Dynamic Verification of OCaml Software with Gospel and Ortac/QCheck-STM

Nikolaus Huber[1,2](✉), Naomi Spargo[1,3], Nicolas Osborne[1], Samuel Hym[1], and Jan Midtgaard[1]

[1] Tarides, France
https://tarides.com
{nicolas.osborne,samuel,jan}@tarides.com
[2] Uppsala University, Sweden
nikolaus.huber@it.uu.se
[3] Galois Inc, Arlington, VA, USA
nspargo@galois.com

Abstract This paper introduces the QCheck-STM plugin for Ortac, a framework for dynamic verification of OCaml code. Ortac/QCheck-STM consumes OCaml module signatures annotated with behavioural specification contracts expressed in the Gospel language, extracts a functional model of a mutable data structure from it, and generates code for automated runtime assertion checking. We report on the implementation of the tool, the structure of the generated code, and on errors found in established OCaml libraries.

Keywords: OCaml · verification · property-based testing · functional models · runtime assertion checking · random testing

1 Introduction

OCaml is an industrial strength, multi-paradigm programming language. While fundamentally functional at its core, OCaml includes many imperative features, such as references, mutable arrays, I/O, and exceptions. These pose unique challenges when trying to test and verify programs written in it.

While OCaml has been used as a platform for the implementation of various code analysis and verification tools, e.g., the interactive theorem prover Coq [44] and the C code analysis framework Frama-C [15], there is a lack of general purpose tools for the verification of OCaml programs themselves. In order to remedy this void, the Gospel project [9] equips OCaml with its own behavioural specification language.

The Gospel language is tool-agnostic, it only offers a way of expressing formal contracts, which can be leveraged by separate tools in order to perform analysis and verification tasks. Different such tools have been developed, including Cameleer [39], a deductive verification tool, and gospel2cfml[4], a translator of

[4] https://github.com/ocaml-gospel/gospel2cfml

annotated OCaml module signatures into separation logic terms embedded in Coq. In this paper, we focus on another tool consuming Gospel annotations called Ortac. Ortac provides a framework for automated runtime assertion checking, and is therefore a member of the family of dynamic verification tools. Ortac offers a modular architecture, where analysis and verification tasks are implemented as *plugins*. This paper highlights the QCheck-STM plugin, which focuses on black-box, model-based state-machine testing in the style of QuickCheck [3,26].

Given its multi-paradigm nature, OCaml is naturally suited to a number of different verification strategies. While it is possible to test purely functional code with Ortac/QCheck-STM, its strength lies in the verification of specifications relating to mutable data structures. Therefore, this paper will put emphasis on such programs. It is customary to refer to such a data structure as the *System Under Test* (SUT), and the functions provided to work with it as its *Application Programming Interface* (API).

This paper provides an overview of how Ortac/QCheck-STM is implemented, and how it may be used as a dynamic verification tool. To do so, we demonstrate the translation of specifications for a simple array library. The OCaml interface for the library is introduced in Section 2, the Gospel contracts for it in Section 3. After a short overview of Ortac and its plugin structure in Section 4 we showcase the generated code for the array example in Section 5. In Section 6 we evaluate the approach and share examples of bugs found in existing OCaml libraries. Finally, we discuss related work in Section 7, before we remark on future work and conclude in Section 8.

Ortac is an open-source project and its source code is available from the following URL:

`https://github.com/ocaml-gospel/ortac`

2 Running Example: Array

For illustration, we will test a library providing mutable arrays, which is an excerpt from OCaml's standard library `Array` module:

```
type 'a t

val make : int -> 'a -> 'a t
val length : 'a t -> int
val get : 'a t -> int -> 'a
val set : 'a t -> int -> 'a -> unit
```

For those unfamiliar with the syntax of OCaml, the code above defines a `type 'a t` representing arrays of a parametric type `'a`. In addition, this lists the type signatures of four OCaml functions. The first function `make` creates a fresh array from a given size and initialisation element. Function `length` accepts an array parameter and returns the size of it. Finally, `get` and `set` both take an array parameter and return and modify the element of an array at a given index, respectively.

3 Gospel by Example

In order to test the array library, the type and function signatures need to be annotated with Gospel specifications. To start, the type must be annotated with a model. In the tradition of other specification languages [6,29], annotations are added as special comments:

```
type 'a t
(* @ model size : integer
     mutable model contents : 'a sequence *)
```

An array can conceptually be thought of as a fixed capacity container. This logical model can be directly translated to Gospel by annotating `type 'a t` with two *model fields*, one for the immutable ***size*** and one for the mutable ***contents*** of the array. The types ***integer*** and ***'a sequence*** are part of the Gospel standard library and describe arbitrary precision integers and lists of values of type `'a`, respectively.

The `make` function creates new array instances given a size and an initial element:

```
val make : int -> 'a -> 'a t
(* @ t = make size a
     checks size >= 0
     ensures t.size = size
     ensures t.contents = Sequence.init size (fun j -> a) *)
```

The ***checks*** clause introduces a *pre-condition* that must hold at function entry. The two ***ensures*** clauses express that the resulting array has the expected size and that all entries are initialised to the given element ***a***. In addition to a ***checks*** clause, Gospel also offers a ***requires*** clause. Unlike a ***requires*** clause, with the above ***checks*** clause the behaviour of `make` is well-defined in case the pre-state does not meet the condition, as it means that the function raises an `Invalid_argument` exception in that case. The function ***Sequence.init*** is again part of the Gospel standard library.

The `set` function changes the value at a given position in the array:

```
val set : 'a t -> int -> 'a -> unit
(* @ set t i a
     checks 0 <= i < t.size
     modifies t.contents
     ensures t.contents = Sequence.set (old t.contents) i a *)
```

Again, the function ***Sequence.set*** is part of the Gospel standard library. The specification of `set` checks if the given index ***i*** is within the array bounds and

modifies the contents of the argument array, which is indicated by the ***modifies*** clause. For each model field marked as modified, the user needs to provide a corresponding ***ensures*** clause specifying how to construct the modified model. Note that it is implicitly assumed that in case the ***check*** fails, the argument SUT remains unchanged. It is possible to define custom exceptions and give equations for the model state after such an exception has been raised. For further information about other Gospel features, the interested reader is referred to the documentation[5].

For brevity, we will not explain all function contracts here, as both `length` and `get` follow analogously. The full specification of the example array library is shown below:

```
type 'a t
(*@ model size : integer
    mutable model contents : 'a sequence *)

val make : int -> 'a -> 'a t
(*@ t = make size a
    checks size >= 0
    ensures t.size = size
    ensures t.contents = Sequence.init size (fun j -> a) *)

val length : 'a t -> int
(*@ i = length t
    ensures i = t.size *)

val get : 'a t -> int -> 'a
(*@ a = get t i
    checks 0 <= i < t.size
    ensures a = t.contents[i] *)

val set : 'a t -> int -> 'a -> unit
(*@ set t i a
    checks 0 <= i < t.size
    modifies t.contents
    ensures t.contents = Sequence.set (old t.contents) i a *)
```

4 Ortac

Gospel itself does not perform any kind of verification. It is the job of other tools to take the provided specifications and perform further analysis.

The Ortac [18] tool provides functions for converting the given annotations into OCaml code. Ortac is extensible through *plugins*, which can make use of

[5] `https://ocaml-gospel.github.io/gospel/`

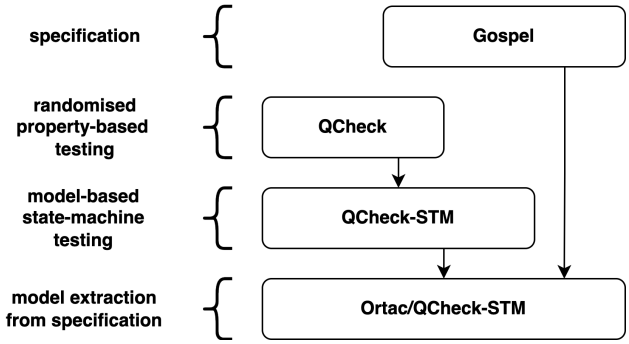

Figure 1. Inner architecture of Ortac/QCheck-STM.

these functions to check specifications. Currently, three different plugins are implemented: Wrapper, Monolith, and QCheck-STM. The original Ortac prototype was developed as part of Clément Pascutto's PhD work [38].

Ortac/Wrapper generates a wrapper module from an annotated module signature, which instruments each function with assertions on the argument and result values according to the given specifications. Ortac/Monolith [35] generates code to interface with Monolith [40], a fuzzing tool for OCaml. However, both the Wrapper and Monolith plugins currently do not support the definition of models. Therefore, they are of limited use when testing mutable data structures. The new QCheck-STM plugin lifts this limitation, by translating Gospel specifications into tests using the QCheck-STM framework. The basic idea behind Ortac/QCheck-STM is to extract a purely functional model of the SUT from the provided Gospel specifications and compare the behaviour of both while running random call sequences of the associated API.

5 Implementation

In this section, we describe the overall architecture of Ortac/QCheck-STM. We then describe how to generate code for testing a single SUT, and finally, how to generalise this approach to support multiple SUTs.

5.1 Ortac Underneath the Hood

In order to understand the working of Ortac/QCheck-STM, it is insightful to look at its constituent parts, as illustrated in Figure 1. At its core, it uses the QCheck library[6], which offers utilities for randomised property-based testing. If a user provides a random generator for a type `'a` and a property as a function `'a -> bool`, QCheck can then run a sequence of tests by randomly generating test inputs of the given type and checking that the property holds for each of

[6] `https://github.com/c-cube/qcheck`

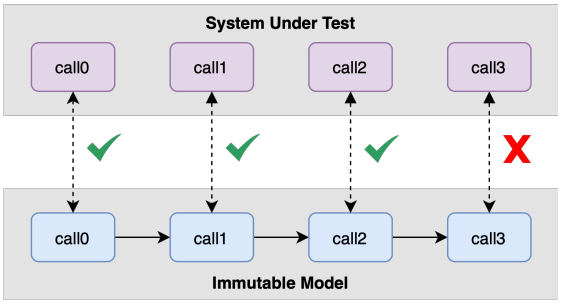

Figure 2. Comparing the behaviour of the SUT and model on randomly generated API call sequences.

them. In case of a violation, QCheck automatically shrinks the given test input to show a minimised counterexample to the user.

QCheck-STM [32] builds upon this functionality by providing a framework for testing a mutable data structure (the SUT) against an immutable reference model, as illustrated in Figure 2. It generates random API call sequences, and then checks the property that the behaviour of the SUT and its functional model coincide for each such test input. In case of a violation of that property, QCheck-STM also reports minimised call sequence traces.

To test a particular data structure and its API with QCheck-STM, the user has to write the model manually. Ortac/QCheck-STM offers the ability to generate the functional model automatically from the Gospel specification. It uses the ***model*** annotations in order to generate the functional model, and the ***modifies*** and ***ensures*** clauses to update the model on each function call.

5.2 Generated code

To illustrate the working of Ortac/QCheck-STM, we will show parts of the generated code from the array specifications introduced in Section 2. First, however, Ortac/QCheck-STM needs another input besides the annotated interface file, which is the configuration of the SUT:

```
type sut = char Array.t
let init_sut = Array.make 16 'a'
```

A minimal configuration needs to provide the `type sut` and a value of that type named `init_sut`. As the name suggests, `type sut` defines the particular type we would like to test, which needs to be fully instantiated. E.g., here we instantiate the polymorphic array type to `char`. The `init_sut` value specifies the initial SUT value from which to start each test. If the Gospel annotated interface is in a file *array.mli* and the configuration in *config.ml*, Ortac/QCheck-STM can be invoked as follows:

```
ortac qcheck-stm array.mli config.ml
```

The generated code consists of multiple functor specifications and error reporting information. We will simplify the shown code examples to aid conciseness. As a starting point, the SUT and model are defined:

Generated

```
type sut = char Array.t
let init_sut () = Array.make 16 'a'
type state = {
    size : integer;
    contents : char sequence;
}
let init_state = {
    size = integer_of_int 16;
    contents = Sequence.init (integer_of_int 16) (fun _ -> 'a');
}
```

The astute reader will have realised that the definitions of `type sut` and `init_sut` are taken from the provided configuration module (`init_sut` is turned into a function here, as the generated test executable will run multiple instances of random API sequences, for which fresh initial SUT values need to be created). The `type state` defines the two model fields from the specification of `type 'a t` shown in Section 2, where the type variable `'a` has been instantiated to `char` according to the configuration. The initial state value has been synthesised from the given specification of the `make` function by comparing its signature to the `init_sut` function from the configuration module. It does not need to be turned into a function, since it is immutable. The functions `integer_of_int` and `Sequence.init` are provided by the Gospel standard library.

Next, the type of available function calls (i.e., commands) is defined:

Generated

```
type cmd =
  | Length
  | Get of int
  | Set of int * char
```

Each constructor carries argument values according to the formal arguments of the respective signature. Notice, that the SUT argument is missing, as it is not randomly generated, but rather kept and updated by the testing runtime. Furthermore, the `make` function is not present, as it returns a new SUT. This will be amended in Section 5.3.

In order to perform randomised property-based testing with QCheck-STM, a QCheck generator for the `cmd` type is defined:

Generated

```
let arb_cmd state = QCheck.make show_cmd Gen.(oneof [
  pure Length;
  pure (fun i -> Get i) <*> int;
  pure (fun i a -> Set (i, a)) <*> int <*> char;
])
```

This definition needs some explanation. The function `arb_cmd` takes as the only input the current state (i.e., the functional model). This is currently unused, but might be used in the future to define smarter random command generators (see Section 8). `QCheck.make` creates new instances of random generators for a given type by taking both a function that can print values (here defined by `show_cmd` which is left out for brevity) and a generator which can create random values of that type. The `Gen` module provides basic generators and combinators to define new ones. `Gen.oneof` randomly selects from a list of generators. `Gen.pure` always returns its argument value. For simple cases like `Length`, this is enough, however, some command constructors carry fields for their argument values, which need to be provided by random generators as well. For example, the `Get` constructor needs an integer for the index it shall fetch from the array. The infix operator `val ( <*> ) : ('a -> 'b) Gen.t -> 'a Gen.t -> 'b Gen.t` can be used to turn a function generator into a generator of its return type by providing a generator of its argument type. This is done by using the provided generators of base types such as `int` and `char`.

Next, we generate a function that can run a command on a given SUT:

Generated

```
let run cmd sut = match cmd with
  | Length -> Res (int, length sut)
  | Get i -> Res ((result char exn), protect (get sut i))
  | Set (i, a) -> Res ((result unit exn), protect (set sut i a))
```

The function `run` matches on the current command and calls the respective array function. The result constructor `Res` is provided by QCheck-STM and carries as fields the returned value and a pretty-printer for its respective type (e.g., `int` and `result char exn`). The function `protect` turns functions raising exceptions into functions returning values of type `result` (in OCaml all exceptions are part of the extensible variant type `exn`).

As the SUT value is mutable, its internal state will change in-place during the execution of `run`. The functional model of the SUT has to be updated separately. Therefore, a function `next_state` is defined:

Generated

```
let next_state cmd state = match cmd with
  | Length -> state
  | Get _ -> state
  | Set (i, a) ->
```

```
      if (0 <= i) && (i < state.size) then
        {
          size = state.size;
          contents = Sequence.set state.contents i a;
        }
      else state
```

Functions `length` and `get` do not mutate the array, and therefore they return the argument state unchanged. When setting a value at a particular index, the next state depends on if the index is within the array bounds. If so, the new state has the same size as the old one, and the contents are the same besides at the given index (`Sequence.set` is again part of the Gospel standard library). If the check fails, the underlying array stays unchanged. The individual cases within `next_state` are extracted from the *ensures* clauses of the respective function contract in Section 3. This is why each field that is marked as modified needs to provide a corresponding post-condition describing the model of the post-state.

Finally, a post-condition function is generated:

Generated

```
let ortac_postcond cmd state res =
  let new_state = next_state cmd state in
  match (cmd, res) with
    | Length, Res (_, i) ->
          if i = new_state.size then None
          else (* error report *)
    | Get i, Res (_, a) ->
        if (0 <= i) && (i < new_state.size) then
          (match a with
            | Ok a -> if a = Sequence.get new_state i then None
                else (* error report *)
            | _ -> (* error report *))
        else
          (match a with
            | Error (Invalid_argument _) -> None
            | _ -> (* error report *))
    (* further cases *)
```

The function `ortac_postcond` takes the current command, the current state, and the result after calling `run` as input, and returns an optional error report. As all post-conditions refer to the state after executing a given command, it first defines the new state by calling `next_state` (Gospel offers the *old* operator to refer to the pre-state, which we utilise in the specification of the `set` function). We only show the post-conditions for the `length` and `get` functions, the others follow analogously.

In the case of `length` we expect the returned integer to coincide with the `size` field of `new_state`. For `get` the returned value depends on if the given index was

within bounds. If it was, the returned character should be the same as the one taken from the functional model, and otherwise we expect an `Invalid_argument` exception.

We have left out the actual error reporting mechanism for brevity. The careful reader may have realised that there are in fact two different categories of post-conditions. In the case of `make` and `set`, the *ensures* clauses define the model after executing the respective function, which is used in `next_state`. For `length` and `get` they state a property of the return value, which can be checked in `ortac_postcond`.

5.3 Functions returning and consuming multiple SUTs

Thus far, all testable functions only took one SUT argument as input, and did not return a SUT value as an output (recall that the `make` function was temporarily omitted). Let us extend the available array API with another function `append` from the OCaml standard library's `Array` module. The function `append` takes two arrays, and returns a fresh array with the contents of both arguments appended. The specification is straightforward, when utilising the sequence concatenation operator `( ++ )` from the Gospel standard library:

```
val append : 'a t -> 'a t -> 'a t
(* @ t = append a b
    ensures t.size = a.size + b.size
    ensures t.contents = a.contents ++ b.contents *)
```

In order to allow testing functions that take multiple arguments of the SUT type, or likewise return a value of that type, the generated code is adapted:

Generated

```
type sut = char Array.t Stack.t
let init_sut () =
    let s = Stack.create () in
    for _ = 0 to max_sut - 1 do
        Stack.push (Array.make 16 'a') s
    done;
    s
type element = {
    size : integer;
    contents : char sequence;
}
type state = element list
let init_state = List.init max_sut (fun _ -> {
    size = integer_of_int 16;
    contents = Sequence.init (integer_of_int 16) (fun _ -> 'a');
})
```

The `type sut` is not a single SUT any more, it represents a (mutable) stack of SUT values (`Stack` is part of the OCaml standard library). Correspondingly, `type state` must describe a functional model of the SUT stack, as is done here through a list of model elements.

The variable `max_sut` is defined as the maximum number of SUT arguments ever needed by any single function of the API under test. For our running array example, `max_sut` is 2, as `append` expects two arguments of type `t`. This number can easily be determined during code generation. By starting with a SUT stack already filled with the maximum number of initial elements ever needed by any API call, each available function can immediately be used.

The generator for arbitrary commands now includes the full API:

Generated

```
let arb_cmd state = QCheck.make show_cmd Gen.(oneof [
  pure (fun s a -> Make (s, a)) <*> small_signed_int <*> char;
  pure Length;
  pure (fun i -> Get i) <*> int;
  pure (fun i a -> Set (i, a)) <*> int <*> char;
  pure Append
])
```

Notice, that the size argument `s` to `Make` is not provided by the `int` generator. Ortac/QCheck-STM uses a simple heuristic of classifying functions that produce a SUT but take no SUT argument as *initialisation functions*, i.e., functions that create new SUT instances. For these functions, any `int` generator is automatically changed to `small_signed_int` in order to keep the runtime of the generated test executable low.

For brevity, we will not show all the adapted code examples from the previous section again, but only describe the overall behaviour. When a function requires multiple SUT arguments, the required amount of SUTs is popped from the stack, the function is run, and the SUTs pushed back onto the stack in reverse order (this allows to capture changes to the argument SUTs in the post-condition). If a function returns a SUT, this value is pushed on the stack as well (so that the post-condition has access to it). This is illustrated in Figure 3 for the `append` function.

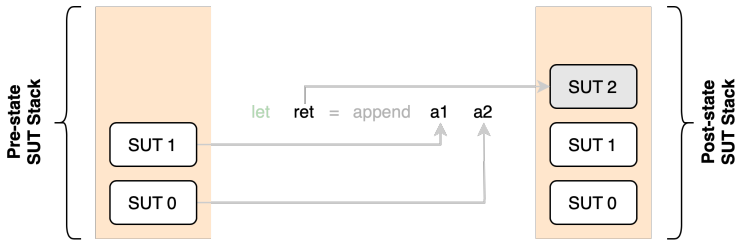

Figure 3. Function call with arguments and return value on the SUT stack.

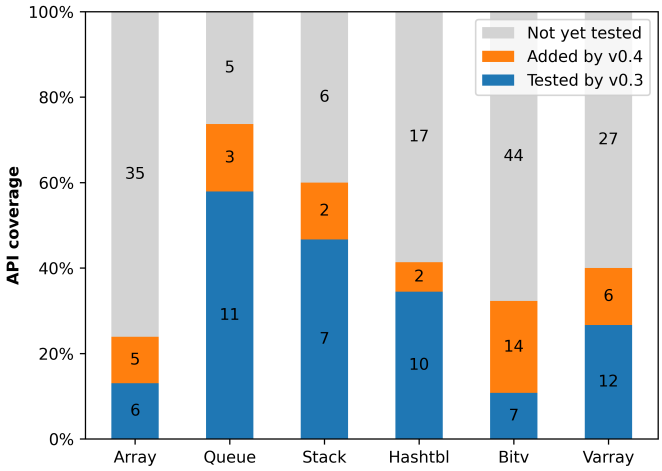

Figure 4. Number of API functions covered by the generated code.

6 Evaluation

We have used Ortac/QCheck-STM to test 6 OCaml modules as summarised in Figure 4, including the `Array`, `Stack`, `Queue`, and `Hashtbl` modules from the OCaml standard library. We have found 5 crashing bugs, and 1 function from the standard library needing a documentation fix (the source code for these tests, including the respective Gospel contracts, is available online [25]). Resolving the reported issues later revealed 2 additional unexpected exceptions in one of the tested modules. We will elaborate further on these findings later in this section.

Figure 4 displays the number of API functions covered by the generated testing code. With Ortac/QCheck-STM version 0.3 without the extension described in Section 5.3, this covers 11% (7/65) to 58% (11/19) of the module APIs. Ortac/QCheck-STM version 0.4 adds support for testing functions that take multiple SUT arguments and return new SUT values, as described in Section 5.3. Doing so increases the module API coverage further to 24% (11/46) to 74% (14/19). Interestingly, 7 out of 8 of the reported errors in this paper are all due to functions that were not testable with version 0.3. Ortac automatically skips a function for which no suitable Gospel annotation is provided. The biggest remaining contributor to untested functions is higher-order functions, such as `map` and `iter`. We expand on lifting this restriction in Section 8.

While no bugs were found within the standard library modules, one curiosity was discovered: The function `Hashtbl.create` is documented in the following way[7]:

[7] `https://ocaml.org/manual/5.2/api/Hashtbl.html`

```
val create : ?random:bool -> int -> ('a, 'b) t
(** [Hashtbl.create n] creates a new, empty hash table, with
   initial size [n].  For best results, [n] should be on the
   order of the expected number of elements that will be in
   the table.  The table grows as needed, so [n] is just an
   initial guess. ... *)
```

A natural Gospel specification would therefore be to model the hash table type as an association list (similar to dictionaries in other languages). The specification of the `create` function could then look similar to the following:

```
type ('a, 'b) t
(*@ mutable model contents : ('a * 'b) list *)

val create : ?random:bool -> int -> ('a, 'b) t
(*@ h = create ?random size
    checks size >= 0
    ensures h.contents = [] *)
```

Running the test generated by Ortac/QCheck-STM reveals the following:

```
Gospel specification violation in function create

  File "hashtbl.mli", line 7, characters 11-20:
    size >= 0

when executing the following sequence of operations:

  [@@@ocaml.warning "-8"]
  open Hashtbl
  let protect f = try Ok (f ()) with e -> Error e
  let sut0 = create ~random:false 16
  let r = protect (fun () -> create true (-8))
  assert (match r with
            | Error (Invalid_argument _) -> true
            | _ -> false)
  (* returned Ok (<sut>) *)
```

It turns out, that the initial guess can be negative, in which case it has the same effect as providing zero. After reporting this, the documentation for the `create` function has been revised in OCaml 5.3[8].

[8] `https://github.com/ocaml/ocaml/pull/13535`

Outside of the standard library, 2 libraries from the official OCaml package repository have been tested as well, which revealed errors in both of them:

The `Bitv` library[9] is a mature, 25-year-old OCaml library implementing mutable bit-vectors of arbitrary but fixed length, with an API very similar to arrays. In three of its functions (`fill`, `sub`, and `blit`) Ortac/QCheck-STM tests discovered that their index-bound checking could lead to an integer overflow, resulting in a segmentation fault on at least one tested platform (the usage of unsafe language features or external code can lead to diverging behaviour on different operating systems or processor architectures). The issue has been reported and fixed[10] consequently. Furthermore, Ortac/QCheck-STM found two cases of unexpected exceptions being raised when trying to rotate a zero-length vector. The issue has been reported[11] and fixed[12] as well.

The `Varray` library[13] implements extensible arrays. It uses an intricate data structure [22] along with some unsafe OCaml tricks in order to obtain good amortised performance, of which many can lead to crashing programs if used incorrectly. Such a crashing scenario was found when starting from an empty array and then adding and removing an element from different ends of the array[14]. Anecdotally, tests of the `Varray` library have been part of the Ortac code-base almost from the initial release for internal testing. These discovered an initial bug early on[15]. The latest error, however, was only recently discovered when Ortac/QCheck-STM was extended to cover functions returning SUT values as described in Section 5.3.

Given the automated nature of the code generated by Ortac/QCheck-STM, it is particularly suited to be included in a *Continuous-Integration* (CI) pipeline, as is demonstrated by the CI used in Ortac's GitHub repository[16]. So far, all observed test runs have shown runtimes in the range of hundreds of milliseconds, even with the extension described in Section 5.3. Despite these tests still not reaching full API coverage, we believe this highlights the tool's usability in CI.

7 Related Work

Fundamental ideas within modern verification can be traced back to Hoare. This is the case for invariants and pre- and post-conditions as found in Hoare logic triples [23] as well as proving an implementation correct with respect to a model [24]. Meyer later put the concepts into use in the *design-by-contract* methodology of the Eiffel programming language [31]. The `require(s)` and `ensure(s)` keywords of modern specification languages such as Gospel thus have roots in Eiffel.

[9] `https://github.com/backtracking/bitv`
[10] `https://github.com/backtracking/bitv/pull/32`
[11] `https://github.com/backtracking/bitv/issues/33`
[12] `https://github.com/backtracking/bitv/commit/f30e7a8`
[13] `https://github.com/art-w/varray`
[14] `https://github.com/art-w/varray/issues/2`
[15] `https://discuss.ocaml.org/t/ann-varray-0-2/13492`
[16] `https://github.com/ocaml-gospel/ortac`

The Gospel specification language for OCaml follows a line of specification languages such as the Java Modeling Language (JML) for Java programs [29], the ANSI/ISO C Specification Language (ACSL) for C programs [6], and Spec# for C# programs [5].

Software engineering tools to validate program specifications can roughly be divided into two categories: One group of tools works by dynamic *runtime assertion checking*, whereas another group of tools performs static verification. The JML-consuming ESC/Java tool [21] targets both of these categories. The ACSL-consuming Frama-C tool [15] targets the latter. The Gospel-consuming Ortac tool targets the former, but other Gospel tools (under development) [39] target the latter.

Whereas the above specification languages and verification tools have been developed for existing programming languages a posteriori, a newer class of programming languages have verification fundamentally built in from the beginning. This is the case for Lean [33], Why3 [19], F* [43], and Dafny [30], among others.

Another approach to formally verified software development is *correct-by-construction* techniques. Filliâtre et al. [17] describe a development process that builds upon both Gospel and Why3 as part of the VOCAL [10] project. The user provides an OCaml module signature with Gospel annotations, which is translated to a WhyML specification. The module is then implemented in WhyML, proven correct with respect to the translated specification by Why3, and automatically translated back to OCaml code.

The term *property-based testing* was introduced by Fink and Bishop [20] originally. It became popular with QuickCheck, an embedded domain-specific language for the functional programming language Haskell [12], and has since been ported to numerous other languages, including OCaml [46]. QuickCheck introduced modular combinators for building up generators of complex test inputs, how each input is tested on properties in the form of Boolean-valued functions, and test input shrinking when finding a counterexample. Whereas the original QuickCheck formulation targeted purely functional code, in follow-up work Claessen and Hughes presented extensions to target monadic, effectful Haskell code, including the idea of model-based testing [13]. While QuickCheck was primarily conceived as a testing tool, the method has since been introduced in various interactive theorem provers in order to quickly provide counterexamples during the proof development process. Examples of this can be found in Isabelle [7,8], Coq [37], and Agda [16].

With roots in *model-based testing* from outside the functional programming language community [45], the Gast framework for the Clean programming language offered *state machines* to specify the intended behaviour of stateful reactive systems [27]. The design of the state-machine framework for the commercial Erlang QuickCheck [3,26] has since influenced framework ports for other languages, e.g., for Scala [34] and QCheck-STM for OCaml as used in this work [32]. The state-machine approach furthermore extends to testing stateful code for race conditions under concurrent usage [14].

In an impressive feat, Arts et al. [4] have developed state-machine models of the AUTOSAR specification to test automotive software. The range of defects found in doing so underlines the usefulness of the approach. Other successful QuickCheck applications include testing of telecommunication software [3], data structures [2], election software [28], computational geometry algorithms [41], compilers [36], and run-time systems [32].

8 Conclusion and Future Work

In this paper, we have presented Ortac/QCheck-STM, a tool that consumes behavioural contracts expressed in the Gospel specification language, and generates code to automatically test a given OCaml module against a functional reference model derived from these contracts. Despite being a relatively young tool with the first version released in October 2023, Ortac/QCheck-STM has already proven useful in finding bugs in established OCaml libraries, as well as pointing out inconsistencies in documentation. We expect to find more errors with it as we continue annotating more libraries with Gospel contracts.

While this paper focuses on Ortac/QCheck-STM, previous work [42] has investigated verification of OCaml code by leveraging both static and dynamic verification tools for Gospel, including Ortac. In that work, the authors remark on various limitations of Ortac, of which many have been lifted (including, for example, the verification of functions taking multiple SUT arguments, or returning SUT values). However, Ortac/QCheck-STM still has various limitations, which we would like to lift in future work:

Given its nature as a dynamic verification tool, Ortac is inherently restricted to the executable fragment of the Gospel specification language. At times, this results in contracts that are not as natural as their purely logical counterpart. By extending the accepted syntactic forms, contracts could be written in ways that would make them more amenable to other forms of verification (and their respective tools) as well.

By design, the Gospel specification language implicitly requires that mutable arguments do not alias, i.e., that they occupy separate memory locations in the OCaml heap [9]. Therefore, Ortac/QCheck-STM does not attempt to generate calls with aliased SUTs. Once Gospel is enhanced to express aliasing properties, we would like to extend Ortac/QCheck-STM accordingly to exercise such specifications.

The majority of functions in Figure 4 currently not covered by the generated code comprise idiomatic higher-order functions such as `map`, `fold`, and `iter`. Gospel currently does not have a way of specifying effectful function arguments, whereas it is possible to specify the behaviour of functions accepting pure function arguments [9]. As a first step, Ortac should be able to lift a restriction for the latter, e.g., using Claessen's approach to function generation [11]. Secondly, once Gospel has set on a way for specifying effectful function arguments, we hope to extend Ortac to cover such API calls as well.

Currently, preconditions introduced by *requires* clauses are not considered during the generation of arbitrary command lists on which to test the SUT and the model. During the execution of each test case, if a given pre-state does not fulfil the stated pre-condition for a particular command, it is simply skipped. The random generator could be extended to take *requires* clauses into account while generating random sequences of commands, which would increase the efficiency of the tool.

QCheck-STM was originally developed to test the new multicore runtime arriving with OCaml 5 [32]. It can therefore also produce parallel sequences of random API calls, and test if the observed behaviour is sequentially consistent by reconciling each run with a sequential execution of a given model. By extending Ortac to use the parallel test generator, it would be possible to also test concurrent data-structures (as for example done by Artho et al. [1]).

Acknowledgments. This work was funded by ANR grant ANR-22-CE48-0013 and Tarides.

References

1. Artho, C., Gros, Q., Rousset, G., Banzai, K., Ma, L., Kitamura, T., Hagiya, M., Tanabe, Y., Yamamoto, M.: Model-Based API Testing of Apache ZooKeeper. In: 2017 IEEE International Conference on Software Testing, Verification and Validation (ICST). pp. 288–298 (2017). https://doi.org/10.1109/ICST.2017.33
2. Arts, T., Castro, L.M., Hughes, J.: Testing Erlang data types with Quviq QuickCheck. In: Proceedings of the 7th ACM SIGPLAN Workshop on Erlang, ERLANG'08. pp. 1–8 (2008). https://doi.org/10.1145/1411273.1411275
3. Arts, T., Hughes, J., Johansson, J., Wiger, U.T.: Testing telecoms software with Quviq QuickCheck. In: Proceedings of the 2006 ACM SIGPLAN Workshop on Erlang. pp. 2–10 (2006). https://doi.org/10.1145/1159789.1159792
4. Arts, T., Hughes, J., Norell, U., Svensson, H.: Testing AUTOSAR software with QuickCheck. In: Eighth IEEE International Conference on Software Testing, Verification and Validation, ICST 2015 Workshops. pp. 1–4 (2015). https://doi.org/10.1109/ICSTW.2015.7107466
5. Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# programming system: An overview. In: Construction and Analysis of Safe, Secure, and Interoperable Smart Devices. pp. 49–69 (2005). https://doi.org/10.1007/978-3-540-30569-9_3
6. Baudin, P., Filliâtre, J.C., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C Specification Language, http://frama-c.com/download/acsl.pdf
7. Berghofer, S., Nipkow, T.: Random Testing in Isabelle/HOL. In: SEFM. vol. 4, pp. 230–239 (2004). https://doi.org/10.1109/SEFM.2004.10049
8. Bulwahn, L.: The New Quickcheck for Isabelle. In: Certified Programs and Proofs. pp. 92–108 (2012). https://doi.org/10.1007/978-3-642-35308-6_10
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 (2019). https://doi.org/10.1007/978-3-030-30942-8_29
10. Charguéraud, A., Filliâtre, J.C., Pereira, M., Pottier, F.: VOCAL – A Verified OCAml Library (Sep 2017), https://inria.hal.science/hal-01561094, ML Family Workshop 2017

11. Claessen, K.: Shrinking and showing functions: *(functional pearl)*. In: Proceedings of the 5th ACM SIGPLAN Symposium on Haskell, Haskell 2012. pp. 73–80 (2012). https://doi.org/10.1145/2364506.2364516
12. Claessen, K., Hughes, J.: QuickCheck: a lightweight tool for random testing of Haskell programs. In: Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming, ICFP'00. pp. 268–279 (2000). https://doi.org/10.1145/351240.351266
13. Claessen, K., Hughes, J.: Testing monadic code with QuickCheck. In: Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell, Haskell 2002. pp. 65–77 (2002). https://doi.org/10.1145/581690.581696
14. Claessen, K., Palka, M.H., Smallbone, N., Hughes, J., Svensson, H., Arts, T., Wiger, U.T.: Finding race conditions in Erlang with QuickCheck and PULSE. In: Proceedings of the 14th ACM SIGPLAN international conference on Functional programming, ICFP 2009. pp. 149–160 (2009). https://doi.org/10.1145/1596550.1596574
15. Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C. In: Software Engineering and Formal Methods. pp. 233–247 (2012). https://doi.org/10.1007/978-3-642-33826-7_16
16. Dybjer, P., Haiyan, Q., Takeyama, M.: Combining Testing and Proving in Dependent Type Theory. In: Basin, D., Wolff, B. (eds.) Theorem Proving in Higher Order Logics. pp. 188–203 (2003). https://doi.org/10.1007/10930755_12
17. Filliâtre, J.C., Gondelman, L., Lourenço, C., Paskevich, A., Pereira, M., Melo de Sousa, S., Walch, A.: A Toolchain to Produce Verified OCaml Libraries (Jan 2020), https://hal.science/hal-01783851
18. Filliâtre, J.C., Pascutto, C.: Ortac: Runtime Assertion Checking for OCaml (Tool Paper). In: Runtime Verification: 21st International Conference, RV 2021, Proceedings. pp. 244–253 (2021). https://doi.org/10.1007/978-3-030-88494-9_13
19. Filliâtre, J.C., Paskevich, A.: Why3 — Where Programs Meet Provers. In: Proceedings of the 22nd European Symposium on Programming. Lecture Notes in Computer Science, vol. 7792, pp. 125–128 (2013). https://doi.org/10.1007/978-3-642-37036-6_8
20. Fink, G., Bishop, M.: Property-based testing: a new approach to testing for assurance. SIGSOFT Softw. Eng. Notes **22**(4), 74–80 (1997). https://doi.org/10.1145/263244.263267
21. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for Java. In: Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation. pp. 234–245. PLDI'02 (2002). https://doi.org/10.1145/512529.512558
22. Goodrich, M.T., Kloss, J.G.: Tiered Vectors: Efficient Dynamic Arrays for Rank-Based Sequences. In: Algorithms and Data Structures. pp. 205–216 (1999). https://doi.org/10.1007/3-540-48447-7_21
23. Hoare, C.A.R.: An Axiomatic Basis for Computer Programming. Commun. ACM **12**(10), 576–580 (1969). https://doi.org/10.1145/363235.363259
24. Hoare, C.A.R.: Proof of correctness of data representations. Acta Informatica **1**, 271–281 (1972). https://doi.org/10.1007/BF00289507
25. Huber, N., Osborne, N., Hym, S., Midtgaard, J.: Dynamic Verification of OCaml Software with Gospel and Ortac/QCheck-STM - Software Artifact (Oct 2024). https://doi.org/10.5281/zenodo.13988146
26. Hughes, J.: Software testing with QuickCheck. In: Central European Functional Programming School - Third Summer School, CEFP 2009, Revised Selected Lec-

tures. Lecture Notes in Computer Science, vol. 6299, pp. 183–223 (2009). https://doi.org/10.1007/978-3-642-17685-2_6
27. Koopman, P.W.M., Plasmeijer, R.: Testing reactive systems with GAST. In: Revised Selected Papers from the Fourth Symposium on Trends in Functional Programming, TFP 2003. vol. 4, pp. 111–129 (2003)
28. Koopman, P.W.M., Plasmeijer, R.: Testing with functional reference implementations. In: Trends in Functional Programming - 11th International Symposium, TFP 2010. Revised Selected Papers. Lecture Notes in Computer Science, vol. 6546, pp. 134–149 (2010). https://doi.org/10.1007/978-3-642-22941-1_9
29. Leavens, G.T., Baker, A.L., Ruby, C.: Preliminary design of JML: a behavioral interface specification language for Java. SIGSOFT Softw. Eng. Notes **31**(3), 1–38 (May 2006). https://doi.org/10.1145/1127878.1127884
30. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: Logic for Programming, Artificial Intelligence, and Reasoning. pp. 348–370 (2010). https://doi.org/10.1007/978-3-642-17511-4_20
31. Mandrioli, D., Meyer, B., et al.: Advances in object oriented software engineering. Prentice Hall (1992)
32. Midtgaard, J., Nicole, O., Osborne, N.: Multicoretests — Parallel testing libraries for OCaml 5.0. In: OCaml Users and Developers Workshop (2022), https://github.com/ocaml-multicore/multicoretests
33. de Moura, L., Kong, S., Avigad, J., van Doorn, F., von Raumer, J.: The Lean theorem prover (system description). In: Automated Deduction - CADE-25. pp. 378–388 (2015). https://doi.org/10.1007/978-3-319-21401-6_26
34. Nilsson, R.: ScalaCheck: The Definitive Guide. Artima (2014)
35. Osborne, N., Pascutto, C.: Leveraging Formal Specifications to Generate Fuzzing Suites. In: OCaml Users and Developers Workshop (2021), https://inria.hal.science/hal-03328646
36. Palka, M.H., Claessen, K., Russo, A., Hughes, J.: Testing an optimising compiler by generating random lambda terms. In: Proceedings of the 6th International Workshop on Automation of Software Test. pp. 91–97. AST'11 (2011). https://doi.org/10.1145/1982595.1982615
37. Paraskevopoulou, Z., Hriţcu, C., Dénès, M., Lampropoulos, L., Pierce, B.C.: Foundational Property-Based Testing. In: ITP 2015 - 6th conference on Interactive Theorem Proving. Lecture Notes in Computer Science, vol. 9236 (2015). https://doi.org/10.1007/978-3-319-22102-1_22
38. Pascutto, C.: Runtime verification of OCaml programs. Theses, Université Paris-Saclay (Oct 2023), https://theses.hal.science/tel-04696708
39. Pereira, M., Ravara, A.: Cameleer: A deductive verification tool for OCaml. In: Computer Aided Verification. pp. 677–689 (2021). https://doi.org/10.1007/978-3-030-81688-9_31
40. Pottier, F.: Strong Automated Testing of OCaml Libraries. In: JFLA 2021 - 32es Journées Francophones des Langages Applicatifs (Feb 2021), https://inria.hal.science/hal-03049511
41. Sergey, I.: Experience report: growing and shrinking polygons for random testing of computational geometry algorithms. In: Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016. pp. 193–199 (2016). https://doi.org/10.1145/2951913.2951927
42. Soares, T.L., Chirica, I., Pereira, M.: Static and dynamic verification of OCaml programs: The Gospel ecosystem (extended version) (2024), https://arxiv.org/abs/2407.17289

43. Swamy, N., Hriţcu, C., Keller, C., Rastogi, A., Delignat-Lavaud, A., Forest, S., Bhargavan, K., Fournet, C., Strub, P.Y., Kohlweiss, M., Zinzindohoue, J.K., Zanella-Béguelin, S.: Dependent types and multi-monadic effects in F*. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016. pp. 256–270 (2016). https://doi.org/10.1145/2837614.2837655
44. The Coq development team: The Coq proof assistant (Jun 2024). https://doi.org/10.5281/zenodo.11551307
45. Tretmans, J.: Testing concurrent systems: A formal approach. In: CONCUR '99: Concurrency Theory, 10th International Conference, Proceedings. Lecture Notes in Computer Science, vol. 1664, pp. 46–65 (1999). https://doi.org/10.1007/3-540-48320-9_6
46. Wikipedia: QuickCheck Wikipedia page (Accessed: 2024-10-06), https://en.wikipedia.org/wiki/QuickCheck