Owi: cross-language symbolic execution for bug-finding and solver-aided programming

Léo Andrès, Pierre Chambart,
Arthur Carcano, Zhicheng Hui @ OCamlPro

Filipe Marques @ University of Lisbon

With contributions from Eric Patrizio, Olivier Pierre, Vasu Singh, Basile Clément, Hichem Rami Ait El Hara, Dario Pinto, Neha Chriss, Frederico Ramos, Jean-Christophe Filliâtre, José Fragoso Santos

13th of February, 2025 – PPS Seminar

Context

The Web client-side: HTML

HTML

World Wide Web

The WorldWideWeb (W3) is a wide-area hypermedia information retrieval initiative aiming to give universal access to a large universe of documents.

Everything there is online about W3 is linked directly or indirectly to this document, including an executive summary of the project, Mailing lists , Policy , November's W3 news , Frequently Asked Questions .

What's out there?
: Pointers to the world's online information, subjects , W3 servers, etc.

Help
: on the browser you are using

Software Products
: A list of W3 project components and their current state. (e.g. Line Mode ,X11 Viola , NeXTStep , Servers , Tools , Mail robot , Library)

Technical
: Details of protocols, formats, program internals etc

Bibliography
: Paper documentation on W3 and references.

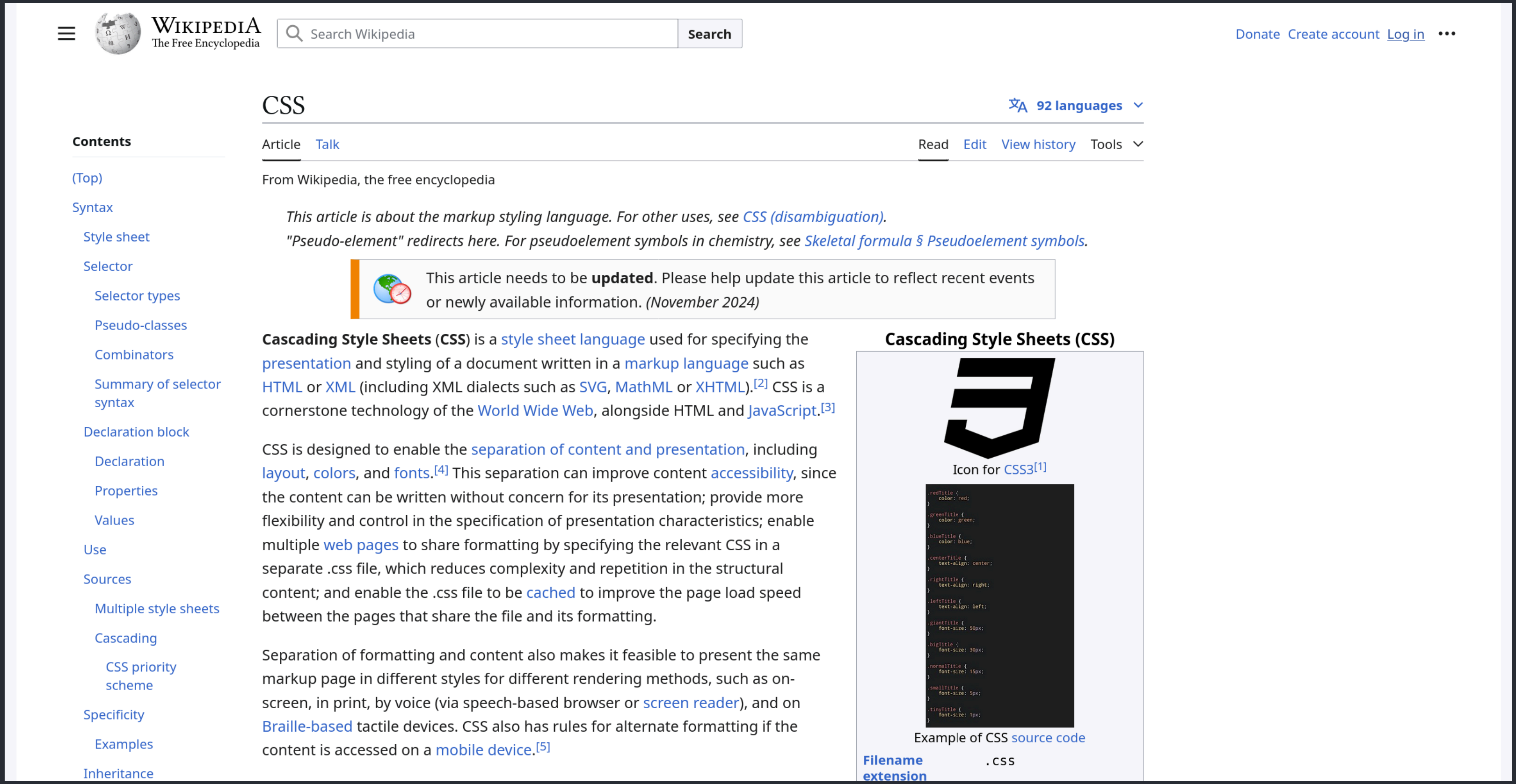
People
: A list of some people involved in the project.

History
: A summary of the history of the project.

How can I help ?
: If you would like to support the web..

Getting code
: Getting the code by anonymous FTP , etc.

The Web client-side: CSS



Wikipedia — The Free Encyclopedia | Search Wikipedia | Search | Donate | Create account | Log in

CSS

92 languages

Article | Talk | Read | Edit | View history | Tools

Contents

From Wikipedia, the free encyclopedia

This article is about the markup styling language. For other uses, see CSS (disambiguation).
"Pseudo-element" redirects here. For pseudoelement symbols in chemistry, see Skeletal formula § Pseudoelement symbols.

This article needs to be **updated**. Please help update this article to reflect recent events or newly available information. *(November 2024)*

Cascading Style Sheets (**CSS**) is a style sheet language used for specifying the presentation and styling of a document written in a markup language such as HTML or XML (including XML dialects such as SVG, MathML or XHTML).[2] CSS is a cornerstone technology of the World Wide Web, alongside HTML and JavaScript.[3]

CSS is designed to enable the separation of content and presentation, including layout, colors, and fonts.[4] This separation can improve content accessibility, since the content can be written without concern for its presentation; provide more flexibility and control in the specification of presentation characteristics; enable multiple web pages to share formatting by specifying the relevant CSS in a separate .css file, which reduces complexity and repetition in the structural content; and enable the .css file to be cached to improve the page load speed between the pages that share the file and its formatting.

Separation of formatting and content also makes it feasible to present the same markup page in different styles for different rendering methods, such as on-screen, in print, by voice (via speech-based browser or screen reader), and on Braille-based tactile devices. CSS also has rules for alternate formatting if the content is accessed on a mobile device.[5]

Cascading Style Sheets (CSS)

Icon for CSS3[1]

Example of CSS source code

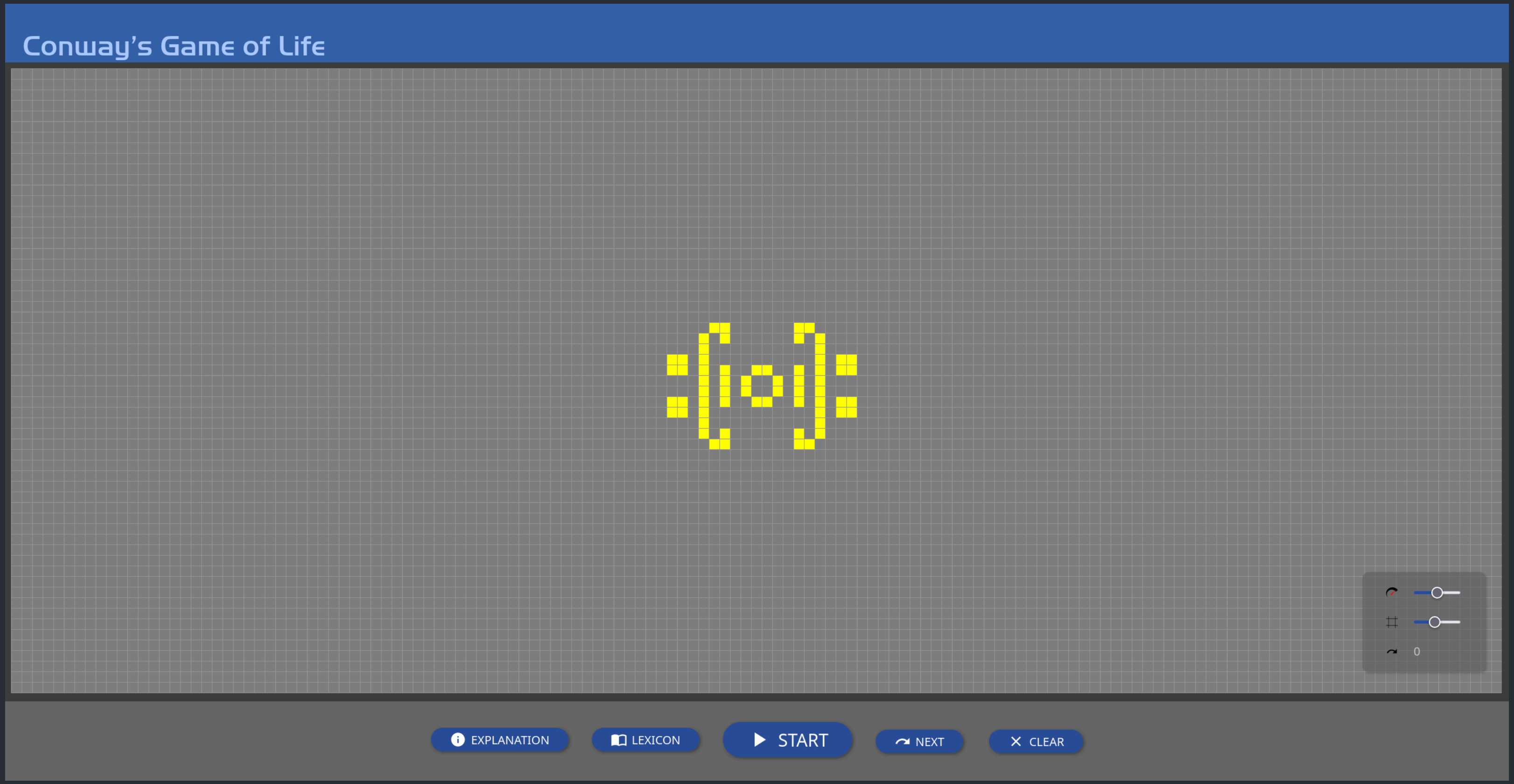
Filename extension	.css

The Web client-side: JavaScript

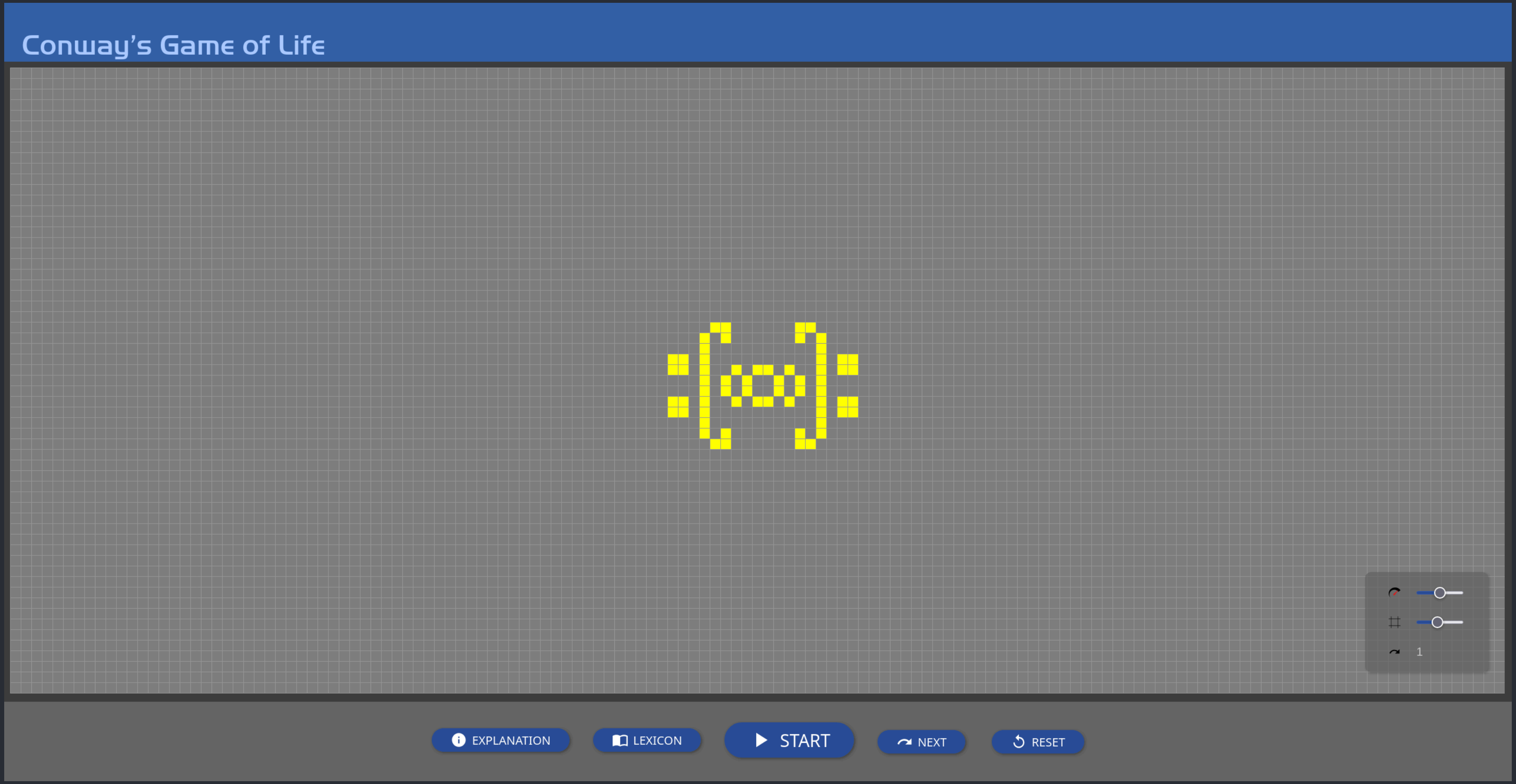
HTML

CSS

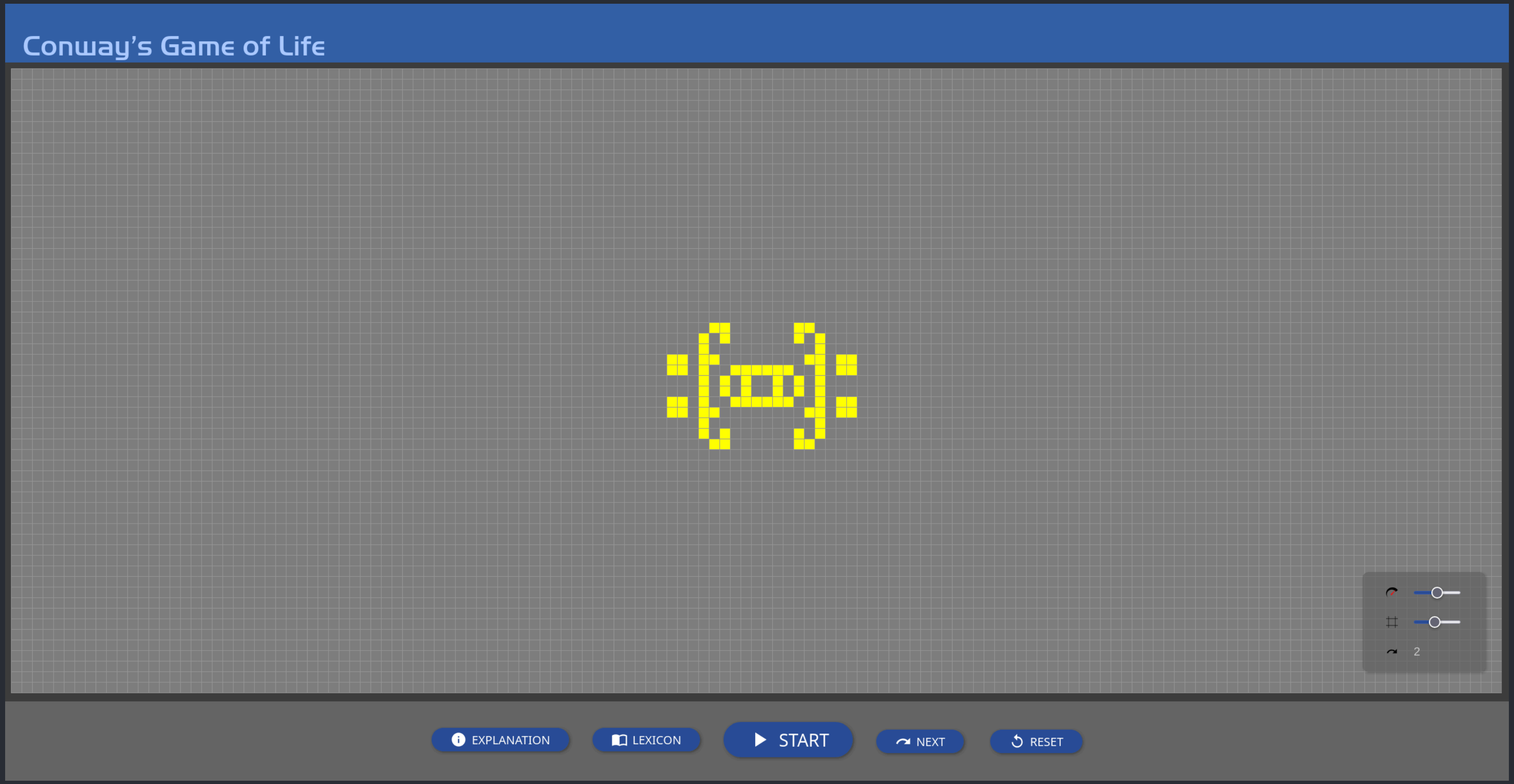
JS



The Web client-side: JavaScript



The Web client-side: JavaScript

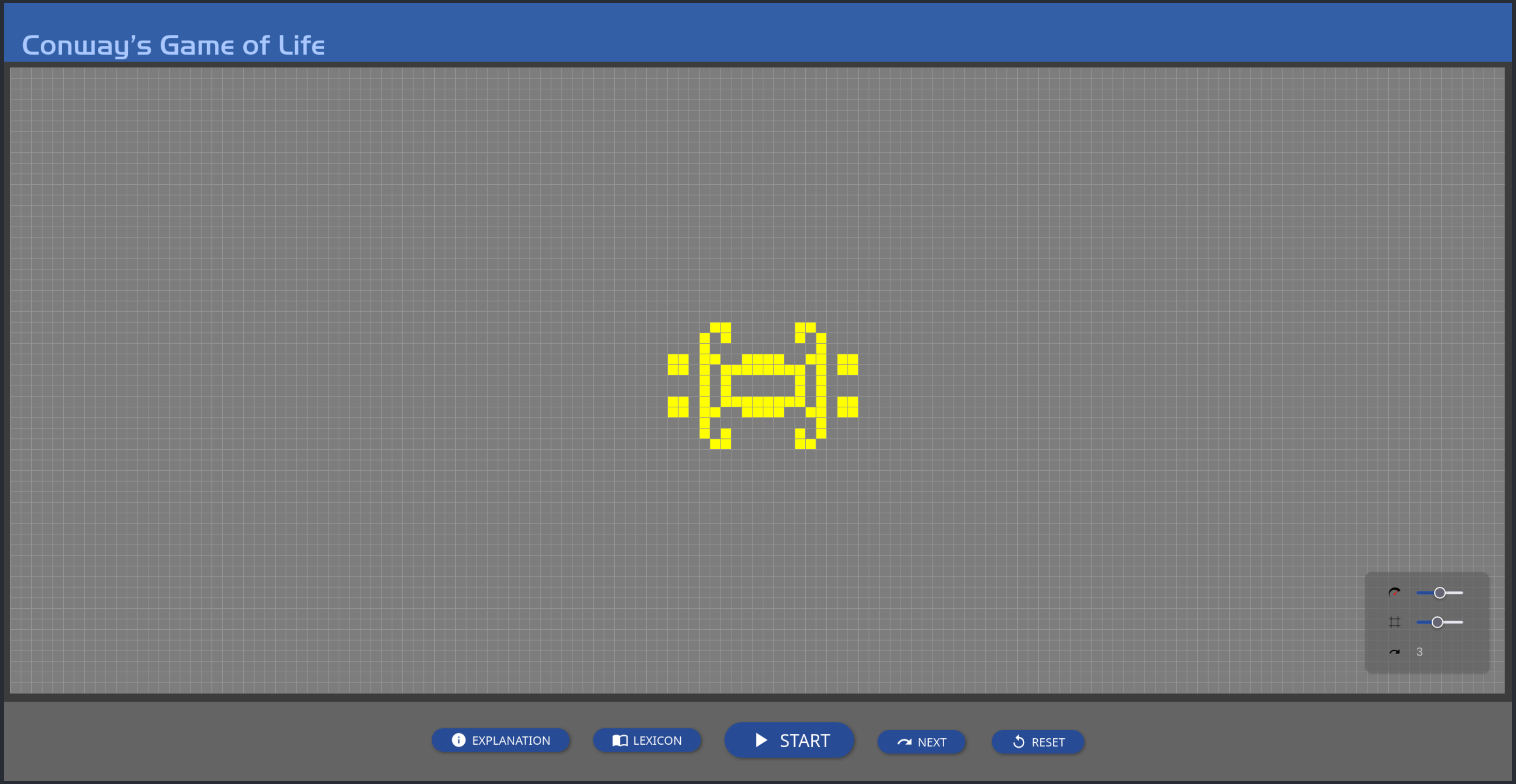


The Web client-side: JavaScript

HTML

CSS

JS

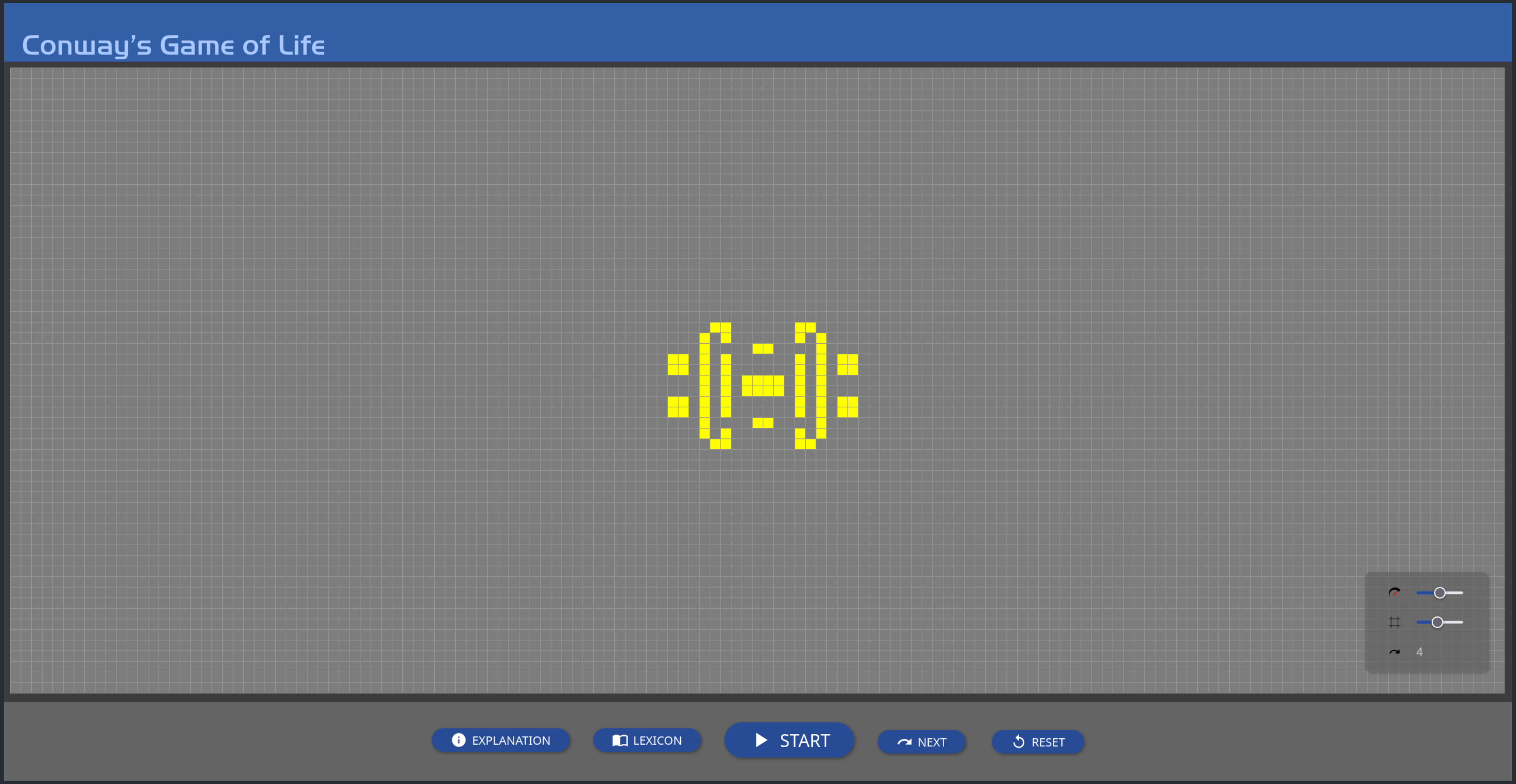


The Web client-side: JavaScript

HTML

CSS

JS



JavaScript Downsides

JavaScript is hard:

- as a programming language: we want to use other languages;
- as a compilation target: we need another compilation target.

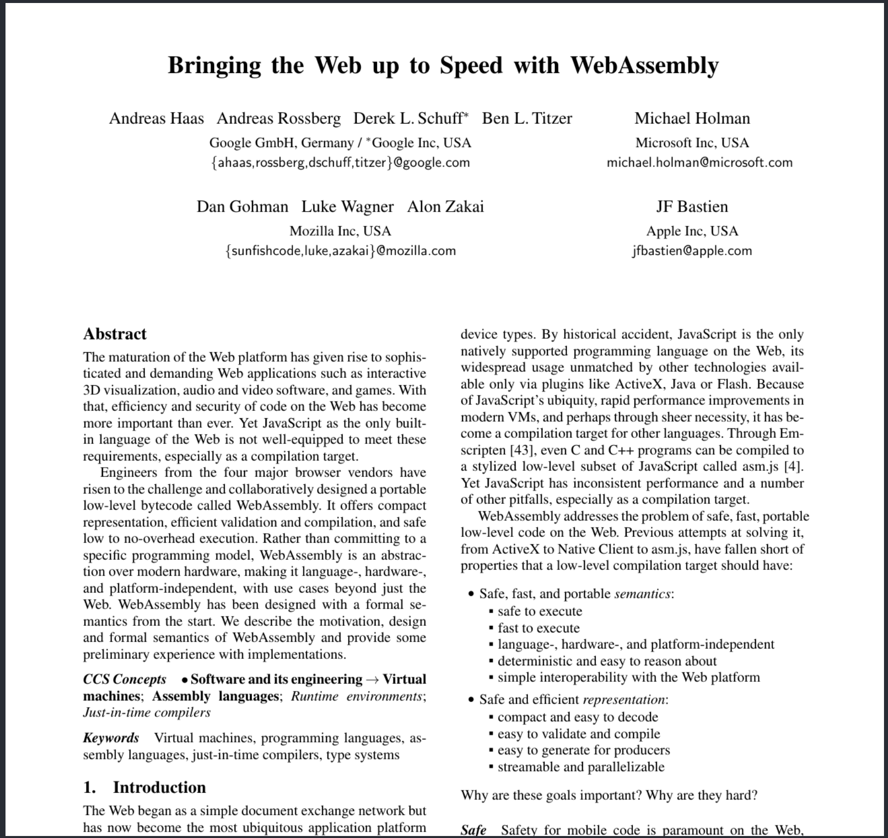
Consensus to provide an alternative that is:

- fast
- safe
- portable
- a good compilation target

WebAssembly (Wasm)

- announced in 2015
- available since 2017 in browsers

- today, many languages can compile to Wasm: C, C++, Rust, OCaml, Java, Guile, Go, Haskell
- tt is used for server deployments (a lot) and embedded



Bringing the Web up to Speed with WebAssembly

Andreas Haas Andreas Rossberg Derek L. Schuff* Ben L. Titzer
Google GmbH, Germany / *Google Inc, USA
{ahaas,rossberg,dschuff,titzer}@google.com

Michael Holman
Microsoft Inc, USA
michael.holman@microsoft.com

Dan Gohman Luke Wagner Alon Zakai
Mozilla Inc, USA
{sunfishcode,luke,azakai}@mozilla.com

JF Bastien
Apple Inc, USA
jfbastien@apple.com

Abstract

The maturation of the Web platform has given rise to sophisticated and demanding Web applications such as interactive 3D visualization, audio and video software, and games. With that, efficiency and security of code on the Web has become more important than ever. Yet JavaScript as the only built-in language of the Web is not well-equipped to meet these requirements, especially as a compilation target.

Engineers from the four major browser vendors have risen to the challenge and collaboratively designed a portable low-level bytecode called WebAssembly. It offers compact representation, efficient validation and compilation, and safe low to no-overhead execution. Rather than committing to a specific programming model, WebAssembly is an abstraction over modern hardware, making it language-, hardware-, and platform-independent, with use cases beyond just the Web. WebAssembly has been designed with a formal semantics from the start. We describe the motivation, design and formal semantics of WebAssembly and provide some preliminary experience with implementations.

CCS Concepts • **Software and its engineering** → **Virtual machines**; **Assembly languages**; *Runtime environments*; *Just-in-time compilers*

Keywords Virtual machines, programming languages, assembly languages, just-in-time compilers, type systems

1. Introduction

The Web began as a simple document exchange network but has now become the most ubiquitous application platform device types. By historical accident, JavaScript is the only natively supported programming language on the Web, its widespread usage unmatched by other technologies available only via plugins like ActiveX, Java or Flash. Because of JavaScript's ubiquity, rapid performance improvements in modern VMs, and perhaps through sheer necessity, it has become a compilation target for other languages. Through Emscripten [43], even C and C++ programs can be compiled to a stylized low-level subset of JavaScript called asm.js [4]. Yet JavaScript has inconsistent performance and a number of other pitfalls, especially as a compilation target.

WebAssembly addresses the problem of safe, fast, portable low-level code on the Web. Previous attempts at solving it, from ActiveX to Native Client to asm.js, have fallen short of properties that a low-level compilation target should have:

- Safe, fast, and portable *semantics*:
  - safe to execute
  - fast to execute
  - language-, hardware-, and platform-independent
  - deterministic and easy to reason about
  - simple interoperability with the Web platform
- Safe and efficient *representation*:
  - compact and easy to decode
  - easy to validate and compile
  - easy to generate for producers
  - streamable and parallelizable

Why are these goals important? Why are they hard?

Safe Safety for mobile code is paramount on the Web,

Outline

1. introduction to Wasm
2. Owi as a Wasm toolkit
3. symbolic execution in a nutshell
4. from a concrete to a parallel symbolic interpreter
5. C, C++ and Rust symbolic execution
6. cross-language bug-finding
7. solver-aided programming
8. towards proofs
9. benchmarks

Introduction to Wasm

Wasm 1.0 (2017)

- stack-based language;
- simple types (`i32`, `i64`, `f32`, `f64`) ;
- statically typed (`[ i32 ; f32 ] -> [ i32 ]`);
- functions ;
- a linear memory (an array of bytes) ;
- possibility to import and export functions;
- a formal semantics, with no undefined behabiour.

Wasm 2.0 (2022) Non-trapping float-to-int conversion, Sign-extension operators, Multi-value, Bulk memory operations, Fixed-width SIMD…

Wasm 3.0 (2024/2025) Typed Function References, Tail Call, Garbage Collection, Exception handling…

Example of a Wasm Program

```
(module

  (func $f (param $n i32) (result i32)
                                                       ;; []
    (i32.lt_s (local.get $n) (i32.const 2)) ;; [ n < 2 ]
    (if (then      ;; [ ]
      local.get $n ;; [ n ]
      return ))    ;; early return
                                           ;; [ ]
    (i32.sub (local.get $n) (i32.const 2)) ;; [ n-2 ]
    call $f                                ;; [ f(n-2) ]
    (i32.sub (local.get $n) (i32.const 1)) ;; [ n-1; f(n-2) ]
    call $f                                ;; [ f(n-1); f(n-2) ]
    i32.add                                ;; [ f(n-1) + f(n-2)]
                                           ;; implicit return

  ))
```

Linear Memory

```
(module
  (memory 1) ;; initial size of 1 page
  (func $f (param $addr i32) (result f32)
                        ;; []
    local.get $addr ;; [ addr ]
    i32.const 4     ;; [ 4; addr ]
    i32.mul         ;; [ 4 * addr ]
    f32.load        ;; [ float(memory[4 * addr]) ]
  )
)
```

Host Interactions

```
(module
  (import "stdlib" "print_i32" (func $print_i32 (param i32)))
  (func $f
                         ;; []
    i32.const 42         ;; [ 42 ]
    call $print_i32      ;; [] ; 42 is printed
  )
)
```

Owi as a Wasm toolkit

Owi?

Owi is a Wasm interpreter written in OCaml.

Initially, it was made to:

- learn Wasm;
- experiment with proposals needed to compile OCaml to Wasm.

But it ended-up being well-tested and supporting all of Wasm…

List of Subcommands

Owi provides many commands to work with Wasm:

- `owi fmt`
- `owi opt`
- `owi run`
- `owi script`
- `owi validate`
- `owi version`
- `owi wasm2wat`
- `owi wat2wam`

Supported extensions

Extension	Status
Import/Export of Mutable Globals	✓
Non-trapping float-to-int conversions	✓
Sign-extension operators	✓
Multi-value	✓
Reference Types	✓
Bulk memory operations	✓
Fixed-width SIMD	❌
Tail calls	✓
Typed Function References	✓
GC	❌
Custom Annotation Syntax in the Text Format	✓
Extended Constant Expressions	✓
Exception handling	❌

Dagstuhl Mars 2023

In March 2023, I was giving a talk about Wasocaml, an OCaml to Wasm compiler, and briefly mentionned Owi.

> What about making a symbolic interpreter with Owi?
>
> — José Fragoso Santos (Assistant Professor in Lisbon)

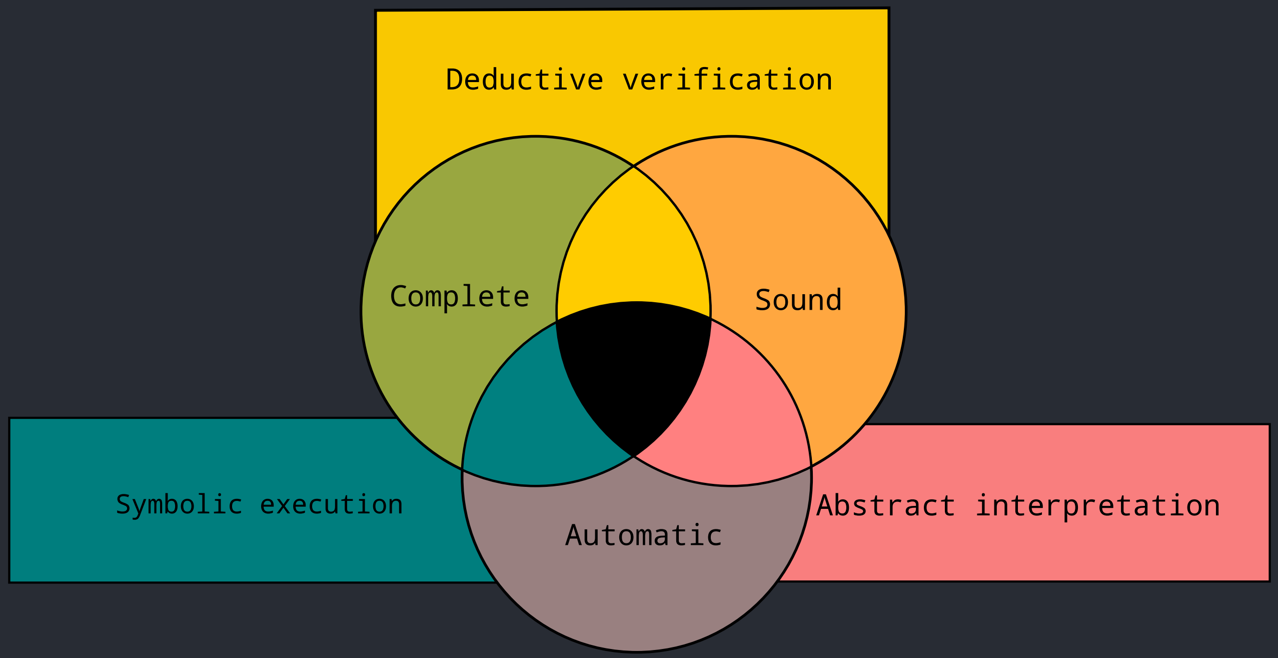
Problem: I don't really know what symbolic execution is…

Symbolic Execution in a Nutshell

Symbolic Execution Purposes

A technique for:

- finding bugs in programs (and proving properties);
- implementing solver-aided programming;
- test-case generation.



Core ideas

We want to find input values leading to a state S.

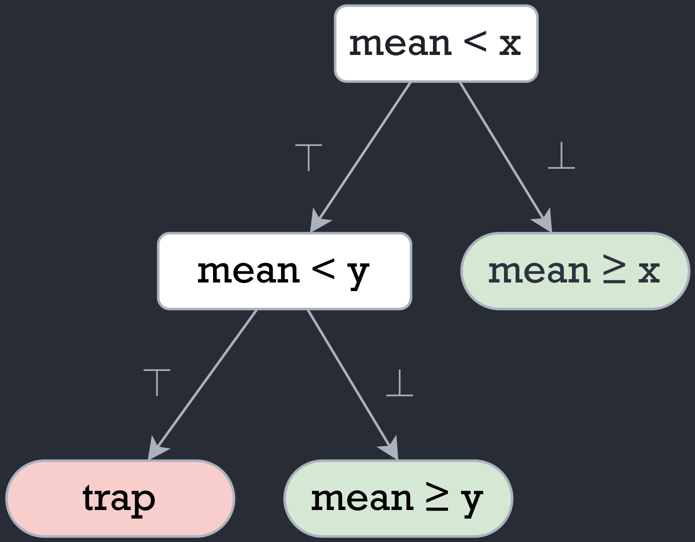
- input values are represented by symbols
- the program executes with expressions made of concrete values + symbols
- when branching both branches are explored
- information about previous branches is kept in the path condition (PC)
- when S is reached, a model is generated by an SMT solver from the PC

This model corresponds to the input values leading to the state S.

Execution Tree

```
unsigned int mean(unsigned int x,
                  unsigned int y) {

  unsigned int mean = (x + y) / 2;
  if (mean < x) {
    if (mean < y) { assert(false); }
  }
  return mean;
}
```



Symbols, harness and model

```c
unsigned int mean(unsigned int x,
                  unsigned int y) {

  unsigned int mean = (x + y) / 2;
  if (mean < x) {
    if (mean < y) { assert(false); }
  }
  return mean;
}

void harness(void) {
  unsigned int x = symbol_int(),
               y = symbol_int();
  mean(x, y);
}
```

```
Assert failure
model {
  symbol x 2147483650
  symbol y 2147483655
}
```

Indeed:

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

From a Concrete to a Parallel Symbolic Interpreter

Owi's Old Concrete Interpreter

Initially, something like:

```
match instr, stack with
  Binop Add      , x :: y :: stack -> (add_i32 x y) :: stack
| If_else (t, f),   cond :: stack ->
  let cond = bool_of_i32 cond in
  if cond then eval t stack
          else eval f stack
```

How to get a symbolic interpreter from this?

Step 1/2 : Abstract Over the Type of Values

```
module type Value = sig
  type t
  val add_i32 : t -> t -> t
  type bool
  val bool_of_i32 : t -> bool
end
```

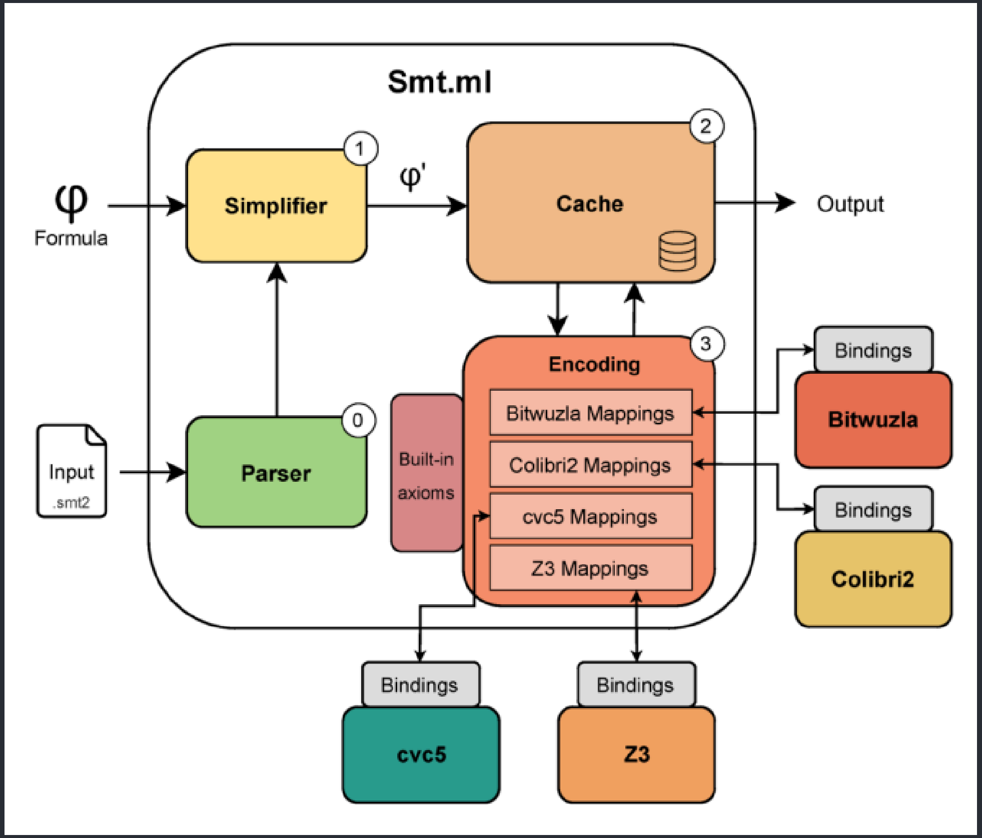
```
| Binop Add, x :: y :: stack ->
  (Value.add_i32 x y) :: t
| If_else (if_t, if_f), cond :: stack ->
  let cond = Value.bool_of_i32 cond in
  if cond then eval if_t stack
          else eval if_f stack
```

Definition of `type t` can change :

- concrete case : a concrete value (`42`)
- symbolic case : a symbolic expression (`x < 42 && y = x || y = 22`)

What is our expression language?

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

Step 2/2 : Abstract Over the Execution Strategy

```
module type Choice = sig
 type 'a t
 val return: 'a -> 'a t
 val bind: 'a t -> ('a -> 'b t) -> 'b t
 val select: Value.bool -> bool t
end
```

```
| If_else (if_t, if_f), cond::stack ->
  let cond = Value.bool_of_i32 cond in
  (* the single new line: *)
  let* cond = Choice.select cond in
  if cond then eval if_t stack
          else eval if_f stack
```

- most of the code unchanged
- we must insert `Choice.select` and `Choice.bind` at branching point

The definition of `Choice` can change :

- concrete case: identity monad
- symbolic case, it must:
  - evaluates both branches
  - store the state (Wasm and PC)

Parallel, Symbolic, Choice Monad Implementation

Symbolic implementation is an actual choice monad made of:

- an error monad;
- a state monad;
- a coroutine monad.

Branches are explored in parallel thanks to OCaml 5!

Want More Details?

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

Simple Wasm example

```
(module
  (import "symbolic" "i32_symbol"
    (func $sym_i32 (result i32)))

  (func $start (local $x i32)
    (local.set $x (call $sym_i32))
    (if
      (i32.lt_s
        (i32.const 5)
        (local.get $x))
    (then unreachable)))

  (start $start)
)
```

Then, simply use `owi sym`:

```
$ owi sym file.wat
Trap: unreachable
model {
  symbol symbol_0 i32 6
}
Reached problem!
```

We also have:

- `owi conc` for concolic execution;
- `owi replay` to re-run and replace symbols by concrete values from a model.

C, C++ and Rust Symbolic Execution

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 main(void) {
  unsigned int x = owi_i32();
  unsigned int y = owi_i32();
  owi_assert(mean1(x, y) == mean2(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 cpp` 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

```
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.

Cross-Language Bug-Finding

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]
-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

Solver-Aided Programming

Polynomial Example

```
#include <owi.h>

class Poly {
private: int poly;
public:
  Poly(int a, int b, int c, int d){
    int x = owi_i32();
    int x2 = x * x;
    int x3 = x2 * x;
    poly = a*x3 + b*x2 + c*x + d; }
  int hasRoot() const {
    return poly == 0; } };

int main() {
  Poly p(1, -7, 14, -8);
  owi_assert(not(p.hasRoot())); }
```

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 leading tone resolves to the tonic
- instruments form chords
- no parallel fifths or octaves

Towards Proofs

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).

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

```c
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 targetting 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.

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.

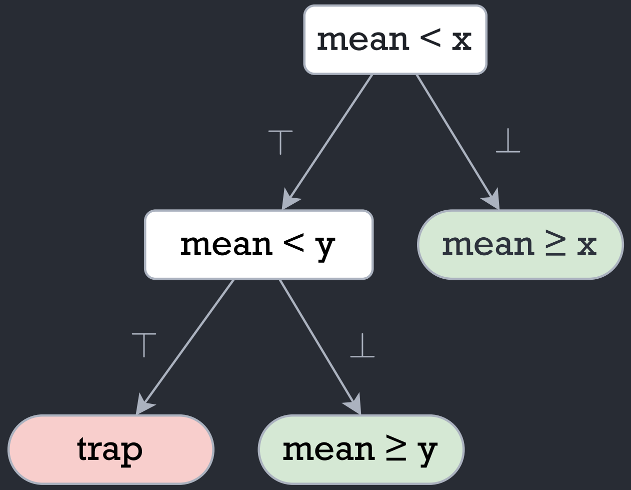
Tool	Bug found	Timeout	Bug not found
KLEE	782	368	65
Owi	676	539	0
Symbiotic	489	657	69

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.

Current goals

- support Wasm GC to handle Java, OCaml and Guile
- add heuristics to explore interesting paths first
- proper symbolic system interface
- automatic harness generation
- complex coverage-critera test-case generation (MCDC)
- better error reporting (editor integration)

OCaml PRO

We want to explore industrial applications and welcome discussions with users interested in Owi, as well as R&D on Wasm and programming languages. We are welcoming interns and can co-supervize PhD students.

Bonus

No bonus this time!