloc` and `free` to detect use-after-free or double-free.

C++ Symbolic Execution

```cpp
#include <owi.h>

struct IntPair {
  int x, y;
  int mean1() const {
    return (x & y) + ((x ^ y) >> 1);
  }
  int mean2() const {
    return (x + y) / 2;
  }
};

int main() {
  IntPair p{owi_i32(), owi_i32()};
  owi_assert(p.mean1() == p.mean2());
}
```

The subcommand `owi c++` takes care of compiling and linking:

```
$ owi c++ ./poly.cpp
Assert failure
model {
  symbol symbol_0 i32 -2147483648
  symbol symbol_1 i32 -2147483646
}
Reached problem!
```

Re-using the symbolic libc.

Rust Symbolic Execution

```rust
fn mean1(x: i32, y: i32) -> i32 {
  (x + y) / 2
}

fn mean2(x: i32, y: i32) -> i32 {
  (x & y) + ((x ^ y) >> 1)
}

fn main() {
  let x = owi_sym::u32_symbol() as i32;
  let y = owi_sym::u32_symbol() as i32;
  owi_sym::assert(mean1(x, y) == mean2(x, y))
}
```

The subcommand `owi rust` takes care of compiling and linking:

```
$ owi rust ./main.rs
Assert failure
model {
  symbol symbol_0 i32 1073741835
  symbol symbol_1 i32 -2147483642
}
Reached problem!
```

Re-using the symbolic libc.

Zig symbolic execution

```
fn fibonacci(n: i32) i32 {
    if (n < 0) {
        @panic("expected a positive number");
    }
    if (n <= 2) return n;
    return fibonacci(n - 1) + fibonacci(n -
2);
}

pub fn main() void {
    const n: i32 = i32_symbol();
    assume(n > 0);
    assume(n < 10);
    const result = fibonacci(n);
    assert(result != 21);
}
```

The subcommand `owi zig` takes care of compiling and linking:

```
$ owi zig ./fib.zig
owi: [ERROR] Assert failure
model {
  symbol symbol_0 i32 7
}
owi: [ERROR] Reached problem!
```

Re-using the symbolic libc.

Cross-Language: Moving a Codebase from C to Rust

Original C version:

```c
float dot_product(float x[2], float y[2]) {
    return (x[0]*y[0] + x[1]*y[1]);
}
```

New Rust version:

```rust
fn dot_product_rust(x: &[f32; 2], y: &[f32; 2]) -> f32 {
    x.iter().zip(y).map(|(xi, yi)| xi * yi).sum()
}
```

Is It Correct?

Owi says no:

```
model {
  symbol_0 f32 -0.
  symbol_1 f32 -0.
  symbol_2 f32 0.
  symbol_3 f32 0.
}
```

Breaking it Down

C version:

```
x[0] * y[0] + x[1] * y[1]
```

Rust version:

```
x.iter().zip(y).map(|(xi, yi)| xi * yi).sum()
```

Breaking it Down

C version:

```
x[0] * y[0] + x[1] * y[1]
-0. *   0. +  -0. *   0.
```

Rust version:

```
x.iter().zip(y).map(|(xi, yi)| xi * yi).sum()
[-0.,-0.].iter().zip([0.,0.]).map(|(xi,yi)|xi*yi).sum()
[(-0.,0.),(-0.,0.)].map(|(xi,yi)|xi*yi).sum()
```

Breaking it Down

C version:

```
x[0] * y[0] + x[1] * y[1]
-0. *   0. +  -0. *   0.


-0. + -0.
```

Rust version:

```
x.iter().zip(y).map(|(xi, yi)| xi * yi).sum()
[-0.,-0.].iter().zip([0.,0.]).map(|(xi,yi)|xi*yi).sum()
[(-0.,0.),(-0.,0.)].map(|(xi,yi)|xi*yi).sum()
[-0., -0.].sum()
```

Breaking it Down

C version:

```
x[0] * y[0] + x[1] * y[1]
-0. *   0. +  -0. *   0.


-0. + -0.
```

Rust version:

```
x.iter().zip(y).map(|(xi, yi)| xi * yi).sum()
[-0.,-0.].iter().zip([0.,0.]).map(|(xi,yi)|xi*yi).sum()
[(-0.,0.),(-0.,0.)].map(|(xi,yi)|xi*yi).sum()
[-0., -0.].sum()
+0. + -0. + -0.
```

Breaking it Down

C version:

```
x[0] * y[0] + x[1] * y[1]
-0. *   0. +  -0. *   0.

-0. + -0.

-0.
```

Rust version:

```
x.iter().zip(y).map(|(xi, yi)| xi * yi).sum()
[-0.,-0.].iter().zip([0.,0.]).map(|(xi,yi)|xi*yi).sum()
[(-0.,0.),(-0.,0.)].map(|(xi,yi)|xi*yi).sum()
[-0., -0.].sum()
+0. + -0. + -0.
+0.
```

Breaking it Down

C version:

```
x[0] * y[0] + x[1] * y[1]
-0. *   0. +  -0. *   0.

-0. + -0.

-0.
```

Rust version:

```
x.iter().zip(y).map(|(xi, yi)| xi * yi).sum()
[-0.,-0.].iter().zip([0.,0.]).map(|(xi,yi)|xi*yi).sum()
[(-0.,0.),(-0.,0.)].map(|(xi,yi)|xi*yi).sum()
[-0., -0.].sum()
+0. + -0. + -0.
+0.
```

- fixed in the Rust standard library

Breaking it Down

C version:

```
x[0] * y[0] + x[1] * y[1]
-0. *   0. +  -0. *   0.

-0. + -0.

-0.
```

Rust version:

```
x.iter().zip(y).map(|(xi, yi)| xi * yi).sum()
[-0.,-0.].iter().zip([0.,0.]).map(|(xi,yi)|xi*yi).sum()
[(-0.,0.),(-0.,0.)].map(|(xi,yi)|xi*yi).sum()
[-0., -0.].sum()
+0. + -0. + -0.
+0.
```

- fixed in the Rust standard library
- initial accumulator for `sum()` is now `-0.`

Breaking it Down

C version:

```
x[0] * y[0] + x[1] * y[1]
-0. *   0. +  -0. *   0.


-0. + -0.


-0.
```

Rust version:

```
x.iter().zip(y).map(|(xi, yi)| xi * yi).sum()
[-0.,-0.].iter().zip([0.,0.]).map(|(xi,yi)|xi*yi).sum()
[(-0.,0.),(-0.,0.)].map(|(xi,yi)|xi*yi).sum()
[-0., -0.].sum()
+0. + -0. + -0.
+0.
```

- fixed in the Rust standard library
- initial accumulator for `sum()` is now `-0.`
- it broke typst (the tool used to make these slides) that was relying on this behaviour

Checking Binaryen's Optimizations

Binaryen is written in C++ and can optimize Wasm programs.

1. It has fuzzer based on AFL that generates random Wasm files and is guided by how much the C++ code is covered while optimizing the Wasm module
2. At the end, we have `a.wasm` and `optimized.wasm`
3. Output is compared when using basic inputs like `0`, `1` and `MAX_INT`

=> Instead of 3. we can use Owi to check they are the same for any input

The PR is almost ready to get merged.

Test-Case Generation (Saïd Zuhair's internship)

You get it for free

Any symbolic execution engine can generate tests for free. But we want to go further…

Advanced Coverage Criteria

1. (FC) Function Coverage (% of function called)
2. (SC) Statement Coverage (% of statement called)

Let's say you have `if (e1 && e2)`

3. (DC) Decision Coverage: `e1 && e2` and `not(e1 && e2)`
4. (CC) Condition Coverage: `e1` and `not(e1)` and `e2` and `not(e2)`
5. (DCC) Decision-Condition Coverage: DC + CC
6. (MCC) Multiple-Condition Coverage: `e1 && e2` and `not(e1) && e2` and `e1 && not(e2)` and `not(e1) && not(e2)`
7. (MC/DC) Modified Condition/Decision Coverage: each sub expression has an impact on the whole decision and more weird stuff
    1. (GACC) General Active Clause Coverage
    2. (GICC) General Inactive Clause Coverage
8. ...

Why Is It Useful?

Some of them are required for certification in some industry. For instance, MC/DC is required in Aeronautics.

We don’t want to support all of them, it is very tedious.

Labels to the rescue

Specify and measure, cover and reveal: A unified framework for automated test generation by Bardin et al. For a given criteria+program, it generates an instrumented program with labels:

```
int numPos(int a) { int n = 0;

  pc_label(a > 0 && a < 20,1,"MCC");
  pc_label(a > 0 && ! (a < 20),2,"MCC");
  pc_label(! (a > 0) && a < 20,3,"MCC");
  pc_label(! (a > 0) && ! (a < 20),4,"MCC");

  if (a > 0 && a < 20) n++;

  return n; }
```

Your test only need to call a `pc_label` with `true` to validate it.

Label support in Owi

Built-in support of labels is hard, if you do it naïvely, the number of paths explode because each of them adds a lot of branching. We have an efficient white-box version in Owi. Previously it has been done in a black-box way for KLEE.

Built-in supports allows to dynamically guide the execution towards unseen labels.

We would like to re-implement the label generation part at the Wasm level to be able to provides advanced coverage criteria to many languages!

Proof of Programs with Owi (Zhicheng Hui's internship)

ACSL

The ANSI/ISO C Specification Language (ACSL).

Allows to write function contracts:

```
/*@ requires precondition
    ensures  postcondition
*/
int f(int n) { ... }
```

But also assertions, loop invariants, type invariants…

E-ACSL

The Executable subset of ACSL (E-ACSL).

```
/*@ requires n <= INT_MAX - 3;
    ensures \result == n + 3; */
int plus_three(int n) {
  return n + 3;
}
```

```
int __gen_e_acsl_plus_three(int n) {
  long __gen_e_acsl_at = (long)n;
  int __retres;
    { __e_acsl_assert_register_int(...);
    __e_acsl_assert(n <= 2147483644);
    __e_acsl_assert_clean(...); }
  __retres = plus_three(n);
    { __e_acsl_assert_register_int(...);
    __e_acsl_assert_register_long(...);
    __e_acsl_assert((long)__retres ==
__gen_e_acsl_at + 3L);
    __e_acsl_assert_clean(...); }
}
```

E-ACSL Support in Owi

We re-use the code generator from E-ACSL, but uses our own symbolic E-ACSL runtime:

```
void __e_acsl_assert(int predicate, __e_acsl_assert_data_t *data) {
    owi_assert(predicate);
}
```

Available through `owi c --e-acsl`. It allows to symbolically execute code annotated by specifications by generating executable assertions.

Described in our paper Cross-Language Symbolic Runtime Annotation Checking. There we show how the subset of ACSL supported by E-ACSL can be extended when targeting symbolic execution.

Weasel

We started to do the same in Wasm:

```
(module
  (@contract $plus_three
    (ensures (= result (+ $n 3))))
  (func $plus_three (param $n i32)
      (result i32)
    local.get $n
    i32.const 3
    i32.add
  ))
```

Design of Weasel (WEbAssembly SpEcification Language).

It uses the custom annotation syntax proposal.

Generating Assertions from Weasel

We did something similar to E-ACSL, still experimental :

```
(module
  (@contract $plus_three
    (ensures (= result (+ $n 3))))
  (func $plus_three (param $n i32)
(result i32)
    local.get $n
    i32.const 3
    i32.add
  )
  (start $plus_three)
)
```

```
(import "symbolic" "assert"
  (func $assert (param i32)))
(func $__weasel_plus_three (param $n i32)
(result i32) (local $__weasel_temp i32)
(local $__weasel_res_0 i32)
  (call $plus_three (local.get 0))
  local.set 2
  (i32.eq (local.get 2) (i32.add (local.get
0) (i32.const 3)))
  call $assert
  local.get 2
)
(start $__weasel_plus_three)
```

What can we do with this?

Contrary to most symbolic execution engine, Owi does not perform any approximation (modulo the source language compiler approximation wrt. undefined behaviours).

When the analysis terminates, we've got a proof!

It could be used to proove programs or functions on its own.

It could also combined with:

- a deductive verification tool to automate the proof, or find counter-examples;
- an abstract interpretation engine to confirm or infirm bugs found.

Solver-aided Programming

Polynomial Example

```c
#include <owi.h>

void f(void) {
  int x = owi_i32();
  int x2 = x * x;
  int x3 = x * x * x;

  int a = 1;  int b = -7;
  int c = 14; int d = -8;

  int poly =
    a * x3 + b * x2 + c * x + d;

  owi_assert(poly != 0);
}
```

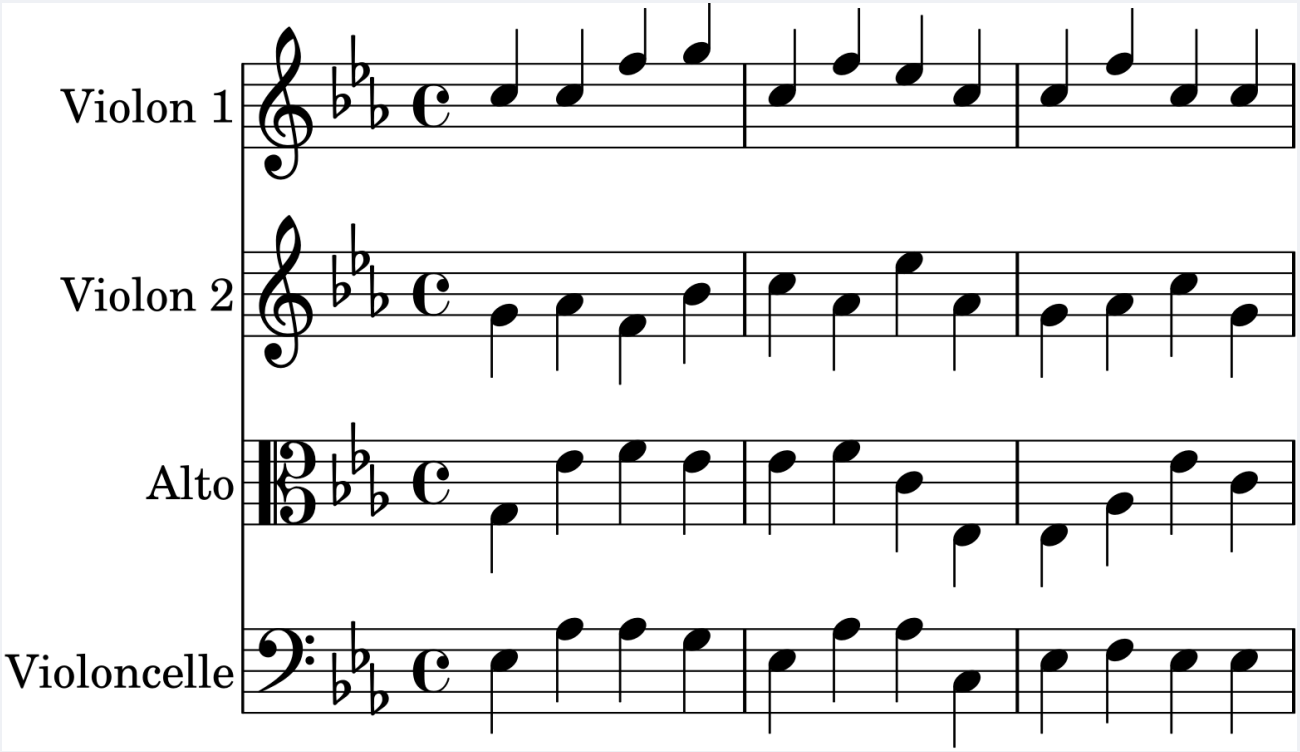
This is similar to Rosette for Racket ("solver-aided programming") but:

- parallel
- multi/cross-language

We used it for :

- solving a maze
- generate a set of cards for dobble
- generate strongly regular graph with parameters (9,4,1,2)
- generate music sheet for a string quartet

Music Generation



- limit on the instruments' range
- no crossing
- no leap of more than an octave
- notes belongs to the key
- the 7th degree resolves to the tonic
- instruments form chords
- no parallel fifths or octaves

Conclusion

Why Wasm Was a Good Choice

- small (C++ would have been infeasible at the source level)
- no undefined behaviour (we don't have to make any choice)
- people write compilers for us (Vyper people have to do it themselves)
- backward compatibility (KLEE lags behind several LLVM version)
- easier for cross-language (Rust + C at the LLVM level with KLEE is too hard!)
- well optimized (optimizations for concrete execution are beneficial for symbolic execution too)
- close to the smtlib

Stuff I did not talked about

- `owi run`, `owi script`, `owi wasm2wat`, `owi fmt`, `owi wat2wasm`
- it can run Wasm in a Unikernel with Solo5 and MirageOS!
- the Smt.ml library
- automatic harness generation
- a fuzzer for Wasm interpreters
- benchmarks (we are the best for Wasm, close to KLEE, the best one for C)
- concolic execution
- optimisations we have (path-condition independence, negation shortcut)

Future work already funded

- add heuristics to explore interesting paths first (Julie, starting on Monday)
- abstract interpretation of Wasm (PhD student starting in September)
- proper symbolic system interface (WASI, Componen Model, Common ABI)
- support missing proposals (SIMD, multi-memories, exceptions, memory64, GC)
- new languages: Haskell, TinyGo, OCaml, Guile, (maybe Dart, Java, Kotlin)
- build system integration (`CC='owi clang'`, `cargo owi`)
- case study on real libraries (started to check Wayland stuff with emersion)

If you have ideas, please tell me!

Conclusion

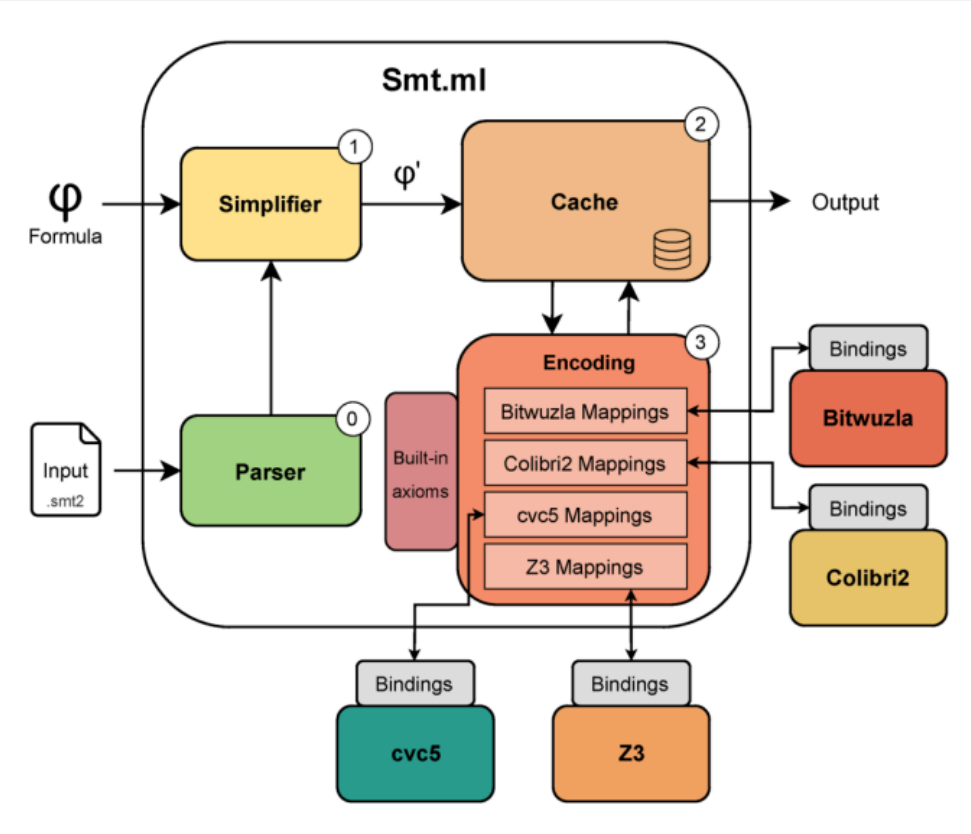
Owi is an efficient symbolic execution engine for Wasm, C, C++, Rust and Zig that can perform cross-language analysis but can also be used as a Wasm toolkit for developpers and researchers. We want to make it a real-world tool.

It is free software! You can try it at github.com/ocamlpro/owi . Documentation is available at ocamlpro.github.io/owi . You must built it from sources to get most of what I presented but I am preparing a new release.

We want to explore industrial applications and welcome discussions with users interested in Owi, as well as R&D on Wasm and programming languages.

Bonus

The Smt.ml library



- provides a type of symbolic expressions
- can map expressions to many SMT-solver
- provides optimisations (simplifications, cache through hash-consing)
- ease of use (more typing)
- incremental mode

Automatic Harness Generation

```
void f(unsigned int x, unsigned int y) {
  // ... complicated stuff
}

void f_harness(void) {
  unsigned int x = owi_i32(); unsigned int y = owi_i32();
  f(x, y); }
```

It is annoying to write, so we have automatic harness generation:

```
void f(unsigned int x, unsigned int y) {
  // ... complicated stuff
}
```

```
$ owi c ./function_equiv.c --entry-point=f --invoke-with-symbols
```

No need to touch source code to test the program anymore!

(Old) Benchmarks

On Wasm Code

A hand-written Wasm B-Tree library with 27 possible configurations (number of symbols):

Tool	Min	Max	Mean
Owi-24	1.0	1.0	1.0
Owi-1	0.6	14.0	4.5
WASP	0.4	16.4	4.1
SeeWasm	2.5	101	57.1
Manticore	17.2	844	312

On C Code

1215 C programs from Test-Comp.

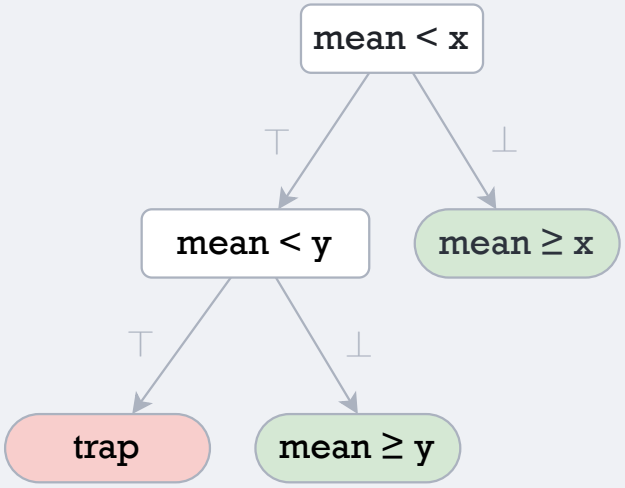
Tool	Bug found	Timeout	Bug not found
KLEE	782	368	65
Owi	676	539	0
Symbiotic	489	657	69

Good results, especially when we know that Owi:

- does no approximation;
- has no optimisation appart from the multi-core;
- these are old benchmarks…

Most of the time is spent in the solver. What can we do about it?

One new optimisation: concolic execution



- we begin with random values for symbols
- we keep the symbolic and concrete state
- no need to call the solver at each branch
- but we still keep the PC
- if we found a bug, we're done
- otherwise, we start again but with values leading to a new branch (we ask the SMT using our list of PC)

This is what most engine are doing. AKA "dynamic symbolic execution".

Another new optimisation: path-condition slicing

The PC contains many formulas unrelated to most branching conditions.

This is slowing-down the SMT-solver.

We use a union-find data-structure where keys are variables and nodes are set of (related) constraints.

When meeting a new branch, we add the condition to the PC, then slice it, and only send the slice to the solver.