Ortac: Runtime Assertion Checking for OCaml

Jean-Christophe Filliâtre, Clément Pascutto

Ortac: Runtime Assertion Checking for OCaml (tool paper)

Jean-Christophe Filliâtre[1] and Clément Pascutto[1,2] [0000−0002−5658−7731]

[1] Université Paris-Saclay, CNRS, ENS Paris-Saclay, Inria, Laboratoire Méthodes Formelles, 91190, Gif-sur-Yvette, France
jean-christophe.filliatre@lri.fr, clement.pascutto@lri.fr
[2] Tarides, 75005, Paris, France

Abstract. Runtime assertion checking (RAC) is a convenient set of techniques that lets developers abstract away the process of verifying the correctness of their programs by writing formal specifications and automating their verification at runtime.
In this work, we present `ortac`, a runtime assertion checking tool for OCaml libraries and programs. OCaml is a functional programming language in which idioms rely on an expressive type system, modules, and interface abstractions. `ortac` consumes interfaces annotated with type invariants and function contracts and produces code wrappers with the same signature that check these specifications at runtime. It provides a flexible framework for traditional assertion checking, monitoring misbehaviors without interruptions, and automated fuzz testing for OCaml programs.
This paper presents an overview of `ortac` features and highlights its main design choices.

Keywords: Runtime Assertion Checking· OCaml· Software Engineering

1 Introduction

OCaml is a general-purpose programming language featuring imperative, functional, and object-oriented paradigms. OCaml code is structured into modules. Each module comes with an interface, which exposes some of its contents (*e.g.* types, functions, or exceptions) and ensures a proper abstraction barrier. OCaml features an expressive type system that provides strong guarantees. An OCaml program will never try dereferencing a null pointer, for instance. Yet, it does not protect against programming errors such as accessing arrays out of their bounds, incorrectly implementing an algorithm, etc. Thus, good practices of software engineering apply to OCaml as well, including rigorous testing using technologies such as QuickCheck or fuzzing. We extend this toolset with `ortac`, a runtime assertion checking tool for OCaml. This tool is still under development. It is an open-source project available at https://github.com/ocaml-gospel/ortac.

We build upon Gospel, a behavioral specification language for OCaml [9]. Interfaces are annotated with formal specification, such as function contracts, type models and invariants, and logical definitions. Our tool consumes these interfaces and produces wrappers that check the Gospel function contracts and type invariants at runtime while maintaining the abstraction barrier. This paper presents an overview of `ortac` features and highlights its main design choices.

A critical feature of `ortac` is the identification of the executable subset of Gospel. Indeed, Gospel was not explicitly designed for runtime assertion checking—it is also used for deductive verification of OCaml code via the Why3 platform [13]—and annotations may contain unbounded quantifiers or uninterpreted logical symbols. In the process, `ortac` ignores anything that is not executable.

We address several features of OCaml that are challenging for runtime assertion checking. For instance, `ortac` suitably wraps OCaml functors (modules parameterized over modules) to ensure proper usage of their parameters. Interesting issues also arise in Gospel specification. For instance, contracts can express that a function is responsible for verifying a precondition, which coincides with a defensive programming idiom. In that case, the `ortac` wrapper checks that the function indeed performs this verification. Another convenient feature of Gospel is a structural, polymorphic equality. Consequently, `ortac` has to identify uses of this equality that can be implemented and checked at runtime.

We start with a motivating example (Sec. 2). Then we describe the tool architecture and discuss some technical aspects of `ortac` (Sec. 3). We conclude with related work (Sec. 4) and perspectives (Sec. 5).

2 Overview and Motivating Example

Let us illustrate the workflow of `ortac` on a simple OCaml module for modular arithmetic. Our interface is contained in a `modular.mli` file shown in Fig. 1. The corresponding implementation in `modular.ml` is omitted as its contents is not used by `ortac`. The Modular module is parameterized by a Modulus module, and exposes an abstract type t and three functions: a constructor of_int , an exponentiation power, and a division div.

Invoking the `ortac` executable on `modular.mli` produces a new module implementation in `modular_rtac.ml` containing a wrapper around the implementation, with the same interface as the unwrapped module. The generated code depends on the unwrapped module at runtime as it calls the original functions, but its implementation is not used nor altered during the generation. The generated code also depends on an external lightweight support library: `ortac-runtime`. We illustrate this procedure in Fig. 2. The client code can then freely call this wrapper, either in testing code or fuzzing code, or directly in production.

For instance `modular.mli` declares a function power that takes two arguments x and n. Its contract consists of two clauses. The checks clauses specifies a dynamic precondition n >= 0, meaning that the function must verify this property in its implementation and raise Invalid_argument if it is not met. The ensures

```
module type Modulus = sig
  val m : int
  (*@ ensures m > 0 *)
end

module Modular (M : Modulus) : sig
  type t = private { v : int }
  (*@ invariant 0 <= v < M.m *)

  val of_int : int -> t
  (*@ r = of_int x
      requires x >= 0
      ensures  r.v = x mod M.m *)

  val power : t -> int -> t
  (*@ r = power x n
      checks  n >= 0
      ensures r.v = (pow x.v n) mod M.m *)

  val div : t -> t -> t
  (*@ q = div x y
      ensures x.v = (y.v * q.v) mod M.m
      raises  Division_by_zero -> y.v = 0 *)
end
```

Fig. 1. Modular module interface (`modular.mli`).

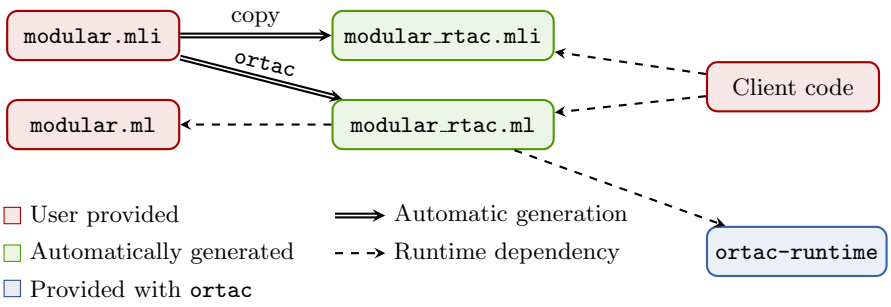

Fig. 2. Workflow using `ortac`.

clause is a postcondition, stating that r.v = (pow x.v n) mod M.m holds in the post-state, after calling the function. Therefore, `modular_rtac.ml` also contains a power function, with the same arguments, and verifies the contract with the following steps before returning the result:

– Call the original unwrapped power with arguments x and n, and keep the result in r.
– Check that if n is negative, the call raised the Invalid_argument exception, and that no exception was raised otherwise.
– Check that the postcondition r.v = (pow x.v n) mod M.m holds.

Note that Gospel's semantics is defined using mathematical integers. In the formula above, the functions pow and mod operate over arbitrary-precision integers and machine integers r.v, x.v, n, and M.m are implicitly promoted to arbitrary-precision integers by the Gospel type-ckecker.

If any of these verification steps fails during the execution of the client, the contract is violated, and the instrumented code reports the error precisely using the location of the contract and the clauses that failed:

```
$ ./client
File "modular.mli", lines 15-18, characters 2-43:
 Runtime error in function ‘power’:
 - the post-condition ‘r.v = (pow x.v n) mod M.m’ was violated.
```

3 Code Generation and Tool Architecture

In this section, we describe the translation of Gospel's formulas into Boolean OCaml expressions (Sec. 3.1). We then show how they are combined to generate function wrappers that check contracts and invariants (Sec. 3.2). Finally, we provide a few highlights on the modularity of `ortac` and how it helps developers customize its behavior (Sec. 3.3).

3.1 Translating formulas

Gospel is a tool-agnostic specification language: one can use it for both Deductive Verification (DV) and Runtime Verification (RV). Because of this, most formulas are not executable. For instance, they may contain unbounded quantifiers or logical symbols that are axiomatized but not implemented. The first step of the generation consists of identifying the executable subset of these specifications and compiling them into Boolean OCaml expressions whenever possible. In this process, `ortac` ignores formulas that do not translate and emits a warning to the user.

Mathematical integers. As mentioned in the previous section, integers in specifications use mathematical, arbitrary precision arithmetic. We use the `zarith` library to implement integers with these semantics with a marginal performance cost. It uses native integers whenever possible and `GMP` integers when needed. This implies that the instrumented code detects discrepancies between machine computations and their mathematical counterpart, and thus report overflows to the user.

Quantifiers. The use of exists and forall quantifiers in formulas generally makes them non-executable, because the quantifier domain may be unbounded. However, ortac identifies some bounded quantifiers patterns, such as `forall i. <term> <= i < <term> -> ...`, and properly translates them.

Equality. Gospel provides a logical equality that lets developers write easy to read formulas with the = operator. Although OCaml provides a polymorphic structural equality function, its use is limited. Its semantics may not coincide with the logical equality in the specification, and its execution may fail or diverge on functional or cyclic values. Instead, `ortac` leverages the type-checking information to generate safe monomorphic equality functions whenever the type permits.

Gospel standard library. Specifications can use functions from the Gospel standard library, an interface containing purely logical declarations for arithmetic operations and data containers such as arrays, bags, or maps. To execute them, `ortac-runtime` provides an implementation of this library using the most appropriate data structures. Some of these data structures require additional information to be implemented efficiently. For instance, an implementation of maps based on balanced trees requires a total order over the elements. In those cases, we use the same approach as for equality and derive them from the typing information when possible.

Undefineness and runtime exceptions. After `ortac` translates a formula into an OCaml expression, it is still possible that its execution raises an exception, *e.g.*, due to a division by zero or an array access out of bounds written in the specification. We catch any such failure in the wrapper and report it accordingly to the user.

3.2 Wrapping the Functions

After identifying and translating the executable subset of the specification formulas, `ortac` generates a wrapper around each function from the interface. Let us illustrate the translation schema on a generic example with two declarations, contained in a module Original. In the following, <X> denotes the translation of an executable formula X, and <failure> includes the code reporting the specification violations to the user.

```
1 type t = ...
2 (*@ invariant I *)
3
4 val f : t -> t
5 (*@ r = f x
6     checks   C
7     requires P
8     ensures  Q
9     raises   E -> R *)
```

The wrapper for f first verifies that the pre-state satisfies the preconditions and the invariants on input values. We also compute special checks preconditions at this point, but nothing is reported yet, since we expect the wrapped function itself to raise an exception in that case.

```
let f x =
  if not <I x> then <failure>;
  if not <P> then <failure>;
  let c = <C> in
  ...
```

The wrapper then calls the unwrapped function and examines any raised exception. In any exceptional function exit, the wrapper checks the invariants on the input values.

- If f raises an Invalid_argument exception, reporting the checks precondition C was not met, the wrapper indeed checks that property.
- If f raises an exception registered in a raises clause, the associated postcondition is checked along with checks clauses: in case of a violation, f should raise Invalid_argument instead.
- Finally, we consider any other exception a violation of the specification, since all possibly raised exceptions should appear in a raises clause.

```
  ...
  let r = try Original.f x with
    | Invalid_argument _ ->
      if c then <failure>;
      if not <I x> then <failure>
    | E ->
      if not c then <failure>;
      if not <R> then <failure>;
      if not <I x> then <failure>
    | e ->
      if not c then <failure>;
      <failure>
  in
  ...
```

If f exits normally, we check that c is verified—otherwise, f should have raised an exception— along with the postcondition Q, and invariants on both the inputs (when modified) and outputs. Finally, we return the value computed by f.

```
  ...
  if not c then <failure>;
  if not <Q> then <failure>;
  if not <I r> then <failure>;
  r
```

Note that our wrappers only test function calls that go through this interface, because only the interface contains specifications; the wrappers never check internal calls, including for recursive functions. For instance, if the function f calls a function g internally, then the wrapper for f still calls the original g, and not its wrapper. However, when `ortac` wraps a functor module F(A) = B, both the argument module A and the output module B are wrapped, and thus internal calls from B to A are checked.

3.3 A Modular Architecture

On top of the wrapper generator, `ortac` provides a framework for using the generated code in multiple settings. Its modular architecture lets contributors write custom *frontends* that control various aspects of the generated code.

Failure modes. By defining the exact contents of the <failure> code, frontends can set up various reporting strategies. For instance, effectively failing solely on preconditions enables `ortac` to only check the correct usage of the library by the client. One can also easily define a monitor-only mode, where errors are logged but do not interrupt the execution of the program.

Custom code. Arbitrary code can come along with the wrapper. For instance, this allows frontend developers to integrate it into a test framework or expose it with a different interface. Frontends can also redefine or extend the runtime library used by the instrumented code.

The `ortac` framework comes with a default frontend that immediately fails when a failure occurs to report it as soon as possible, as shown in Sec. 2. We also provide a fuzzing frontend, which pipes the instrumented code into an `american fuzzy lop` [1] powered executable that feeds the library with random inputs to find specification violations without the presence of user-written client code.

4 Related Work

Runtime assertion checking techniques have been implemented for many languages, with various design choices. Eiffel [17] is the first introduction of behavioral contract-based specifications in a programming language, together with runtime checking. JML [7, 10] is a behavioral specification language for Java, (mostly) executable by design and thus amenable to runtime assertion checking, *e.g.*, OpenJML [11]. SPEC# [3] extends the C# programming language with support for function contracts. AsmL [4, 5], then Code Contracts [2], implement similar yet less intrusive approaches for the .NET Framework. SPARK [16] also integrates program specifications into its host language, Ada. Frama-C [19] provides runtime assertion checking for C with its E-ACSL plugin [12, 20], which identifies and translates an executable subset of ACSL [6].

JML initially enforced machine arithmetic [10] before adding support for mathematical integers [8], and now uses the latter by default. SPARK also supports both modes but uses machine arithmetic by default. E-ACSL only supports mathematical integers, with significant optimisations [15] to limit the performance overhead. `ortac` supports mathematical integers by default, although it enables frontends to override this behavior to use machine arithmetic instead.

A distinctive feature of `ortac` is that is implements a rather large logical library, namely the Gospel standard library. Other runtime assertion checking tools either come with a very small logical library (*e.g.*, SPARK) or only provide an annotated subset of the programming standard library (*e.g.*, `libc` for E-ACSL or the Java standard library for OpenJML).

When it comes to undefineness—undefined or exceptional behaviors arising in formulas—`ortac` is similar to E-ACSL and SPARK, and raises exceptions when the evaluation of a formula fails. JML, on the other hand, substitutes undefined values with an arbitrary value of the correct type.

Note that while some tools provide powerful instrumentations to detect pointer unsafety, the type system of OCaml provides such guarantees natively in the language. In particular, `ortac` does not need to support a valid(p) construct—like E-ACSL does [14]—for user code nor Gospel formulas and avoids the subsequent overhead on the assertion checking generated code.

5 Conclusion and Perspectives

We presented `ortac`, a non-invasive runtime assertion checking tool for OCaml, based on formal specifications of module interfaces.

The project is under active development, and we are adding many features that extend the expressiveness of the tool. In particular, we are currently focused on supporting type models, which let the developers abstract their types with logical projections. We also plan to leverage Gospel's equivalent clauses to verify program equivalence and extend the support for user-provided functions such as equality and comparison.

We are already using `ortac` on industrial quality code developed at Tarides, namely in some components of Irmin, a distributed database built on the same principles as Git. First, it provides an additional, more formal documentation of some OCaml functions. Second, it alleviates the writing of tests, since the properties to be checked are written only once and are attached to the interface instead of the test cases. Last, `ortac` generates code that is integrated into a model-based fuzzing framework based on `afl-fuzz` [1] and Monolith [18], which further simplifies the testing process by automatically generating relevant test cases.

Currently, our use of `ortac` is limited to pre-deployment testing. We still need to measure the overhead of the runtime verification to determine if `ortac`-instrumented code can be deployed in contexts where performance is important.

References

1. `afl-fuzz` — `American fuzzy lop`, https://lcamtuf.coredump.cx/afl/
2. Barnett, M.: Code contracts for .net: Runtime verification and so much more. In: Barringer, H., Falcone, Y., Finkbeiner, B., Havelund, K., Lee, I., Pace, G., Roşu, G., Sokolsky, O., Tillmann, N. (eds.) Runtime Verification. pp. 16–17. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
3. Barnett, M., Leino, R., Schulte, W.: The spec# programming system: An overview. In: CASSIS 2004, Construction and Analysis of Safe, Secure and Interoperable Smart devices. Lecture Notes in Computer Science, vol. 3362, pp. 49–69. Springer (January 2005), https://www.microsoft.com/en-us/research/publication/the-spec-programming-system-an-overview/
4. Barnett, M., Schulte, W.: Contracts, components, and their runtime verification on the .net platform. Tech. Rep. MSR-TR-2002-38 (April 2002), https://www.microsoft.com/en-us/research/publication/contracts-components-and-their-runtime-verification-on-the-net-platform/
5. Barnett, M., Schulte, W.: Runtime verification of .net contracts. vol. 65, pp. 199–208. Elsevier (January 2003), https://www.microsoft.com/en-us/research/publication/runtime-verification-of-net-contracts/
6. Baudin, P., Cuoq, P., Filliâtre, J.C., Marché, C., Monate, B., Moy, Y., Prevosto, V.: Acsl: Ansi/iso c specification language.
7. Burdy, L., Cheon, Y., Cok, D., Ernst, M.D., Kiniry, J., Leavens, G.T., Rustan, K., Leino, M., Poll, E.: An overview of jml tools and applications1 1www.jmlspecs.org. Electronic Notes in Theoretical Computer Science **80**, 75–91 (2003). https://doi.org/https://doi.org/10.1016/S1571-0661(04)80810-7, https://www.sciencedirect.com/science/article/pii/S1571066104808107, eighth International Workshop on Formal Methods for Industrial Critical Systems (FMICS'03)
8. Chalin, P.: Jml support for primitive arbitrary precision numeric types: Definition and semantics. J. Object Technol. **3**, 57–79 (2004)
9. Charguéraud, A., Filliâtre, J.C., Lourenço, C., Pereira, M.: GOSPEL -Providing OCaml with a Formal Specification Language. In: FM 2019 - 23rd International Symposium on Formal Methods. Porto, Portugal (Oct 2019), https://hal.inria.fr/hal-02157484
10. Cheon, Y., Leavens, G.: A runtime assertion checker for the java modeling language (jml) (01 2002)
11. Cok, D.R.: Openjml: Jml for java 7 by extending openjdk. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NASA Formal Methods. pp. 472–479. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)
12. Delahaye, M., Kosmatov, N., Signoles, J.: Common Specification Language for Static and Dynamic Analysis of C Programs. In: Proceedings of the ACM Symposium on Applied Computing. pp. 1230–1235 (Mar 2013). https://doi.org/10.1145/2480362.2480593
13. Filliâtre, J.C., Paskevich, A.: Why3 — where programs meet provers. In: Felleisen, M., Gardner, P. (eds.) Proceedings of the 22nd European Symposium on Programming. Lecture Notes in Computer Science, vol. 7792, pp. 125–128. Springer (Mar 2013)
14. Kosmatov, N., Petiot, G., Signoles, J.: An optimized memory monitoring for runtime assertion checking of C programs. In: Runtime Verification. RV 2013. Lecture Notes in Computer Science. vol. 8174 LNCS, pp. 167–182. Rennes, France (2013). https://doi.org/10.1007/978-3-642-40787-1_10, https://hal-cea.archives-ouvertes.

fr/cea-01834990, conference of 4th International Conference on Runtime Verification, RV 2013 ; Conference Date: 24 September 2013 Through 27 September 2013; Conference Code:100802
15. Kosmatov, N., Maurica, F., Signoles, J.: Efficient runtime assertion checking for properties over mathematical numbers. In: Deshmukh, J., Ničković, D. (eds.) Runtime Verification. pp. 310–322. Springer International Publishing, Cham (2020)
16. McCormick, J.W., Chapin, P.C.: Building High Integrity Applications with SPARK. Cambridge University Press (2015)
17. Meyer, B.: Applying "Design by Contract". Computer **25**(10), 40–51 (Oct 1992). https://doi.org/10.1109/2.161279
18. Pottier, F.: Strong automated testing of OCaml libraries. In: Journées Francophones des Langages Applicatifs (JFLA) (Feb 2021), http://cambium.inria.fr/~fpottier/publis/pottier-monolith-2021.pdf
19. Signoles, J., Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Yakobowski, B.: Frama-c: a software analysis perspective. vol. 27 (10 2012). https://doi.org/10.1007/s00165-014-0326-7
20. Signoles, J., Kosmatov, N., Vorobyov, K.: E-ACSL, a Runtime Verification Tool for Safety and Security of C Programs (tool paper). In: RV-CuBES (2017). https://doi.org/10.29007/fpdh