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

One key challenge in ensuring software security is the effective detection of vulnerabilities. Traditional testing methods often fall short in path coverage and bug detection due to their reliance on concrete input values. In this paper, we explore the combination of symbolic execution with runtime annotation checking. By employing symbolic input values, rather than concrete ones, our tool is capable of verifying specified program properties across all the potential corner cases within the power of the underlying solver.

We present 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, which we designed. Our approach leverages the ease of runtime annotation checking with the analytical power of symbolic execution.

1 Introduction

Assertions are widely used to verify desired properties in software, such as function argument ranges, test case outcomes, and loop invariants. Many programming languages support assertions natively or through libraries [CR06], encoding program properties as boolean expressions evaluated by assertions.

However, writing assertions directly within code can be cumbersome, as it mixes functional logic with verification logic. Additionally, the expressiveness of programming languages may limit how effectively properties can be specified.

One solution is to express program properties by annotations written in a specification language, then verify the annotations at runtime after compiling them to program assertions or other verification functionalities offered by an external library. Such a formal method is often referred to as Runtime Annotation Checking (RAC) [BS24]. Specification languages offer several advantages, including non-intrusive integration with code and the ability to automatically generate monitored code. Various behavioral interface specification languages are available for popular languages, such as ACSL for C [Bau+], JML for Java [LBR99], Pearlite for Rust [DJM21], and Gospel for OCaml [Cha+19].

Despite the benefits of RAC, it is limited to concrete input values, meaning it cannot guarantee the absence of property violations across all inputs. To enhance RAC, we propose combining it with symbolic execution, allowing annotations to be checked against symbolic, rather than concrete, values. During symbolic execution, the program executes on symbolic values. The specified program properties containing symbolic values can be seen as universally

quantified over all the input values, significantly strengthening RAC. Additionally, as we demonstrate in Sections 3.3 and 3.4, symbolic execution can optimize RAC-generated code and extend specification languages by relaxing constraints on bounded quantification[1].

We implemented symbolic runtime annotation checkers for both C and WebAssembly (Wasm) [Haa+17], building on Owi [And+21], a state-of-the-art symbolic interpreter [And+24] for Wasm. Owi supports C and Rust through Wasm compilation, making its symbolic execution engine applicable across languages. We integrated two specification languages: Executable ACSL (E-ACSL) [Sig21] for C, and Weasel, a Wasm specification language we developed. This toolchain processes annotated C and Wasm programs, generating executable symbolic annotations through an RAC code generator. The resulting instrumented code is then compiled to Wasm (in the case of a C program) for symbolic execution in Owi.

Inspired by ACSL and leveraging Wasm's custom annotations extension, Weasel is the first standard-compliant Wasm specification language. It integrates seamlessly into Wasm programs, making Weasel annotations part of valid Wasm code. Section 4.2 details its design and implementation. The cross-language support in our symbolic execution tool enables Symbolic Runtime Annotation Checking (SRAC) across languages, such as when C programs call functions from Wasm modules.

Our contributions include:

- methods for optimizing code generated for RAC using symbolic values;
- techniques for relaxing bounded quantification constraints in RAC specification languages using symbolic values;
- development of the first cross-language SRAC tool[2];
- design and implementation of Weasel, the first standard-compliant Wasm specification language.

The remainder of this paper is structured as follows. Section 2 presents symbolic execution for Wasm and C using Owi. Section 3 discusses generating symbolically executable annotations from C programs annotated with E-ACSL and explores how symbolic values shape the design and implementation of E-ACSL. Section 4 introduces the Weasel specification language for Wasm, detailing its design, and implementation, along with how to perform SRAC on Weasel-annotated Wasm programs. Section 5 reviews related research on RAC and Wasm specification languages. Finally, Section 6 concludes the paper and discusses future directions.

2 Symbolic Execution of Wasm and C code

The Wasm symbolic interpreter Owi acts as the backend for our symbolic runtime annotation checker. To understand Owi's role, we provide an overview of Wasm and the symbolic execution technique. We then demonstrate how Owi functions as a symbolic execution engine.

2.1 Introduction to Wasm

Wasm is designed to resemble an assembly language. It is a binary instruction format, intended to become a standard for the web, offering portability, efficiency, and safety. Wasm is especially suited as a compilation target for high-level languages like C, Rust, and OCaml [ACF23], rather than being written manually.

[1]Section 3.4 explains why quantified variables have to be bounded in RAC tools.
[2]The tool is integrated into Owi, https://github.com/ocamlpro/owi

```
(module
  (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 popped from the
    ↪  stack, added together, and 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
  )
)
```

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

A Wasm program is organized into modules, with each module represented either in binary (`.wasm`) or human-readable text (`.wat`) format. Each module contains local definitions for global variables, types, functions, etc[3]. Modules can communicate through importing and exporting functions. Listing 1 shows an example of a Wasm module in text format.

Functions in Wasm module access global and local variables using numerical indices or symbolic identifiers. Wasm uses a stack-based execution model, where instructions manipulate an implicit stack. The comments of Listing 1 illustrates how Wasm instructions operate the program stack.

The simplicity of Wasm's data types and stack-based semantics makes it well-suited for interpretation and property checking.

2.2 Introduction to Symbolic Execution

Symbolic execution is a technique for analyzing all possible execution paths of a program [Kin76]. Instead of using concrete values, symbolic execution employs symbols to represent all potential inputs. When a conditional branch is encountered, the symbolic execution engine builds a logical formula representing the conditions required to take that branch. These formulas accumulate along each path, forming the path condition.

An execution path is considered reachable if the path condition is satisfiable. After exploring all paths, an SMT solver, such as Z3 [MB08], CVC5 [Bar+22], or Alt-Ergo [Con+18], checks whether the path condition can be satisfied by specific input values.

Symbolic execution also allows for verifying assertions. If an assertion fails, the SMT solver checks whether the current path condition contradicts the assertion. The comments of the function `$f` in Listing 2 illustrate examples of execution paths and path conditions of symbolic execution.

In Wasm, the `unreachable` instruction signals that a path should not be executed. The SMT solver verifies whether the corresponding path condition is unsatisfiable.

2.3 Symbolic Execution of Wasm Code in Owi

The above Wasm module imports two functions `$owi_i32` and `$owi_assert` from the `symbolic` module we developed. `$owi_i32` returns a symbol representing a 32-bit integer, and `$owi_assert` takes an argument (an expression possibly containing symbols) and checks its satisfiability via SMT solvers. The code is executed on Owi's Wasm interpreter, in the

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

```
(module
  (import "symbolic" "owi_i32" (func $owi_i32 (result i32)))
  (import "symbolic" "owi_assert" (func $owi_assert (param i32)))
  (func $f (param $x i32) (param $y i32)
    (if (i32.gt_s (local.get $x) (local.get $y)) ;; if x > y
    (then
      ;; Path condition: x > y
      (call $owi_assert (i32.gt_s (i32.add (local.get $x) (i32.const 1))
        (local.get $y))) ;; assert x + 1 > y
      ;; SMT solver checks if x > y implies x + 1 > y
    )
    (else
      ;; Path condition: x <= y
      (if (i32.ge_s (local.get $x) (local.get $y)) ;; if x >= y
      (then
        ;; Path condition x <= y ∧ x >= y
        unreachable ;; this path is unreachable
        ;; SMT solver checks if x <= y ∧ x >= y is unsatisfiable
      ))
    ))
  )
  (func $start
    call $owi_i32 ;; generate a symbol representing a 32-bit integer
    call $owi_i32 ;; generate another symbol representing a 32-bit integer
    call $f ;; call the function $f on two symbolic values
  )
)
```

Listing 2. Execution paths and path conditions of symbolic execution.

same way it would be on a regular interpreter, but with the difference that it is going to execute all reachable paths and manipulate symbols in addition to concrete values in its program stack.

When Owi encounters an assertion violation or an `unreachable` trap, it outputs a model containing values of symbols that can cause the issue. Symbols are indexed by the order they are created. Below are the results of symbolic execution using Owi, where `paths.wat` stores the Wasm module in Listing 2:

```
$ 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 command-line options `--fail-on-assertion-only` and `--fail-on-trap-only` instruct Owi to only report assertion violations and traps, respectively.

2.4 Symbolic Execution of C Code in Owi

The above approach also applies to C code. Owi implements the `owi.h` header file declaring C functions for creating symbols and making assertions, namely `owi_i32` and `owi_assert`. C programs can then use these functions to make assertions on expressions containing symbols. In order to perform symbolic execution, Owi compiles the C program to Wasm, leveraging Owi's Wasm interpreter.

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

Expressing the desired program properties in C code with functions such as `owi_i32` and `owi_assert` can be tedious. Often, it is easier to write them directly as mathematical propositions. This section illustrates how the behavioral interface specification language E-ACSL enhances runtime annotation checking for C code.

3.1 Introduction to ACSL and E-ACSL

ACSL (ANSI/ISO C Specification Language) is a specification language for C, and E-ACSL (Executable ACSL) represents its executable subset, it allows the generation of executable annotations—code containing assertions that can be run to verify program properties.

E-ACSL is integrated as a plug-in in Frama-C [Sig+12], a framework for C program analysis. It takes an annotated C file as input and outputs an instrumented C file containing executable annotations. The instrumented file behaves equivalently to the original, except that it will abort if any property violations are detected.

E-ACSL supports various annotations, such as function contracts, loop invariants, and data invariants. Annotations are written in regular C comments using `/*@ ... */`. Here, we focus on function contracts, which specify the conditions under which a function operates. The two most important clauses in E-ACSL function contract are `requires`, which specifies the preconditions that must hold before the function is executed, and `ensures`, which specifies the postconditions that must hold after the function returns.

For each function contract, E-ACSL generates a new function in the instrumented file. Each function first checks the preconditions, then calls the original annotated function, and finally checks the postconditions.

The code checking preconditions and postconditions makes use of E-ACSL's runtime library, whose interface is declared in `eacsl.h`. Below are some mostly used runtime library functions of E-ACSL:

- `__e_acsl_assert`, which takes an argument and checks if it evaluates to *true*.
- `__e_acsl_assert_register_int`, `__e_acsl_assert_register_long`, etc. They are used to register certain metadata about an assertion. The metadata is passed into the `__e_acsl_assert` function to provide an informative error message in case of assertion violation.
- `__e_acsl_store_block`, `__e_acsl_valid_read`, `__e_acsl_delete_block`, etc. They are used to check annotations related to memory properties. For example, the function `__e_acsl_store_block` makes E-ACSL mark a memory block as allocated, `__e_acsl_valid_read` checks if a memory block (expressed as a C pointer) is valid to read, and `__e_acsl_delete_block` makes E-ACSL mark a memory block as freed.

```
int __gen_e_acsl_and;
if (y > x) __gen_e_acsl_and = z > y;
else __gen_e_acsl_and = 0;
__e_acsl_assert(__gen_e_acsl_and);
```

Listing 3. Code generated by E-ACSL to check a logical conjunction.

```
__e_acsl_assert(y > x && z > y);
```

Listing 4. Code optimized by using symbols to check a logical conjunction.

3.2 Symbolic Execution of E-ACSL Instrumented Code

Traditionally, executable annotations are checked on concrete values. By combining RAC with symbolic execution, it becomes possible to verify annotations on symbolic values, thus covering more execution paths. To make code generated by E-ACSL compatible with symbolic execution, we adapted E-ACSL's runtime to work with Owi's symbolic mechanisms. Concretely, we made the following adaptations:

- `__e_acsl_assert` directly calls Owi's assert function `owi_assert`.
- Functions registering assertion metadata are kept unchanged to retain informative error messages.
- Functions checking annotations related to memory properties are defined as empty functions. It's because Owi will always check the validity of memory read and write operations and does not need explicit function calling instructing it to do so.

3.3 Using Symbols to Generate Better Code

In order to check an annotation, E-ACSL sometimes generates conditional branches or introduces extra variables. Listing 3 shows a simplified example where E-ACSL uses a conditional branch and an extra variable `__gen_e_acsl_and` to check `y > x && z > y`.

While this branching approach works for concrete executions, it unnecessarily complicates symbolic execution. Indeed, 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 of the code generated for performing RAC. Listing 4 shows an optimized version suitable for symbolic execution. By reducing branching, we improve the performance of symbolic execution.

Some logical connectors available in E-ACSL, such as logical implication and logical equivalence, are not natively present in C. However, since the set of logical connectors in C is functionally complete, the general approach to optimize code checking any quantifier-free annotation is to transform it into a single C boolean expression.

3.4 Handling Unbounded Quantifiers Using Symbols

E-ACSL is constrained by its executable subset of ACSL, particularly when it comes to quantifiers, which must be bounded. This limitation arises because checking quantifications during concrete execution involves enumerating all possible values, a process that becomes impractical for unbounded quantifications.

For example, the annotation `//@ assert \forall int i; x != 2 * i;`, which asserts that the integer variable `x` is odd, cannot be expressed using E-ACSL. Symbolic execution, however, relaxes these constraints. Listing 5 illustrates how the above annotation can be

handled using symbolic variables. Owi introduces a symbol `__e_acsl_i` to represent the unbounded quantified variable `i`.

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

Listing 5. Code checking unbounded quantifications by using symbols.

4 Introducing Weasel for Symbolic Runtime Annotation Checking of Wasm Code

While Wasm is frequently used as a compilation target, it is also often written manually—for instance, to create runtime environments or replace assembly code segments. In these contexts, being able to annotate and verify Wasm code becomes useful. This section introduces a specification language designed for Wasm and details the implementation of a runtime annotation checker built on top of it.

4.1 The Custom Annotations Proposal

To allow writing annotations in Wasm's text format, we implemented the Wasm custom annotations proposal [par18] in Owi. This enables adding annotations in a way that adheres to Wasm's standards.

According to the specification, custom annotations can be placed anywhere spaces are allowed, and tools that do not recognize them will simply ignore them. These annotations resemble attributes in OCaml and follow the syntax `(@annotation-id ...)` in Wasm, where `annotation-id` should be a valid Wasm identifier.

4.2 Weasel: a WEbAssembly SpEcification Language

Weasel is a specification language for Wasm, inspired by ACSL. It is capable of expressing Wasm function contracts. Weasel function contracts are placed immediately before function definitions. A contract starts with `(@contract ind` and includes one or more clauses specifying preconditions (requires) and postconditions (ensures). The index of functions or variables can be either a zero-based number or an identifier prefixed with "$", following Wasm's text format syntax.

To align with Wasm's design, Weasel uses S-expression syntax. Below shows a simplified abstract syntax of Weasel:

⟨*contract*⟩ ::= `(@contract` *ind clause** `)`

⟨*ind*⟩ ::= `$` *id*
| *i32*

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

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

⟨*pprop*⟩ ::= *binpred term term*
| *binconnect prop prop*
| *unconnect prop*
| *quant quant_type prop*

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

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

⟨*unconnect*⟩ ::= `!`

⟨*quant*⟩ ::= `forall`
| `exists`

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

⟨*term*⟩ ::= `(` *pterm* `)`
| *ind*

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

⟨*binop*⟩ ::= `+`
| `-`
| `*`
| `/`

The nonterminals *prop* and *pprop* represent propositions, with *pprop* referring to parenthesized propositions. These propositions can be literals, predicates (*binpred*), logical connectors (*unconnect* and *binconnect*), or quantifications (*quant*). Quantifiers in Weasel follow the de Bruijn index system [de 72], though identifiers are supported as well.

The nonterminals *term* and *pterm* represent (parenthesized) terms, which include literals, parameters, global variables, quantifiers, operations, function return values, and memory accesses.

4.3 Generating (Symbolic) Runtime Annotations From Weasel

Weasel specifications can be translated into executable Wasm runtime annotations using Owi's code generator. This is implemented as the `owi instrument` subcommand of Owi. By default, it generates instrumented code that does not depend on symbolic execution, allowing RAC of Wasm code outside of Owi.

This subcommand also provides a `--symbolic` option, which generates code leveraging the aforementioned optimisations and importing Owi's `symbolic` module. In this case, annotations are checked during program execution in Owi's Wasm interpreter, with an SMT solver checking the satisfiability of assertions involving symbolic values at runtime.

Similar to the instrumentation process of E-ACSL, Owi generates a new Wasm file augmented with an additional function for each function contract, these functions call the original annotated function and verify the specified properties. Calls and exports of functions in the original module are replaced with the newly generated ones, implementing a module-level abstraction for RAC.

With the support of E-ACSL and Weasel, it is possible to generate our approach to cross-language programs, e.g., a C program that uses functions from a Wasm module. 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 of a cross-language program, either concretely or 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.

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 an 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 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 of detecting assertion violations. However, since fuzz testing is by nature incomplete, none of these works can provide the same guarantee on program properties as using symbolic execution.

WebAssembly Specification Languages. Currently, there are not many specification languages for Wasm. The work of Munuera Mazzaro [Mun23] proposes VerifiWasm. It verifies 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. First, their specifications are written in a separate file, while our approach embeds the specification in the Wasm module itself. The Wasm file carrying annotations is compact and compliant to Wasm's grammar specifications. In terms of annotation checking, they use a static symbolic execution engine, and we produce a Wasm module which can be run to check annotations of the original program, which is easier to implement and maintain. Their implementation is also restricted to part of the Wasm standard: integer operations are incomplete, floating-point numbers are not supported, and global variable and memory management are absent.

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.

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, using symbols increases the coverage of the execution path, hence strengthening the power of verifying program properties. We have also shown how to optimize code generated 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 integrated into the tool Owi. We demonstrate that specification languages make it possible to specify a 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 Wasm specification language. 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. 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.

[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.

[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.

[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.