Cross-Language Symbolic Runtime Annotation Checking

Zhicheng Hui, Léo Andrès

Cross-Language Symbolic Runtime Annotation Checking

Zhicheng Hui[a,b] and Léo Andrès[a,c]

[a]OCamlPro SAS, 21 rue de Châtillon, 75014, Paris, France
[b]École Polytechnique, Institut Polytechnique de Paris, 91128, Palaiseau, France
[c]Université Paris-Saclay, CNRS, ENS Paris-Saclay, Inria, Laboratoire Méthodes Formelles, 91190, Gif-sur-Yvette, France

Software security has become a highly focused area nowadays. One key problem in this field is to expose software vulnerabilities effectively. Traditional testing methods have shown inadequacy in terms of path coverage and bug detection, due to their reliance on concrete input values. In this paper, we present the idea of combining symbolic execution with runtime annotation checking. By using symbolic input values instead of concrete ones, the specified program properties can be checked on all the corner cases.

We introduce two implementations of symbolic runtime annotation checkers: one for C, using the ANSI/ISO C Specification Language, and one for WebAssembly, using the Weasel specification language designed by us. Our approach combines the ease of use and the expressiveness of runtime annotation checking, as well as the power of program analysis.

1 Introduction

Assertions are widely used in programs. They are capable of stating and checking the desired program properties, such as the range of function arguments, the result of test cases, or loop invariants, etc. Many programming languages have assertions as built-in primitive, or as part of the standard library [CR06]. The desired program properties should be encoded as boolean values within the programming language, upon which assertions are applied.

Working directly with assertions can be tedious, as it requires mixing code written for functionality with code written for assertion checking. Moreover, programming language may not be best suited for specifying properties in terms of expressiveness and conciseness. Therefore, one way to enhance using assertions is Runtime Annotation Checking (RAC), a formal method to check annotations written in a specification language at runtime [BS24]. Using specification languages brings several benefits. They provide better interfacing with respect to the task of RAC, thanks to their non-invasive style, i.e. do not having to touch the source code, of writting formal specifications, and the possibility of generating monitored code automatically. Nowadays, there exist many behavioral interface specification languages designed for mainstream programming languages, such as the ANSI/ISO C Specification Language (ACSL) [Bau+], Java Modeling Language (JML) [LBR99], Pearlite [DJM21] for Rust, and Gospel [Cha+19] for OCaml.

Much as specification languages facilitate checking properties at runtime, they cannot be used to show the absence of property violations, because annotations are still checked on concrete values during execution. This approach is as opposed to deductive verification,

which guarantees properties hold for any program execution. For that reason, we propose to combine RAC with the technique of symbolic execution. Instead of concrete inputs, it allows us to check annotations on symbolic values, which are considered as implicitly universally quantified over the annotation. This combination augments the power of RAC. Furthermore, we demonstrate in Section 3.3 and 3.4 that having symbolic values can also optimize the code generated for RAC, and extend the specification language by releasing the constraint on bounded quantification.

We implement symbolic runtime annotation checkers for C and WebAssembly (Wasm) [Haa+17] programs, respectively. Our work is based upon Owi [And+21], a state-of-the-art symbolic interpreter [And+24] for Wasm. It can also work on C and Rust via Wasm compilation, making its symbolic execution engine reusable for cross-language programs. We integrate into Owi two specification languages, the Executable ANSI/ISO C Specification Language (E-ACSL) [Sig21] and the WEbAssembly SpEcification Language (Weasel), the latter being designed on our own. Together with dedicated code generators, the whole toolchain can be used to generate executable symbolic annotations from annotated C and Wasm programs: the program is first processed by the RAC code generator, which parses the formal specifications and instruments the program with additional codes checking the specified properties. The instrumented program is then compiled to Wasm, so as to perform symbolic execution using Owi.

Inspired by ACSL and based on the Wasm custom annotations extension, Weasel is the first standard-compliant Wasm specification language, meaning that Weasel annotations are a part of valid Wasm programs. The syntax and implementation of Weasel will be elaborated in Section 4.2. The support to multiple specification languages within a single symbolic execution tool allows for cross-language Symbolic Runtime Annotation Checking (SRAC), for instance in the context of a C program calling functions exported from a Wasm module.

The contributions of our work are summarized as follows:

- Methods of using symbolic values to optimize code generated for RAC.
- Methods of using symbolic values to lift the constraint on bounded quantification in specification languages used for RAC.
- Implementation of the first cross-language SRAC tool[1].
- Design and implementation of Weasel, the first standard-compliant specification language for Wasm.

In the rest of this paper, we will first present symbolic execution of Wasm and C using Owi in Section 2. In Section 3, we showcase how we generate symbolically executable annotations from C programs annotated with E-ACSL. We also demonstrate how the presence of symbolic values can evolve the design and implementation of E-ACSL. After that, in Section 4, we present the Weasel specification language for Wasm, including its design, its implementation along with an evaluation. We also present how we generate symbolically executable annotations from Wasm programs annotated with Weasel. In Section 5, we will discuss some related works on RAC and Wasm specification languages. Finally, Section 6 is devoted to conclusions and perspectives.

2 Symbolic Execution of Wasm and C code

The Wasm symbolic interpreter Owi serves as the backend of our symbolic runtime annotation checker. We provide essential background information to better understand the functioning of Owi, focusing on the target language Wasm, and the technique of symbolic execution. We

[1]The tool is integrated in Owi, https://github.com/ocamlpro/owi

```
(module
  (memory 1)
  (import "symbolic" "assert" (func $owi_assert (param i32)))
  (func $plus_three (param $x i32) (result i32)
    ;; [ ]: the stack is empty at the beginning
    local.get 0 ;; [ x ]: the function parameter $x is added to the
    ↪ stack
    i32.const 3 ;; [ 3 ; x ]: the i32 number 3 is added to the stack
    i32.add     ;; [ 3 + x ]: the two i32 numbers are poped from the
    ↪ stack, added together, the result is pushed back to the stack
    ;; [ 3 + x ]: the stack contains one value of type i32 at the end,
    ↪ this value is returned by the function
  )
  (func $start
    ;; [ ]: the stack is empty at the beginning
    i32.const 42     ;; [ 42 ]: the i32 number 42 is added to the
    ↪ stack
    call $plus_three ;; [ 45 ]: the function $plus_three is called,
    ↪ consuming the i32 number 42 and returning the i32 number 45
    i32.const 45     ;; [ 45 ; 45 ]: the i32 number 45 is added to the
    ↪ stack
    i32.eq           ;; [ 1 ]: the two i32 numbers are poped from the
    ↪ stack, checked for equality, the result (represented by a i32
    ↪ number) is pushed back to the stack
    call $owi_assert ;; [ ]: the function $owi_assert is called,
    ↪ consuming the i32 number 1
    ;; [ ]: the stack is empty at the end, no value is returned
  )
)
```

Listing 1. Example of a Wasm module in text format.

proceed by exemplifying using Owi as a symbolic execution engine without the intervention of specification languages.

2.1 Introduction to Wasm

As its name suggests, WebAssembly is intended to resemble assembly languages. It is a binary instruction format designed to become the new standard for the web. Featuring portability, efficiency, and safety, it is especially useful as a compilation target for high-level programming languages, such as C, Rust, or OCaml [ACF23], rather than being written manually.

A Wasm program is organized into modules, each module corresponds to a separate file, which can take either binary format (.wasm extension) or a human readable text format (.wat extension). The two formats are different representations of one same module. Each module contains its local definitions of global variables, types, functions, memories (vectors of bytes), and tables (vectors of references)[2]. Modules can communicate through imports and exports. Listing 1 shows an example of a Wasm module in text format.

In this example, the first clause `(memory 1)` defines a linear memory whose internal representation is a vector of bytes. The initial size of this memory is indicated by the number

[2]For a detailed documentation, see https://webassembly.github.io/spec/core/syntax/modules.html#syntax-table

1, *i.e.* one page of 64 KB. Memories can be accessed by load and store instructions with a memory address and value type.

The second clause imports the `"assert"` function defined in the module `"symbolic"` under the name *£owi_assert* and specifies its type: [i32] → []. There are four primitive types in Wasm, namely i32, i64, f32 and f64, for 32-bit integer, 64-bit integer, 32-bit floating-point number and 64-bit floating-point number, respectively.

The third and forth clauses define two functions *£plus_three* and *£start*. Each function definition is composed of a name, a type, and a function body which is itself composed of a sequence of instructions. Functions have access to global and local variables, which can be referenced with zero-based indices or with symbolic identifiers. In the body of the function *£plus_three*, the first instruction *local.get* 0 gets the zero-th, hence the first local variable, which is the parameter *£x* of type i32.

The execution of Wasm instructions is based on a stack machine, where each instruction manipulates an implicit stack. The state transitions of the implicit stack are illustrated in the comments of Listing 1. Note how the states of the stack correspond to the function types

$$\text{£plus\_three} : [\text{i32}] \to [\text{i32}]$$

,

$$\text{£owi\_assert} : [\text{i32}] \to []$$

and

$$\text{£start} : [] \to []$$

.

The limited set of data types and stack-based instruction semantics facilitate the interpretation and property checking of Wasm programs.

2.2 Introduction to Symbolic Execution

Symbolic execution is a technique used to analyze all the possible execution paths of a program [Kin76]. Rather than using concrete input values, symbolic execution works with symbols, which represent all the possible input values. During execution, each time a conditional branch is met, the symbolic execution engine constructs a logical formula representing the necessary constraints on input values to enter that branch. The formulas constructed along the execution path form together the path condition, which represents the constraints on input values in order to realize a specific execution path.

The reachability of execution paths corresponds to the satisfiability of their path conditions. After an exhaustive search over all the execution paths, the symbolic execution engine calls a SMT solver, such as Z3 [MB08], CVC5 [Bar+22], or Alt-Ergo [Con+18]. An execution path is reachable if the SMT solver can find an initial value to each symbol which satisfies the path condition.

Similar to reachability, it is possible to verify assertions along the execution path. It suffices to check whether the current path condition implies the assertion with the SMT solver. Listing 2 illustrates the execution paths and path conditions of symbolic execution on a Wasm module:

In Wasm, *unreachable* denotes a trap which should not be reachable. The SMT solver checks if the path condition up to that point is unsatisfiable.

2.3 Symbolic Execution of Wasm Code in Owi

The symbolic interpreter Owi implements the aforementioned procedure. Apart from checking the satisfiability of path condition when trapped by *unreachable*, it also provides a module named `"symbolic"`, which contains facilities of symbolic execution. For example,

```
(module
  (import "symbolic" "assert" (func $owi_assert (param i32)))
  (func $f (param $x i32) (param $y i32)
    ;; if x > y
    (if (i32.gt_s (local.get $x) (local.get $y))
    (then
      ;; Path condition: x > y
      ;; assert x + 1 > y
      (call $owi_assert (i32.gt_s (i32.add (local.get $x) (i32.const 1))
        (local.get $y)))
      ;; SMT solver checks if x > y implies x + 1 > y
    )
    (else
      ;; Path condition: x <= y
      ;; if x >= y
      (if (i32.ge_s (local.get $x) (local.get $y))
      (then
        ;; Path condition x <= y ∧ x >= y
        ;; this path is unreachable
        unreachable
        ;; SMT solver checks if x <= y ∧ x >= y is unsatisfiable
      ))
    ))
  )
)
```

Listing 2. Execution paths and path conditions of symbolic execution on Wasm code.

```
(module
  (import "symbolic" "i32_symbol" (func $owi_i32 (result i32)))
  (import "symbolic" "assert" (func $owi_assert (param i32)))
  (func $f (param $x i32) (param $y i32)
    (if (i32.gt_s (local.get $x) (local.get $y))
    (then
      (call $owi_assert (i32.gt_s (i32.add (local.get $x) (i32.const 1))
        (local.get $y))))
    (else
      (if (i32.ge_s (local.get $x) (local.get $y))
      (then unreachable))
    ))
  )
  (func $start
    call $owi_i32
    call $owi_i32
    call $f
  )
  (start $start)
)
```

Listing 3. Symbolic execution of Wasm code using Owi.

`£i32_symbol` generates a symbol of type `i32`, and `£assert` checks an assertion. In Listing 3, we call the function `£f` on two `i32` symbols to perform symbolic execution.

In the case of running into an assertion violation or an `unreachable` trap, Owi gives the witnessing input values. Below shows the results of symbolic execution on Wasm code using Owi:

```
$ owi sym paths.wat --fail-on-assertion-only
Assert failure: (i32.gt (i32.add symbol_0 (i32 1)) symbol_1)
Model:
  (model
    (symbol_0 (i32 2147483647))
    (symbol_1 (i32 -2147483648)))
Reached problem!
$ owi sym paths.wat --fail-on-trap-only
Trap: unreachable
Model:
  (model
    (symbol_0 (i32 1085352552))
    (symbol_1 (i32 1085352552)))
Reached problem!
```

The significations of options `--fail-on-assertion-only` and `--fail-on-trap-only` are inferable from name. We can verify that the generated values correspond to witnesses of assertion violation and `unreachable` trap.

2.4 Symbolic Execution of C Code in Owi

Owi's primitives are exposed to C through the `owi.h` header file, enabling it to perform symbolic execution on C code in a manner similar to that of Wasm. The C program containing Owi primitives is then compiled to Wasm and executed in the same way.

Listing 4 shows using `owi.h` to perform symbolic execution of C code. The function `primes` implements the algorithm of the Sieve of Eratosthenes, setting all the elements in the array with prime index to one. We test if the algorithm computes all the primes within the range by running the algorithm on symbolic inputs and verify the results on an extra function `is_really_prime`.

By running Owi, we can confirm that the function behaves correctly for all the arrays of size bigger than two and smaller than `MAX_SIZE`.

```
$ owi c primes.c
All OK
```

3 Using E-ACSL for Symbolic Runtime Annotation Checking of C Code

We have seen that checking the behavior of the function `primes` requires an extra primality testing function. Generally, expressing properties in a programming language requires some work, and it would be easiler to write them as mathematical propositions. We demonstrate in this section how the behavioral interface specification language E-ACSL can enhance the runtime annotation checking of C code.

3.1 Introduction to ACSL and E-ACSL

ACSL is a specification language for C, while E-ACSL stands for the executable subset of ACSL. It is executable in the sense that some codes containing assertions can be generated

```c
#define MAX_SIZE 100

#include <owi.h>
#include <stdlib.h>

void primes(int *is_prime, int n) {
    for (int i = 0; i < n; ++i) is_prime[i] = 1;
    is_prime[0] = is_prime[1] = 0;
    for (int i = 2; i * i < n; ++i) {
        if (!is_prime[i]) continue;
        for (int j = i; i * j < n; ++j) is_prime[i * j] = 0;
    }
}

int is_really_prime(int n) {
  for (int i = 2; i < n; ++i)
    if (n % i == 0) return 0;
  return 1;
}

int main(void) {
    int *is_prime = malloc(MAX_SIZE * sizeof(int));
    int n = owi_i32();
    owi_assume(n >= 2 && n <= MAX_SIZE);
    primes(is_prime, n);
    for (int i = 2; i < n; ++i)
      if (is_prime[i]) owi_assert(is_really_prime(i));
      else owi_assert(!is_really_prime(i));
    free(is_prime);
    return 0;
}
```

Listing 4. Symbolic execution of C code using Owi.

```
/*@ requires 2 <= n <= MAX_SIZE;
    requires \valid(is_prime + (0 .. (n - 1)));
    ensures  \forall integer i; 0 <= i < n ==>
        (is_prime[i] <==>
            (i >= 2 && \forall integer j; 2 <= j < i ==> i % j != 0));
*/
void primes(int *is_prime, int n) {
    for (int i = 0; i < n; ++i) is_prime[i] = 1;
    is_prime[0] = is_prime[1] = 0;
    for (int i = 2; i * i < n; ++i) {
        if (!is_prime[i]) continue;
        for (int j = i; i * j < n; ++j) is_prime[i * j] = 0;
    }
}
```

Listing 5. Example of E-ACSL function contract.

in order to verify the properties specified in annotations, in the same way as asserting on the results of `is_really_prime` verifies the correctness of the function `primes`. We call these codes executable annotations.

E-ACSL is intergrated as a plug-in in Frama-C [Sig+12]. It takes as input an annotated C file and generates an instrumented C file containing executable annotations, which can be run to check program properties. The instrumented file behaves functionally equivalent to the original file as long as the properties are not violated, and gets aborted if any property violation is detected.

Annotations should be written in regular C comments starting with an `"@"`, there are various kinds of Annotations available in E-ACSL, such as function contracts, loop variants, loop invariants, data invariants, type invariants, and so on. In this paper, we only focus on function contracts, as they play the major role in a behavioral interface specification language.

Listing 5 shows the same `primes` function with E-ACSL annotations, this example takes inspiration from the paper [SKV17].

Surrounded by `/*@ ... */` is the function contract of `primes`. There are two kinds of clauses in the example. `requires` specifies the precondition of the function, which means the property should hold before entering the function. And `ensures` specifies the postcondition of the function, which is the property that should hold after the function returns. Consequently, the clauses in Listing 5 specify that:

- `n` should be between `2` and `MAX_SIZE` when the function is called.
- The memory locations between `is_prime` and `is_prime + n - 1` can be safely read and written when the function is called.
- The non-zero elements in `is_prime` should have prime indices after the function returns.

E-ACSL generates a function `__e_acsl_primes` to check the function contract. It contains in proper order the assertions checking preconditions, a call to the annotated function, and the assertions checking postconditions. The assertions utilize E-ACSL library functions declared in `eacsl.h`, which also take charge of recording assertion metadata, interacting with the E-ACSL memory model, etc. Examples of E-ACSL generated code can be found in Listing 7 and Listing 9.

```c
#define MAX_SIZE 100

#include <owi.h>
#include <stdlib.h>

/*@ requires 2 <= n <= MAX_SIZE;
    requires \valid(is_prime + (0 .. (n - 1)));
    ensures  \forall integer i; 0 <= i < n ==>
        (is_prime[i] <==>
            (i >= 2 && \forall integer j; 2 <= j < i ==> i % j != 0));
*/
void primes(int *is_prime, int n) {
    for (int i = 0; i < n; ++i) is_prime[i] = 1;
    for (int i = 2; i * i < n; ++i) {
        if (!is_prime[i]) continue;
        for (int j = i; i * j < n; ++j) is_prime[i * j] = 0;
    }
}

int main(void) {
    int *is_prime = malloc(MAX_SIZE * sizeof(int));
    int n = owi_i32();
    owi_assume(n >= 2 && n <= MAX_SIZE);
    primes(is_prime, n);
    free(is_prime);
    return 0;
}
```

Listing 6. Example of a faulty program with E-ACSL annotations, to be executed symbolically.

3.2 Symbolic Execution of E-ACSL Instrumented Code

Executable annotations generated by RAC tools are generally run on concrete values, which limits the guarantees on program correctness. It is possible to combine this approach with symbolic execution to run annotations on symbolic values, thus increasing the coverage of execution paths.

Listing 6 shows symbolic execution on a faulty `primes` function, where `0` and `1` are not marked as non-primer initially.

Running this program with Owi generates a symbolic model where an assertion fails for $n = 2$:

```
  $ owi c --e-acsl primes.c
Assert failure: false
Model:
  (model
    (symbol_0 (i32 2)))
Reached problem!
```

If we correct the function, we can see that symbolic execution on a E-ACSL instrumented program verifies the correctness of the function contract.

```
  $ owi c --e-acsl primes2.c
```

```
/*  int x = 1;
    int y = 2;
    int z = 3; */
int __gen_e_acsl_and;
__e_acsl_assert_data_t __gen_e_acsl_assert_data = {.values = (void *)0};
// ...
if (y > x) {
  // ...
  __gen_e_acsl_and = z > y;
}
else __gen_e_acsl_and = 0;
// ...
__e_acsl_assert(__gen_e_acsl_and, & __gen_e_acsl_assert_data);
```

Listing 7. Code generated by E-ACSL to check a logical conjunction.

```
Assert failure: false
All OK
```

To make E-ACSL compatible with symbolic execution, we had to adapt its execution environment. In particular, we had to reimplement the E-ACSL runtime library, such as those related to memory model and assertions, by the corresponding Owi mechanisms.

This approach brings several advantages:

- It allows to reuse already annotated code without having to write specific assertions for Owi.
- It makes it possible to verify a new code base without having to integrate assertions into the program.
- It facilitates the automatic generation of test harnesses from specifications.
- It opens the way to function-by-function verification in Owi, instead of analyzing an entire program at once.

3.3 Using Symbols to Generate Better Code

To calculate the truth value of an expression, the code generated by E-ACSL sometimes uses branches and indroduces additional variables. This multiplies the number of possible execution paths, and increases the complexity of program state. Listing 7 shows a case where E-ACSL uses a branch and an additional variable for the assertion `y > x && z > y`.

The aim of using branches is to simulate the behavior of short circuit operators, such that a *`false`* on the first subexpression prematurely terminates the calculation. However, this is not likely to happen during a symbolic execution. Moreover, though additional branches and variables do not make much difference for a concrete execution, they are critical performance-wise for a symbolic execution. The dual effect motivates an optimization on the code generated for a concrete RAC. For instance, Listing 8 shows the code optimized for SRAC.

It clearly has less branching, and thus is better for symbolic execution.

Some logical connectors in E-ACSL do not exist in C, such as the logical implication and logical equivalence. But since the set of C logical connectors are functionally complete, any quantifier-free assertion can be encoded as a single C boolean expression after a certain transformation.

```
/*  int x = owi_i32();
    int y = owi_i32();
    int z = owi_i32(); */
__e_acsl_assert_data_t __gen_e_acsl_assert_data = {.values = (void *)0};
// ...
__e_acsl_assert(y > x && z > y, & __gen_e_acsl_assert_data);
```

Listing 8. Code optimized by using symbols to check a logical conjunction.

```
int __gen_e_acsl_forall = 1;
int __gen_e_acsl_i = 0;
__e_acsl_assert_data_t __gen_e_acsl_assert_data = {.values = (void *)0};
while (1) {
  if (__gen_e_acsl_i <= 10) ; else break;
  {
    int __gen_e_acsl_valid_read;
    // ...
    if (*(a + __gen_e_acsl_i) <= 100) ;
    else {
      __gen_e_acsl_forall = 0;
      goto e_acsl_end_loop1;
    }
  }
  __gen_e_acsl_i ++;
}
e_acsl_end_loop1: ;
// ...
__e_acsl_assert(__gen_e_acsl_forall,& __gen_e_acsl_assert_data);
```

Listing 9. Code generated by E-ACSL to check an universal quantification.

It is possible to optimize code generated by assertions with quantifiers, too. Consider the following E-ACSL annotation, which states the first ten elements of an array are less than or equal to 100:

```
//@ assert \forall int i; 0 <= i <= 10 ==> a[i] <= 100;
```

To perform RAC, E-ACSL generates the code shown in Listing 9.

This loop is necessary for concrete execution. However, by introducing a symbol for the quantified variable `i`, we can encode this specification more efficiently. The generated code could be optimized as shown in Listing 10.

3.4 Using Symbols to Handle Unbounded Quantifiers

E-ACSL is limited to the executable subset of ACSL, notably with restrictions on quantifiers that must be bounded. This is because checking quantifications during a concrete execution involves enumerating all possible values, which becomes impractical if the quantifiaction is unbounded.

For example, the annotation below stating that the integer variable `x` is odd can not be written using E-ACSL:

```
//@ assert \forall int i; x != 2 * i;
```

```
int __e_acsl_i = owi_i32();
if (0 <= i && i <= 10) {
  owi_assert(a[i] <= 100);
}
```

Listing 10. Code optimized by using symbols to check an universal quantification.

```
int __e_acsl_i = owi_i32();
owi_assert(x != 2 * i);
```

Listing 11. Code checking unbounded quantifications by using symbols.

Running this with E-ACSL emits the following warning:

```
$ owi --e-acsl unbounded.c
...
Warning:
  E-ACSL construct `unguarded \forall quantification' is not yet supported.
  Ignoring annotation.
...
```

Symbolic execution allows some of these constraints to be lifted. Listing 11 shows how the above annotation can be checked by using symbols.

To perform this transformation in the general case, it is necessary to correctly handle existential quantifiers as well as nested quantifications. Since all the symbols are introuced in the program state, functions such as `model_exist(var, prop)` and `model_forall(var, prop)` should be added to be able to refer to a single symbol.

It is necessary to follow several steps:

- Convert the propositions to prenex normal form.
- Perform skolemization in the presence of existential quantifiers.
- Associate each quantifier with a symbol of the appropriate type.
- Encode the inner quantifier-free proposition using program expressions.
- Call functions `model_exist(var, prop)` and `model_forall(var, prop)` to iterate over leading quantifiers.
- Finally, assert on the result.

4 Introducing Weasel for Symbolic Runtime Annotation Checking of Wasm Code

Much as Wasm is often used as a compilation target, it is also common for Wasm code to be written manually, for example to create a runtime environment or to replace assembler segments. Being able to annotate and verify Wasm code in this context thus becomes useful. In this section, we present the design of a specification language dedicated to Wasm and the implementation of a runtime annotation checker based on that.

```
(module (@name "I'm a module with a fancy name")
  (@custom "my-fancy-section" "Hello, I'm useless don't try to understand
  ↪  me")
  (func (@inline) $lambda (@name "λ") (param $x (@name "α") i32) (result
  ↪  i32)
    (local.get $x)
  )
)
```

Listing 12. Example of custom annotations in Wasm.

```
(module
  (@contract $plus_three
    (ensures (= result (+ $x 3)))
  )
  (func $plus_three (param $x i32) (result i32)
    local.get $x
    i32.const 3
    i32.add
  )
  (func $start
    i32.const 42
    call $plus_three
    drop
  )
  (start $start)
)
```

Listing 13. Example of Weasel function contract.

4.1 The Custom Annotations Proposal

In order to annotate Wasm programs in the text format, we have chosen to implement the Wasm custom annotations proposal [par18] in Owi. It allows writting annotations while respecting the standards of Wasm. According to Wasm's specification, custom annotations can appear wherever a space is allowed, and will be ignored by all the tools unable to recognize them. They are similar to attributes in OCaml, and they always take the form of `(@annotation-id ...)` in Wasm. Listing 12 shows an example of custom annotations in Wasm.

4.2 Weasel: a WEbAssembly SpEcification Language

Weasel is a behavior specification language of Wasm inspired by ACSL. Listing 13 shows an example of a function contract written in Weasel. It specifies the function should return three plus the function parameter.

In order to be coherent with Wasm, Weasel utilizes a syntax of S-expression. Below shows a slightly simplified abstract syntax of Weasel:

$\langle ind \rangle$ ::= `$` *id* | *i32*

$\langle contract \rangle$ ::= `(@contract` *ind clause** `)`

⟨*clause*⟩ ::= `(` `requires` *prop* `)` | `(` `ensures` *prop* `)`

⟨*prop*⟩ ::= `(` *pprop* `)` | `true` | `false`

⟨*pprop*⟩ ::= *binpred term term* | *unconnect prop* | *binconnect prop* | *binder binder_type prop*

⟨*term*⟩ ::= `(` *pterm* `)` | *ind*

⟨*pterm*⟩ ::= `i32` *i32* | `i64` *i64* | `f32` *f32* | `f64` *f64* | `param` *ind* | `global` *ind* | `binder` *ind* | *binop term* | `result` *i32* | `memory` *term*

⟨*binpred*⟩ ::= `>=` | `>` | `<=` | `<` | `=` | `!=`

⟨*unconnect*⟩ ::= `!`

⟨*binconnect*⟩ ::= `&&` | `||` | `==>` | `<==>`

⟨*binder*⟩ ::= `forall` | `exists`

⟨*binder_type*⟩ ::= `i32` | `i64` | `f32` | `f64`

⟨*binop*⟩ ::= `+` | `-` | `*` | `/`

Firstly, *ind* represents the index of functions or variables. It can either be a zero-based index number or `"$"` followed by an identifier, which is again coherent with Wasm text format syntax.

Currently, Weasel specification focuses on function contracts, which are written immediately before function definitions. A function contract begins with `(@contract ind` and follows with zero or more clauses. Each clause can either begin with `requires` or `ensures`, specifying the preconditions and postconditions of the function.

The nonterminals *prop* and *pprop* represent propositions, *pprop* standing for parenthesized propositions. Propositions can be literals, predicates, connectors, or quantifications. Quantifiers are refered to by de Bruijin indices [de 72] in the above syntax, though using identifiers is also possible in our implementation.

The nonterminals *term* and *pterm* represent terms. Terms can be literals of primitive types, parameters, global variables, binder variables, operations, function return values, or memory contents.

4.3 Generating (Symbolic) Runtime Annotations From Weasel

We implement a code generator in Owi which generates executable annotations from Weasel specifications.

```
$ owi instrument plus_three.wat
```

This augments the original Wasm file with an extra function *`£__weasel_plus_three`*, which calls *`£plus_three`* and checks all the properties specified in annotations. Calls and exports of the annotated function are replaced with the monitored version, imposing a module-level abstraction of RAC. Listing 14 shows the code generated from Weasel annotations to check the function contract.

This file performs runtime annotation checking of Wasm functions, and is directly executable in a concrete environment as long as the host provides the function *`£assert`*. In addition to this, the `owi instrument` command also provides a `--symbolic` option that generates symbolically executable annotations, to be executed on symbolic values.

```
(module
  (import "symbolic" "assert" (func $assert (param i32)))
  (func $plus_three (param $x i32) (result i32)
    local.get $x
    i32.const 3
    i32.add
  )
  (func $start
    i32.const 42
    call $__weasel_plus_three
    drop
  )
  (func $__weasel_plus_three (param $x i32) (result i32)
    (local $__weasel_temp i32) (local $__weasel_res_0 i32)
    local.get $x
    call $plus_three
    local.set $__weasel_res_0
    local.get $__weasel_res_0
    local.get $x
    i32.const 3
    i32.add
    i32.eq
    call $assert
    local.get $__weasel_res_0
  )
  (start $start)
)
```

Listing 14. Code generated by Weasel to check a function contract.

```
struct list {
    int element;
    struct list *next;
};

/*@ ghost
int sorted(struct list* l) {
    if (l == NULL || l->next == NULL) return true;
    if (l->element > l->next->element) return false;
    return sorted(l->next);
}
*/

void merge_sort(struct list **headRef) {
    struct list *head = *headRef;

    if (is_empty(head) || is_empty(head->next)) return;

    struct list *l1, *l2;
    split(head, &l1, &l2);

    merge_sort(&l1);
    merge_sort(&l2);

    *headRef = merge(l1, l2);

    //@ ghost int p = sorted(*headRef);
    //@ assert p != 0;
}
```

Listing 15. C implementation of merge sort on a linked list, using functions from a Wasm module.

4.4 Evaluation

With the support of E-ACSL and Weasel, it is possible to generate our approach to cross-language programs. Listing 15 shows a C implementation of merge sort on a linked list, using list manipulating functions in a small hand-written Wasm module. These functions are declared in C by the `__attribute__` mechanism:

```
bool is_empty(struct list *) __attribute__((import_module("list"),
↪  import_name("is_empty")));
void push(struct list**, int) __attribute__((import_module("list"),
↪  import_name("push")));
void split(struct list*, struct list**, struct list**)
↪  __attribute__((import_module("list"), import_name("split")));
struct list* merge(struct list*, struct list*)
↪  __attribute__((import_module("list"), import_name("merge")));
```

This piece of C code carries E-ACSL annotation specifying its partial correctness by using ghost codes. And the underlying Wasm module functions are themselves annotated by Weasel function contracts. By compiling the E-ACSL instrumented C code to Wasm and linking it with the Weasel instrumented module functions, we can use Owi to check runtime annotations at different program levels symbolically.

Since Weasel is still in its prototype stage, there are some gaps in its ability to support truly practical and useful annotations. In particular, this requires integrating more Wasm constructs into Weasel, and implementing a memory model that checks the separation of C and Wasm program memories.

5 Related Work

Runtime Annotation Checking. Being a lightweight formal method, RAC dates back to assertions in programming languages, such as the extension of the Program Evaluator and Tester System for FORTRAN [SF75], and the C preprocessor macro `assert`. In recent decades, RAC tools typically rely on a specification language, and take charge of generating executable annotations from specifications. Apart from the example of E-ACSL, the tool OpenJML [Cok11] is a RAC implementation of the JML specification language.

RAC is often used together with fuzz testing. The tool JMLKelinci+ [Nil+24] combines it with a coverage-guided fuzzing tool in order to cover branches with valid inputs and detect semantic bugs. The OCaml runtime annotation checker Ortac [FP21] also comes with a fuzzing frontend which tests the program with random inputs in the hope to detect assertion violations. However, since fuzz testing is by nature incomplete, none of these works can provide guarantee on program properties as using symbolic execution.

It is also possible to combine RAC with static analysis. In the work of Julien Signoles [KS13], two static analyzers, WP [Bla+24] and VALUE [Büh17], are combined with the E-ACSL plug-in in Frama-C. It works by discharging some verification conditions with static analyzers, and let E-ACSL take care of the remaining part.

WebAssembly Specification Languages. Currently, there are not many specification languages for Wasm. The work of Munuera Mazzaro [Mun23] proposes VerifiWasm, where specifications are written in a separate file. Their tool verifies that the specifications are respected for a given Wasm binary. Similar to our approach, they use symbolic execution to generate verification conditions that are later transmitted to SMT solvers.

However, their method has several limitations compared to ours. First, they do not produce a Wasm module integrating executable assertions from the original program. Moreover, their system is limited to binary modules distinct from the specifications, which complicates the implementation and maintenance. Their implementation is also restricted to a small part of the Wasm standard: integer operations are incomplete, floating-point numbers are not supported, and global variable and memory management are absent. These limitations contrast sharply with Owi, which offers much broader support.

Finally, their tool supports only one SMT solver (Z3), while Owi is able to interact with multiple solvers, thanks to the use of Smtml [PMA23], a multi back-end front-end for SMT Solvers in OCaml. This offers greater flexibility in terms of solving assertion formulas.

In addition to this, there have been some works focusing on the Wasm specification itself. The work of Conrad Watt [Wat18] presents a mechanised Isabelle specification for the WebAssembly language. Based upon that, a verified executable interpreter and type checker are implemented in the proof assistant Isabelle. This facilitates the specification of Wasm language, as well as the analysis and verification of Wasm programs. SpecTec [You+24] is a domain-specific language to express Wasm's formal semantics. Various backends have been integrated into this tool, which generates different styles of spefication documentds, and gives a reliable interpretation of Wasm programs. They can help with the standardisation of future Wasm features, as well as validating existing specifications.

6 Conclusion and Perspectives

In this paper, we present how to combine symbolic execution and RAC, with the help of specification languages. Compared to monitoring a concrete program execution, the

introduction of symbols increases the coverage of execution path, hence strengthens the power of verifying program properties. We have also shown how to optimize code for RAC in the presence of symbols, and how to release the restriction on bounded quantifiers.

We back up the idea of SRAC with two implementations: C and Wasm, using E-ACSL and Weasel respectively. These two tools are currently integrated in the tool Owi. We demonstrate specification languages make it possible to specify richer set of program properties, and to verify a new code base without having to integrate assertions into the program. Moreover, the notion of function contracts boosts modular function-by-function verification.

We present the design and implementation of Weasel, the first standard-compliant specification language for Wasm program. It is based on the custom annotations extension, which makes Weasel annotations portable to text and binary format. We imitate the approach of E-ACSL for code generation, and have integrated a small set of core Wasm instructions.

There are several possible directions to continue our work. First, we plan to improve the code generation of E-ACSL. This could increase the efficiency of our tool. Second, we plan to continue the development of Weasel, which is still at an early stage. Many improvements remain to be made, in particular to extend the specification language to a wider range of Wasm constructs.

Another interesting perspective concerns the verification of code combining several languages, each annotated with its own specifications. For example, it would be relevant to verify an optimized library written in Wasm, embedded in a C algorithm, where the correctness proofs of one depend on the assertions generated from the specifications of the other.

Furthermore, we plan to apply this approach to other languages, such as Rust. By reusing Pearlite, a deductive verification tool for Rust, and its specification language Creusot, it would be possible to generate instrumented code executable via Owi.

Finally, another promising path would be to compile specification languages, such as those used in C, to Weasel, which would allow unifying verification tools for different languages through a common infrastructure.

References

[ACF23] Léo Andrès, Pierre Chambart, and Jean-Christophe Filliâtre. "Wasocaml: compiling OCaml to WebAssembly". In: *IFL 2023 - The 35th Symposium on Implementation and Application of Functional Languages*. João Saraiva and João Fernandes. Braga, Portugal, Aug. 2023. URL: `https://inria.hal.science/hal-04311345`.

[And+21] Léo Andrès et al. *Owi*. 2021. URL: `https://github.com/ocamlpro/owi`.

[And+24] Léo Andrès et al. "Owi: Performant Parallel Symbolic Execution Made Easy, an Application to WebAssembly". In: *The Art, Science, and Engineering of Programming* 9.2 (Oct. 2024). URL: `https://hal.science/hal-04627413`.

[Bar+22] Haniel Barbosa et al. "cvc5: A Versatile and Industrial-Strength SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Dana Fisman and Grigore Rosu. Cham: Springer International Publishing, 2022, pp. 415–442. ISBN: 978-3-030-99524-9.

[Bau+] Patrick Baudin et al. *ACSL: ANSI/ISO C Specification Language*. URL: `http://frama-c.com/acsl.html`.

[Bla+24] Allan Blanchard et al. "Formally Verifying that a Program Does What It Should: The WpWp (Frama-C plug-in) Plug-in". In: *Guide to Software Verification with Frama-C: Core Components, Usages, and Applications*. Ed. by Nikolai Kosmatov, Virgile Prevosto, and Julien Signoles. Cham: Springer International Publishing, 2024, pp. 187–261. ISBN: 978-3-031-55608-1. DOI: 10.1007/978-3-031-55608-1_4. URL: https://doi.org/10.1007/978-3-031-55608-1_4.

[BS24] Thibaut Benjamin and Julien Signoles. "Runtime Annotation Checking with Frama-C: The E-ACSLE-ACSL (Frama-C plug-in) Plug-in". In: *Guide to Software Verification with Frama-C: Core Components, Usages, and Applications*. Ed. by Nikolai Kosmatov, Virgile Prevosto, and Julien Signoles. Cham: Springer International Publishing, 2024, pp. 263–303. ISBN: 978-3-031-55608-1. DOI: 10.1007/978-3-031-55608-1_5. URL: https://doi.org/10.1007/978-3-031-55608-1_5.

[Büh17] David Bühler. "EVA, an Evolved Value Analysis for Frama-C : structuring an abstract interpreter through value and state abstractions". 2017REN1S016. PhD thesis. 2017. URL: http://www.theses.fr/2017REN1S016/document.

[Cha+19] Arthur Charguéraud et al. "GOSPEL -Providing OCaml with a Formal Specification Language". In: *FM 2019 - 23rd International Symposium on Formal Methods*. Porto, Portugal, Oct. 2019. URL: https://inria.hal.science/hal-02157484.

[Cok11] David R. Cok. "OpenJML: JML for Java 7 by Extending OpenJDK". In: *NASA Formal Methods*. Ed. by Mihaela Bobaru et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 472–479. ISBN: 978-3-642-20398-5.

[Con+18] Sylvain Conchon et al. "Alt-Ergo 2.2". In: *SMT Workshop: International Workshop on Satisfiability Modulo Theories*. Oxford, United Kingdom, July 2018. URL: https://inria.hal.science/hal-01960203.

[CR06] Lori A. Clarke and David S. Rosenblum. "A historical perspective on runtime assertion checking in software development". In: *SIGSOFT Softw. Eng. Notes* 31.3 (May 2006), pp. 25–37. ISSN: 0163-5948. DOI: 10.1145/1127878.1127900. URL: https://doi.org/10.1145/1127878.1127900.

[de 72] N.G de Bruijn. "Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem". In: *Indagationes Mathematicae (Proceedings)* 75.5 (1972), pp. 381–392. ISSN: 1385-7258. DOI: https://doi.org/10.1016/1385-7258(72)90034-0. URL: https://www.sciencedirect.com/science/article/pii/1385725872900340.

[DJM21] Xavier Denis, Jacques-Henri Jourdan, and Claude Marché. *The CREUSOT Environment for the Deductive Verification of Rust Programs*. Research Report RR-9448. Inria Saclay - Île de France, Dec. 2021. URL: https://inria.hal.science/hal-03526634.

[FP21] Jean-Christophe Filliâtre and Clément Pascutto. "Ortac: Runtime Assertion Checking for OCaml (Tool Paper)". In: *Runtime Verification*. Ed. by Lu Feng and Dana Fisman. Cham: Springer International Publishing, 2021, pp. 244–253. ISBN: 978-3-030-88494-9.

[Haa+17] Andreas Haas et al. "Bringing the web up to speed with WebAssembly". In: *ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2017.

[Kin76] James C. King. "Symbolic execution and program testing". In: *Commun. ACM* 19.7 (July 1976), pp. 385–394. ISSN: 0001-0782. DOI: 10.1145/360248.360252. URL: https://doi.org/10.1145/360248.360252.

[KS13] Nikolai Kosmatov and Julien Signoles. "A Lesson on Runtime Assertion Checking with Frama-C". In: *Runtime Verification*. Ed. by Axel Legay and Saddek Bensalem. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 386–399. ISBN: 978-3-642-40787-1.

[LBR99] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. "JML: A Notation for Detailed Design". In: *Behavioral Specifications of Businesses and Systems*. Ed. by Haim Kilov, Bernhard Rumpe, and Ian Simmonds. Boston, MA: Springer US, 1999, pp. 175–188. ISBN: 978-1-4615-5229-1. DOI: 10.1007/978-1-4615-5229-1_12. URL: https://doi.org/10.1007/978-1-4615-5229-1_12.

[MB08] Leonardo de Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by C. R. Ramakrishnan and Jakob Rehof. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340. ISBN: 978-3-540-78800-3.

[Mun23] David Munuera Mazarro. "Specification and verification of WebAssembly programs". Unpublished. July 2023. URL: https://oa.upm.es/75802/.

[Nil+24] Amirfarhad Nilizadeh et al. "JMLKelinci+: Detecting Semantic Bugs and Covering Branches with Valid Inputs Using Coverage-guided Fuzzing and Runtime Assertion Checking". In: *Form. Asp. Comput.* 36.1 (Mar. 2024). ISSN: 0934-5043. DOI: 10.1145/3607538. URL: https://doi.org/10.1145/3607538.

[par18] WebAssembly Community Group participant. *Custom Annotations Proposal*. 2018. URL: https://github.com/WebAssembly/annotations.

[PMA23] João Pereira, Filipe Marques, and Pedro Adão. *Smtml*. 2023. URL: https://github.com/formalsec/smtml.

[SF75] Leon G. Stucki and Gary L. Foshee. "New assertion concepts for self-metric software validation". In: *Proceedings of the International Conference on Reliable Software*. Los Angeles, California: Association for Computing Machinery, 1975, pp. 59–71. ISBN: 9781450373852. DOI: 10.1145/800027.808425. URL: https://doi.org/10.1145/800027.808425.

[Sig+12] Julien Signoles et al. "Frama-c: a Software Analysis Perspective". In: vol. 27. Oct. 2012. DOI: 10.1007/s00165-014-0326-7.

[Sig21] Julien Signoles. "The e-ACSL perspective on runtime assertion checking". In: *Proceedings of the 5th ACM International Workshop on Verification and MOnitoring at Runtime EXecution*. VORTEX 2021. Virtual, Denmark: Association for Computing Machinery, 2021, pp. 8–12. ISBN: 9781450385466. DOI: 10.1145/3464974.3468451. URL: https://doi.org/10.1145/3464974.3468451.

[SKV17] Julien Signoles, Nikolai Kosmatov, and Kostyantyn Vorobyov. "E-ACSL, a runtime verification tool for safety and security of C programs (tool paper)". In: *RV-CuBES 2017-International Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools*. 2017.

[Wat18] Conrad Watt. "Mechanising and verifying the WebAssembly specification". In: *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*. CPP 2018. Los Angeles, CA, USA: Association for Computing Machinery, 2018, pp. 53–65. ISBN: 9781450355865. DOI: 10.1145/3167082. URL: https://doi.org/10.1145/3167082.

[You+24] Dongjun Youn et al. "Bringing the WebAssembly Standard up to Speed with SpecTec". In: *Proceedings of the ACM on Programming Languages* 8 (June 2024), pp. 1559–1584. DOI: 10.1145/3656440.