Property-Based Testing of OCaml 5's Runtime System

Fun and Segfaults with Interpreters and State Transition Functions

Jan Midtgaard*
Independent
Aarhus, Denmark

Abstract

We describe our effort on using property-based testing to test the OCaml 5 multicore runtime system. In particular, we cover three case studies of increasing complexity that utilize a model-based state machine framework: (a) Testing the Array module, (b) testing weak hash sets, and (c) testing the garbage collector, with the latter two behaving non-deterministically from the point of view of the black-box testing process. We evaluate the approach empirically by analyzing the bugs found, and discuss both limitations and challenges we have met underway.

CCS Concepts: • **Software and its engineering** → **Software testing and debugging**; **Correctness**; **Runtime environments**.

Keywords: property-based testing, model-based testing, runtime system, garbage collection

ACM Reference Format:
Jan Midtgaard. 2025. Property-Based Testing of OCaml 5's Runtime System: Fun and Segfaults with Interpreters and State Transition Functions. In *Proceedings of the Workshop Dedicated to Olivier Danvy on the Occasion of His 64th Birthday (OLIVIERFEST '25), October 12–18, 2025, Singapore, Singapore*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3759427.3760378

1 Introduction

Since the introduction of QuickCheck [6], property-based testing (PBT) has seen a range of success stories applying the approach to locate software errors in everything from telecommunication software [3] and computational geometry algorithms [35] to election software [23] and AUTOSAR software in Volvo cars [19].

Some effort from the programming language community has been dedicated to using PBT to test language processors in the form of compilers [26, 29] and static analyzers [17, 27], both of which are complex and hard to assure the quality of. Most compilers, however, operate by emitting code that interact with a runtime system, e.g., including a garbage collector. OCaml 5's transition from a single-core system to support multicore execution required a major rewrite of the garbage collector and runtime system [36]. As these parts are mostly written in C, notorious for its *undefined behaviour* and memory unsafety, this rewrite came with the risk of introducing errors and setting back OCaml's stability. In this paper, we report on our effort to utilize PBT to test OCaml 5's runtime system. In particular, we consider three representative case studies to test OCaml's `Array` module, its weak hash sets, and its garbage collector, by utilizing a model-based state-machine framework [3, 18]. This approach has the advantage of being able to test random combinations of calls and furthermore extends to parallel tests for race conditions [8, 20]. Finally, the approach rests on interpreters and state transition functions for expressing semantics, both of which have been central topics throughout Olivier Danvy's research career [1, 2, 10–14].

1.1 A QuickCheck Reminder

At its core, PBT is built on two concepts: a *generator* to deliver random test cases, and a *property* which is believed to hold for all such test cases. For example, we can test that `@`, the inline `List` concatenation operator from OCaml's standard library, is associative: $\forall xs, ys, zs.\ xs\,@\,(yz\,@\,zs) = (xs\,@\,yz)\,@\,zs$. To do so, we need a generator of (three) lists of some element type, e.g., integers. Using OCaml's QCheck library [5] we can express the PBT by utilizing the `triple`, `list`, and `int` combinators to generate random triples of integer lists, and by replacing the universal quantifier with an anonymous function accepting each generated test case:

```
open QCheck
let concat_assoc =
  Test.make ~name:"Concat␣assoc"
    (triple (list int) (list int) (list int))
    (fun (xs,ys,zs) ->
      (xs @ ys) @ zs = xs @ (ys @ zs))
```

*Part of this work was done while at Tarides.

OLIVIERFEST '25, Singapore, Singapore

ACM ISBN 979-8-4007-2150-2/25/10
https://doi.org/10.1145/3759427.3760378

We can then run the PBT as follows:

```
let _ =
  QCheck_base_runner.run_tests_main [concat_assoc]
```

By default, this will generate and check the property on 100 random test cases, which passes as expected:

```
generated error fail pass / total  time  test name
[✓]  100    0    0  100 / 100    0.0s Concat assoc
==================================================
success (ran 1 tests)
```

Now suppose we have written a false property, e.g., mistakingly swapping `yz` and `zs` in the right-hand-side:

```
let concat_assoc' =
  Test.make ~name:"Concat␣assoc'"
    (triple (list int) (list int) (list int))
    (fun (xs,ys,zs) ->
      (xs @ ys) @ zs = xs @ (zs @ ys))
```

In this case QCheck will find and automatically *shrink* the test case into a minimal counterexample. Below the first generated test case did not satisfy the buggy property, and QCheck then spent 134 shrinking steps reducing it to a minimal counterexample:

```
generated error fail pass / total  time  test name
[✗]    1    0    1    0 /  100   0.0s Concat assoc'

--- Failure --------------------------------------

Test Concat assoc' failed (134 shrink steps):

([], [0], [-1])
```

1.2 Paper Outline

The rest of this paper is structured as follows: In the following sections we cover three case studies of increasing complexity. In Sec. 2 as a warm-up, we describe PBTs of OCaml's `Array` module. As a second case study, in Sec. 3 we describe PBTs of non-determistic weak hash tables. As our last and biggest case study, in Sec. 4 we describe PBTs of OCaml 5's garbage collector. Sec. 5 evaluates and discusses challenges and limitations of the approach. Finally, Sec. 6 and Sec. 7 discusses related work and concludes, respectively.

2 Warm Up: Specifying and Testing Arrays

To get familiar with model-based state machine tests and the `QCheck-STM` framework [28], as a warm-up we will first consider how one can test OCaml's built-in arrays. For presentational purposes we will limit ourselves to only a small selection of the `Array` module's signature. We first declare a `cmd` type that describes the `Array` definitions we want to target, along with a to-string function for easy printing. For example, since `Array.set` expects an integer index and a replacement element as arguments (in addition to the targeted array), we let the `Set` constructor take these two as arguments. We also declare a separate type of array elements, `elem`, along with a `show` function for them:

```
type elem = int
type cmd =
  | Get of int
  | Set of int * elem
  | To_list
  | Sort

let show_elem e = Printf.sprintf "0x%X" e
let show_cmd cmd = match cmd with
  | Get i      -> Printf.sprintf "Get␣%i" i
  | Set (i, e) ->
    Printf.sprintf "Set␣(%i,␣%s)" i (show_elem e)
  | To_list    -> "To_list"
  | Sort       -> "Sort"
```

To specify the intended behaviour of the `Array` functions we provide a model. In this case, we model the array contents with a simple list of elements. We initialize the model's elements (somewhat arbitrarily) to be `1`. Finally, we write a state transition function `next_state`, describing the state changes that each `cmd` may cause. Both `Get` and `To_list` leave the state untouched and are hence just the identity. We specify the behaviour of `Array.sort` by appeal to (and relying on the correctness of) `List.sort`. Finally, using `List.mapi` we express the effect of `Set` by returning a new state that is the same as `s`, except for the updated entry:

```
type state = elem list

let array_size = 10
let init_state  = List.init array_size (fun _ -> 1)
let next_state c s = match c with
  | Get _     -> s
  | Set (i,e) ->
    List.mapi (fun j e' -> if i=j then e else e') s
  | To_list   -> s
  | Sort      -> List.sort Int.compare s
```

Now we can write a `cmd` generator. Using QCheck's `Gen.oneof` combinator this will choose fairly between generating one of the four different `cmd` types. Our generator of array indices uses `arb_cmd`'s model argument of type `state` to skew the generation to primarily generate indices within bounds, with $\frac{1}{6}$ chance of generating something bigger. For array elements, in `elem_gen` we generate `int`s with a single 1-bit fairly among OCaml's 63-bit integers (on a 64-bit platform):

```
let power_of_2s =
  Array.init 63 (fun i -> 1 lsl i)

let arb_cmd s =
  let int_gen =
    Gen.(frequency
           [1,small_nat;
            5,int_bound (List.length s - 1)]) in
  let elem_gen = Gen.oneofa power_of_2s in
```

```
QCheck.make ~print:show_cmd
  Gen.(oneof
    [ map (fun i -> Get i) int_gen;
      map2
        (fun i e -> Set (i,e)) int_gen elem_gen;
      return To_list;
      return Sort;
    ])
```

Next we specify how to actually interpret a `cmd` over our *system under test* (SUT). We first specify the type of the SUT and two hooks `init_sut` and `cleanup` to create and tear it down, respectively. As unused arrays will be garbage collected, we leave the `cleanup` function empty. In the `run` interpreter, we annotate each result with a combinator denoting its result type. For example, as `sort` reorders the array elements in place as a side effect, it is annotated with the `unit` combinator. In this example, we furthermore make use of an extensible result type `ty`, to extend it with an `Elem` constructor for our array elements, and define a combinator `elem` pairing it up with our element show function. We can now utilize this extension, to declare that `To_list` returns a list of elements, and have any counterexample printed using our dedicated `show_elem` function. Finally, as `Get` and `Set` may raise an exception when indexing an array out of bounds, we wrap their calls in `protect` to convert the result to a `result` type. For example, as `Array.get` returns the element at the given index, we annotate the result with `result elem exn` to indicate either the successful or exceptional behavior:

```
type sut = elem Array.t
let init_sut () = Array.make array_size 1
let cleanup _   = ()

type _ ty += Elem : elem ty
let elem = (Elem, show_elem)

let run c a = match c with
  | Get i ->
    Res (result elem exn,
         protect (fun () -> Array.get a i) ())
  | Set (i,c) ->
    Res (result unit exn,
         protect (fun () -> Array.set a i c) ())
  | To_list -> Res (list elem, Array.to_list a)
  | Sort    -> Res (unit, Array.sort Int.compare a)
```

As a last piece of the puzzle we specify acceptable behaviour as pre- and post-conditions. If we only wanted to test positively, e.g., arrays within bounds, we could put a bounds check as a pre-condition. Instead we want to test both the successful and the error cases and therefore provide a constant `true` function as our pre-condition. The `postcond` function is more interesting. As arguments it accepts the `cmd`, the model, and the combinator-annotated result from the `run` function above. For example, for `Sort` returning a unit there is not much to check about the return value. For `To_list` we check that the returned result agrees with the model. Both `Get` and `Set` check that the result is an exception when indexing out of bounds. If the index `i` is within bounds, `Get` further checks that the model's `i`-th entry agrees with the observed result:

```
let precond _c _s = true
let postcond c (s:state) res = match c, res with
  | Get i, Res ((Result (Elem,Exn),_), r) ->
    if i < 0 || i >= List.length s
    then r = Error (Invalid_argument
                      "index␣out␣of␣bounds")
    else r = Ok (List.nth s i)
  | Set (i,_), Res ((Result (Unit,Exn),_), r) ->
    if i < 0 || i >= List.length s
    then r = Error (Invalid_argument
                      "index␣out␣of␣bounds")
    else r = Ok ()
  | To_list, Res ((List Elem,_),cs) -> cs = s
  | Sort, Res ((Unit,_), ()) -> true
  | _, _ -> false
```

The above constitutes a model-based state-machine specification for `QCheck-STM`. Assuming it is placed in a module `AConf`, we can now use it to drive different kinds of PBT runs.

2.1 Sequential Tests

In the simplest mode, we can use the specification to drive sequential PBTs, using the `STM_sequential.Make` *functor*, i.e., a module-to-module function:

```
module ArraySTM_seq = STM_sequential.Make(AConf)
let _ =
  QCheck_base_runner.run_tests_main
    [ArraySTM_seq.agree_test
                    ~count:1000 ~name:"Array␣seq"]
```

This will generate 1000 random `cmd` lists, interpret each list over the model and the SUT, and check agreement between the two. Running it passes as expected on OCaml 5.3.0:

```
random seed: 406349609
generated error fail pass / total   time test name
[✓] 1000    0    0 1000 / 1000     0.0s Array seq
================================================
success (ran 1 tests)
```

2.2 Parallel Tests and Negative Tests

Following Quviq QuickCheck [8, 20] we can reuse the same model to run parallel tests. To do so, we first generate and then interpret two random `cmd` sequences each in their own, freshly-spawned parallel *domain* (OCaml's terminology for a parallel operating system thread). Second we perform an interleaving search for a sequential model interpretation that is consistent with the observed parallel result. The parallel model-based testing property succeeds if the search is successful.

Race conditions are non-deterministic by nature. To make matters worse, they may be timing sensitive and trigger rarely. By default we therefore check each test case 25 times

```
random seed: 288227847
generated error fail pass / total     time test name
[✓]     1    0    1    0 / 1000     1.2s Array par

--- Info ----------------------------------------------------------------------

Negative test Array par failed as expected (29 shrink steps):

[...]

  Results incompatible with linearized model

                                    |
                                    |
                  .-----------------------------------.
                  |                                   |
     Set (2, 0x10) : Ok (())                      Sort : ()
                                                Get 7 : Ok (0x10)

==============================================================================
success (ran 1 tests)
```

Figure 1. The output of QCheck-STM running a parallel Array test

and fail if just one of the repetitions fail. This has the effect of increasing the error rate of such rare failures. To prevent always starting the parallel execution off in the same initial SUT state, we prepend a sequential prefix of random `cmd`s leading to some arbitrary SUT state.

The `Array` module from OCaml's standard library is an archetypical example of a module from ~30 years of single-core OCaml, which is unsafe for parallel usage without further coordination between the involved domains, e.g., using a mutex. We can confirm this suspicion by running a parallel PBT. To do so, we reuse our specification module `AConf` and pass it to the `STM_domain.Make` functor. As we expect to find a counterexample, we phrase this parallel test as a negative test with `neg_agree_test_par`., i.e., a PBT which will *pass if it finds a counterexample* and fail otherwise:[1]

```
module ArraySTM_dom = STM_domain.Make(AConf)
let _ =
  QCheck_base_runner.run_tests_main tests
    [ArraySTM_dom.neg_agree_test_par
                  ~count:1000 ~name:"Array␣par"]
```

Although the counterexamples differ between runs, due to both non-determinism and different randomness seeding, QCheck and QCheck-STM may find a counterexample already in the first test case that it checks as shown in Fig. 1. After 29 shrink steps, the counterexample has been reduced to just 3 parallel `cmd`s, with an empty sequential prefix. The counterexample is annotated with the result observed from each of the 3 `cmd`s. It shows that when executing `Array.set 2 0x10` in parallel with `Array.sort` on a 10-element integer array initialized with ones, after completing `Array.sort` the `0x10` entry may be observed at index 7. Clearly, no interleaving of the `Set` command before, in the middle of, or after `Sort` and `Get` can reconcile this observed behaviour with a sequential execution.

2.3 Stress Tests

Stopping after finding a counterexample in the first test case leaves little parallel testing of OCaml 5's runtime system. To supplement the counterexample finding we therefore run random *stress tests* with a property that exercises parallel usage, but skips the interleaving search and thus only fails on an unexpected exception or a crash:

```
let _ =
  QCheck_base_runner.run_tests_main tests
    [ArraySTM_dom.stress_test_par
                  ~count:1000 ~name:"Array␣stress"]
```

This test succeeds as expected, but takes longer to complete:

```
random seed: 330577663
generated error fail pass / total   time test name
[✓] 1000    0    0 1000 / 1000 104.7s Array stress
==================================================
success (ran 1 tests)
```

Having seen how model-based tests in QCheck-STM can be used to test the `Array` module both sequentially, in parallel, and for stress tests, we proceed with our next case study.

[1] Note that this swapping of the PBT success criteria thereby tests an existentially quantified property, mirroring the negation of the more common PBT usage of testing a universally quantified property.

3 Testing Weak Hash Sets

Through the `Weak.Make` functor, OCaml offers weak hash sets. This data structure is based on weak pointers, and hence elements may be removed from it underway. This means that the specification for the module is non-deterministic: Elements may be present or not, and both behaviours are acceptable.

To keep the presentation manageable, we again limit it to a selection of the functor's bindings. As for arrays, we write a `cmd` data type representing the functions that we will target. As weak hash sets operate on garbage collected data, we use OCaml's heap allocated strings as our element type. For brevity we omit the `show` functions for the two types:

```
type data = string
type cmd =
  | Add of data
  | Remove of data
  | Find_opt of data
  | Count
```

To model weak hash sets we use a simple list of data elements. As elements can be removed from the weak hash set underneath the hood, we model the "best case behaviour" in which this does not happen and instead handle the non-determinism in the post-condition. Our initial model will be an empty list. The state transition function to interpret `cmd`s over the model, conses new elements at the front of the list for `Add`. A `Remove` command will remove the first occurrence of the given element using a helper function, whereas `Find_opt` and `Count` will not alter the state of the model:

```
type state = data list
let init_state = []

let rec remove_first d s = match s with
  | [] -> s
  | d'::s' ->
    if d=d' then s' else d'::(remove_first d s')

let next_state c s = match c with
  | Add d    -> d::s
  | Remove d -> remove_first d s
  | Find_opt _
  | Count -> s
```

With our model in place, next we write a generator of commands. Again this just chooses uniformly between the four constructors. More interestingly, we again utilize that the generator accepts a state argument to generate duplicates: If there are elements present we choose to generate one of them with a 50% chance. Otherwise we just generate a small string of printable characters:

```
let arb_cmd s =
  let string_gen = Gen.(string_small_of printable) in
  let data_gen = match s with
    | [] -> string_gen
    | _::_ -> Gen.(oneof [oneofl s; string_gen]) in
  QCheck.make ~print:show_cmd
    Gen.(oneof
           [ map (fun d -> Add d) data_gen;
             map (fun d -> Remove d) data_gen;
             map (fun d -> Find_opt d) data_gen;
             return Count;
           ])
```

We now define our SUT as weak hash sets with strings. To set tests off in a relatively clean state, we invoke a minor garbage collection after each test case run. Finally `run` calls the appropriate operation accordingly. As parallel usage of `Add`, `Remove`, and `Find_opt` may yield an `Invalid_argument "index␣out␣of␣bounds"` exception under parallel usage, we wrap them in a suitable `protect` call to turn either of these outcomes into a `result` type:

```
module WHS = Weak.Make(String)
type sut = WHS.t
let weak_size = 10

let init_sut () = WHS.create weak_size
let cleanup _ = Gc.minor ()

let run c hs = match c with
  | Add d ->
    Res (result unit exn,
         protect (fun () -> WHS.add hs d) ())
  | Remove d ->
    Res (result unit exn,
         protect (fun () -> WHS.remove hs d) ())
  | Find_opt d ->
    Res (result (option string) exn,
         protect (fun () -> WHS.find_opt hs d) ())
  | Count ->
    Res (int, WHS.count hs)
```

Finally we are in position to state the pre- and post-condition. We leave the pre-condition constantly true. As `Add` and `Remove` return a unit under normal usage, our post-condition expects them to do so. The case for `Count` has to take into account that entries can have been removed, and is hence relaxed to ensure that there are at most as many entries as in our best case model. Finally for `Find_opt` to reflect the non-determinism, we specify that "no entry" as well as a model-agreeing entry are both acceptable behaviours:

```
let precond _c _s = true
let postcond c (s:state) res = match c, res with
  | Add _, Res ((Result (Unit,Exn),_),r) ->
    r = Ok ()
  | Remove _, Res ((Result (Unit,Exn),_),r) ->
    r = Ok ()
  | Find_opt d,
      Res ((Result (Option String,Exn),_),r) ->
    r = Ok None
    || (List.mem d s && r = Ok (Some d))
  | Count, Res ((Int,_),r) ->
    r <= List.length s
  | _,_ -> false
```

```
random seed: 362920295
generated error fail pass / total     time test name
[✓]   83    0    1   82 / 5000     7.0s Weak HashSet parallel

--- Info ----------------------------------------------------------------------

Negative test STM Weak HashSet test parallel failed as expected (17 shrink steps):

[...]

  Results incompatible with linearized model

                                    |
                          Add ";K[oJ(y" : Ok ())
                          Add ";K[oJ(y" : Ok ())
                                    |
                  .-----------------------------------.
                  |                                   |
     Remove ";K[oJ(y" : Ok ())           Remove ";K[oJ(y" : Ok ())
      Add ";K[oJ(y" : Ok ())                     Count : 2

==============================================================================
success (ran 1 tests)
```

Figure 2. The output of QCheck-STM running a parallel weak hash set test

3.1 Reconsidering Duplicates

One might wonder whether unconditionally adding a duplicate element to a weak hash set is the correct behaviour. For example, the following modified `next_state` model of `Add` may seem plausible:

```
let next_state c s = match c with
  | Add d    -> if List.mem d s then s else d::s
  | (* ... *)
```

Running a sequential PBT with this change however quickly finds a minimal counterexample illustrating a mismatch between the model and the SUT's element count:

```
Test Weak HashSet seq failed (7 shrink steps):

[...]

  Results incompatible with model

   Add ";1" : Ok ())
   Add ";1" : Ok ())
   Count : 2
```

A closer inspection of the `Weak.Make` specification also reveals that the weak hash set indeed allows multiple instances of the same `data` element.

3.2 Sequential, Parallel, and Stress Testing

Again we utilize the same state machine specification to drive both sequential, parallel, and stress tests of weak hash sets. Of these, the parallel test is the most interesting to study the output of, as the other two positive tests succeed as expected on OCaml 5.3.0. Fig. 2 contains a counterexample output of a parallel weak hash set run. The module is also unsafe to use in parallel, and we therefore also consider this test negative, i.e., succeeding when finding a counterexample. In the counterexample, the sequential prefix first adds two elements (both the string `";K[oJ(y"`) to the weak hash set. It then runs two `Remove ";K[oJ(y"` commands in parallel, with one followed by an `Add` and the other followed by a `Count`. With two `";K[oJ(y"` strings in the weak hash set initially, after removing one occurrence, there is no way to end up with a count of 2, no matter how the remaining `Remove ";K[oJ(y"` and `Add` commands are interleaved.

4 Testing the Garbage Collector

As a third and final example, we discuss how one can test OCaml 5's garbage collector (GC). Let us first recall the design of the OCaml 5 multicore GC [36].

4.1 A Brief Reminder of the OCaml 5 GC

As a garbage collected language, OCaml will automatically reclaim any previously allocated memory it deems no longer reachable and hence unused. Like the OCaml 4 GC (and many others), the OCaml 5 GC is generational, split into a minor heap and a major heap, with memory records promoted from the former to the latter. In the multicore setup, each domain has its own minor heap, whereas the major heap is shared across all domains. The major heap is itself split into a number of pools, and each pool is assigned to a domain.

Major heap compaction was restored with OCaml 5.2.0, thus motivating the need for more GC testing.

To prevent that one domain (or thread) blocked in a call hinders overall progress, e.g., preventing other domains (or threads) from garbage collecting, each domain has a *backup thread*. When a domain thus enters a blocking call, the backup thread therefore takes over to react to, e.g., requests for GC. When the blocking call completes, the domain resumes this duty while the backup thread steps back and awaits again.

4.2 A PBT for the OCaml 5 GC

To test the OCaml 5 GC, for one we want to create a mixture of both reachable and unreachable memory records through dynamic allocations. The OCaml standard library exposes a `Gc` module, through which a programmer can explicitly invoke both minor and major collections, as well as change and query GC parameters dynamically. We expect all of these to act according to their specification, and furthermore that the runtime system does not crash, hang, or fail any assertions. With that in mind, we can write a `cmd` type covering both bindings in the `Gc` module, as well as commands to allocate different kinds of memory records (we abbreviate `Bigarray` as `BA`):

```
type cmd =
  (* cmds targeting the Gc module *)
  | Stat
  | Quick_stat
  | Counters
  | Minor_words
  | Get
  | Set of setcmd
  | Minor
  | Major_slice of int
  | Major
  | Full_major
  | Compact
  | Allocated_bytes
  | Get_minor_free
  (* cmds to allocate memory *)
  | Cons64 of int
  | PreAllocStr of int * string
  | AllocStr of int * int
  | CatStr of int * int * int
  | PreAllocList of int * char list
  | AllocList of int * int
  | RevList of int
  | PreAllocBigarray of
    int * (int, BA.int_elt, BA.c_layout) BA.Array1.t
  | AllocBigarray of int * int
```

Starting last, the second half contains different commands to allocate memory of various forms: 64-bit integers, strings, lists, and bigarrays, as the former is a small heap-allocated record while the latter is a special case in OCaml's GC. We phrase our allocating commands in two variants: One form, e.g. `PreAllocStr`, pre-allocates at test generation time (thus in a parent domain's heap), and another form, e.g. `AllocStr`, allocates at test execution time (thus in the executing domain's heap). Thereby we obtain a mix of heap records allocated in different domain heaps. We furthermore throw in `CatStr` and `RevList` commands for concatenating strings and reversing lists, respectively, as example operations that perform some computation in addition to allocation. The constructors carry integer indices to denote source and target of read and written heap-allocated records. For example, `CatStr` expects 3 integers denoting two string indices being read, and a third string index denoting where to store the resulting concatenated string.

In the first half of the above `cmd` definition, each constructor corresponds to a binding in OCaml's `Gc` module. For example, `Gc.stat` returns a `stat` record with various GC statistics (e.g., minor, major, and live words), and `Gc.compact` triggers a compaction of the major heap, incl. a full major GC. For brevity we again omit `show` functions. The `Set` constructor models `Gc.set : Gc.control -> unit`. In our test setup it expects a `setcmd` that expresses the ability to change one of OCaml's several GC parameters dynamically:

```
type setcmd =
  | Minor_heap_size of int
  | Space_overhead of int
  | Stack_limit of int
  | Custom_major_ratio of int
  | Custom_minor_ratio of int
  | Custom_minor_max_size of int
```

These six constructors each correspond to a field in the standard library's `Gc.control` type, except that we omit four fields which are unavailable in OCaml 5 and hence always 0. We also omit the `verbose` field, as changing it will trigger more output to log files and thus add noise:

```
type control = {
  minor_heap_size : int;
  major_heap_increment : int;    (* omitted *)
  space_overhead : int;
  verbose : int;                 (* omitted *)
  max_overhead : int;            (* omitted *)
  stack_limit : int;
  allocation_policy : int;       (* omitted *)
  window_size : int;             (* omitted *)
  custom_major_ratio : int;
  custom_minor_ratio : int;
  custom_minor_max_size : int;
}
```

Contrary to the `Array` and weak hash set example, in the `Gc` test the model and the SUT will not be strictly aligned, illustrating the flexibility of a model-based state machine framework. We will use our model to express the internal state of the `Gc.control` parameters. Since only `Set` may change this state, this is reflected in our model's state transition function (we omit the definition for `round_heap_size`):

```
type state = Gc.control

let next_state n s = match n with
  | Set subcmd -> (match subcmd with
    | Minor_heap_size m         ->
      { s with Gc.minor_heap_size
                 = round_heap_size m }
    | Space_overhead so         ->
      { s with Gc.space_overhead = so }
    | Stack_limit sl            ->
      { s with Gc.stack_limit = sl }
    | Custom_major_ratio cmr    ->
      { s with Gc.custom_major_ratio = cmr }
    | Custom_minor_ratio cmr    ->
      { s with Gc.custom_minor_ratio = cmr }
    | Custom_minor_max_size ms  ->
      { s with Gc.custom_minor_max_size = ms })
  | _ -> s
```

Initializing our model is slightly more thorny. For one, the state of `Gc.control` changes the `verbose` field when run under the debug runtime—a variant of the OCaml runtime system with additional assertions and debug checks enabled. Second, `Gc.control` can be changed via the `OCAMLRUNPARAM` (and historically `CAMLRUNPARAM`) environment variables, and we would still like to be able to run the test, despite changes to these. We can now phrase `init_state` as follows, where we fall back on `default_control`, an (omitted) record of default values. For brevity we also omit the definitions of `parse_params` and `interpret_params`, which parse and interpret environment variables, respectively:

```
let init_state =
  let control =
    if Sys.runtime_variant () = "d"
    then (* -runtime-variant=d causes verbose=63 *)
      { default_control with Gc.verbose = 63 }
    else default_control in
  let params =
    try Sys.getenv "OCAMLRUNPARAM" with Not_found ->
    try Sys.getenv "CAMLRUNPARAM" with Not_found ->
    "" in
  interpret_params (parse_params params) control
```

We will use a fixed hard-coded length `array_length` as upper bound of our indices. This is sufficiently small to trigger generation of duplicate indices, thus leading to overwrites, and thereby turning allocated records unreachable:

```
let array_length = 8
```

As we seek to test the GC under both explicit invocations, e.g., with `Gc.minor ()` as well as under implicit usage, we define two generators `alloc_cmds` and `gc_cmds`. The former is a generator of allocating commands, whereas the latter is a generator of either allocating commands or explicit `Gc` invoking commands. We include an outline of the two definitions below. As `Stat`, `QuickStat`, `Minor_words`, `Get`, `Allocated_bytes`, and `Get_minor_free` are purely observational (non GC-side-effecting) we include them with the former. To get a spread of minor heap sizes, we use bit-shifts in `minor_heap_size_gen` to generate powers of 2 from 512 to 32768 for `Set Minor_heap_size`. Similarly, to get a spread of string sizes, we use bit-shifts in `str_len_gen` to generate powers of 2 between 0 and 8196, and subsequently use it in both `PreAllocStr` and `AllocStr` to generate a string (via `str_gen`) and a string length, respectively. Indices are generated in `index_gen` wrt. our upper index bound `array_length`. In all of these we utilize QCheck's combinators `Gen.int_bound` and `Gen.int_range` for generating bounded integers:

```
let alloc_cmds =
  let open Gen in
  (* ... *)
  let str_len_gen =
    map (fun s -> 1 lsl (s-1)) (int_bound 14) in
  let str_gen =
    map (fun l -> String.make l 'x') str_len_gen in
  (* ... *)
  let index_gen = int_bound (array_length-1) in
  [ (* purely observational cmds *)
    1,return Stat;
    1,return Quick_stat;
    (* ... *)
    (* allocating cmds to activate the Gc *)
    (* ... *)
    5,map2 (fun i s -> PreAllocStr (i,s))
           index_gen str_gen;
    5,map2 (fun i l -> AllocStr (i,l))
           index_gen str_len_gen;
    (* ... *)
  ]

let gc_cmds =
  let open Gen in
  let minor_heap_size =
    map (fun s -> 1 lsl s) (int_range 9 15) in
  let space_overhead = int_range 20 200 in
  (* ... *)
  [ 1,map (fun i -> Set (Minor_heap_size i))
          minor_heap_size;
    1,map (fun i -> Set (Space_overhead i))
          space_overhead;
    (* ... *)
    1,return Full_major;
    1,return Compact;
  ] @ alloc_cmds
```

Based on these we can now phrase two corresponding variant definitions of `arb_cmd`:

```
let arb_cmd _s = QCheck.make ~print:show_cmd
                   (Gen.frequency gc_cmds)

let arb_alloc_cmd _s = QCheck.make ~print:show_cmd
                         (Gen.frequency alloc_cmds)
```

As our system under test we use a collection of reachable 64-bit integers, strings, lists, and bigarrays. We store these in a record with mutable fields and update entries underway, potentially allocating new memory records and making others unreachable and hence target for GC. In `init_sut` we can now initialize the record fields accordingly, while in `cleanup` we reset them, reset the GC parameters to their initial value with `Gc.set`, and invoke an explicit full major GC to reset the heap to a relatively clean state:

```
type sut =
  { mutable int64s  : int64 list;
    mutable strings : string array;
    mutable lists   : char list array;
    mutable bigarrays :
    (int, BA.int_elt, BA.c_layout) BA.Array1.t array
  }

let init_sut () =
  { int64s = [];
    strings = Array.make array_length "";
    lists   = Array.make array_length [];
    bigarrays =
      Array.make array_length
        BA.(Array1.create int C_layout 0)
  }

let cleanup sut =
  begin
    sut.int64s <- [];
    sut.strings <- [| |];
    sut.lists <- [| |];
    sut.bigarrays <- [| |];
    Gc.set init_state;
    Gc.full_major ()
  end
```

The `run` function will interpret the commands in a straightforward manner. Note how `PreAllocStr` updates the entry for an already allocated string, whereas `AllocStr` allocates a string when `run` is executed. In `Set` commands we first call `Gc.get` to get an up-to-date `control`-record and use it to update the corresponding field, thereby changing only one GC parameter at a time:

```
let run c sut = match c with
  | Stat        -> Res (gcstat, Gc.stat ())
  | Quick_stat  -> Res (gcstat, Gc.quick_stat ())
  (* ... *)
  | PreAllocStr (i,s) ->
    Res (unit, sut.strings.(i) <- s)
  | AllocStr (i,len) ->
    Res (unit,
         sut.strings.(i) <- String.make len 'c')
  (* ... *)
  | Set subcmd -> (match subcmd with
      | Minor_heap_size i ->
        let prev = Gc.get () in
        Res (unit,
          Gc.set { prev with minor_heap_size = i })
      | Space_overhead i ->
        let prev = Gc.get () in
        Res (unit,
          Gc.set { prev with space_overhead = i })
      (* ... *)
    )
  (* ... *)
  | Full_major  -> Res (unit, Gc.full_major ())
  | Compact     -> Res (unit, Gc.compact ())
```

Finally, we leave the pre-condition as constantly true, whereas the post-condition will perform a range of sanity checks. For example, `Stat` and `QuickStat` use a helper function to check the entries of the returned record, e.g., expecting `minor_words` denoting the number of words allocated in the minor heap to be non-negative, and expecting `heap_chunks` to be constantly zero, as documented under OCaml 5.

```
let precond _n _s = true

let check_gc_stats r =
  r.Gc.minor_words >= 0. &&
  r.Gc.promoted_words >= 0. &&
  r.Gc.major_words >= 0. &&
  r.Gc.minor_collections >= 0 &&
  r.Gc.major_collections >= 0 &&
  r.Gc.heap_words >= 0 &&
  r.Gc.heap_chunks = 0 &&
  (* ... *)

let postcond n (s: state) res = match n, res with
  | Stat, Res ((GcStat,_),r) ->
    check_gc_stats r
  | Quick_stat, Res ((GcStat,_),r) ->
    check_gc_stats r
  (* ... *)
  | PreAllocStr _, Res ((Unit,_), ()) -> true
  | AllocStr _, Res ((Unit,_), ()) -> true
  (* ... *)
  | Set _,      Res ((Unit,_), ()) -> true
  | Full_major, Res ((Unit,_), ()) -> true
  | Compact,    Res ((Unit,_), ()) -> true
  | _, _ -> false
```

4.3 Running the GC Tests

Recall that we seek to test the GC under both explicit invocations, e.g., with `Gc.minor ()` as well as under implicit usage. With our specification in `Stm_tests_spec` containing both a definition of `arb_alloc_cmd` generating only allocation commands, and `arb_cmd` generating both allocation commands and explicit `Gc` calls, a regular functor instantiation of `Stm_tests_spec` will allow us to trigger explicit usage tests, whereas a simple shadowing of the `arb_cmd` definition will allow us to instantiate functors with `ImplGCConf` for implicit usage testing:

```
module ImplGCConf =
struct
  include Stm_tests_spec
  let arb_cmd = arb_alloc_cmd
end
```

With the GC as our SUT and a test runner that also depends on it, it becomes even more important to isolate the individual PBTs from each other: An initial sequential PBT run could potentially leave the heap in a bad state, which would then influence, e.g., a subsequent parallel PBT, thereby challenging the isolated reproducibility of any issues observed during the latter. To address this, we run each GC PBT (sequential, parallel, stress test) as separate test executables, each starting a new OCaml process with a fresh heap. We still reuse the same specification across the three variants, which just amounts to 3 different functor instantiations with the same argument module.

With dynamic changes to `stack_limit` during test case execution, a `Stack_overflow` exception becomes acceptable test case behavior. We handle this situation by wrapping the tested property in an exception handler and returning `true` to reflect a passed test case.

5 Evaluation

Since before the release of OCaml 5.0.0, we have been running continuous integration (CI) workflows with the test suite and continue to do so. These workflows run PBTs of OCaml's standard library, including the described `Array`, weak hash set, and `Gc` tests, and run on a variation of

- configurations (native compilation, bytecode compilation, with the debug runtime, 32-bit builds, enabling framepointers, building with the musl-C library),
- hardware (testing AMD64, ARM64, and earlier also on PPC64, s390x), and
- operating systems (Linux, macOS, Windows in the Cygwin, MinGW, and MSVC variants, and earlier on FreeBSD too)

5.1 Bugs Found

Collectively, the test suite has found over 40 issues at the time of writing.[2] Below we focus only on the 8 issues found using the described `Array`, weak hash set, and `Gc` tests. As the OCaml runtime continues to evolve, as does the PBT test suite, these numbers may increase further.

Array. We initally observed that the sequential tests of `Array` would trigger segmentation faults on the PPC64 architecture. This turned out to be caused by this architecture's array bound checks, which unlike the other backends were signal-based, and could cause stack corruption upon indexing out of bounds.[3] It thus proved central to test the out-of-bounds behavior too.

Weak Hash Set. The parallel tests found a combination of weak hash set functions that could cause the runtime system to abort or err with a segmentation fault. This turned out to be caused by race conditions in the `ephe_get_field_copy` function underlying weak hash sets.[4]

Garbage Collector. Our third and longest case study has been the most successful of the three. As the GC test was built incrementally, as soon as we added a `cmd` targeting `Gc.counters` a test run would trigger segmentation faults. We identified this as a known issue, due to improper GC root registration in the runtime's underlying `caml_gc_counters` function.[5]

Secondly, the GC tests would trigger an assertion error when run under the debug runtime:

```
[00] file runtime/shared_heap.c; line 1392
### Assertion failed:
local->stats.pool_live_words == pool_stats.live
```

This also turned out to be a known issue,[6] albeit one which had not been reproduced. Shrinking here helped create a standalone reproducer, revealing that stats were not updated correctly during GC compaction. This was a hard to reproduce case, as it required garbage to remain in the major heap despite 3 rounds of major GC (itself a prerequisite for compaction) to trigger.

We later found outdated documentation for `Gc.quick_stat` which errorneously stated that it would return a `stat` record with six zero fields. This was due to some fields being abandoned by OCaml 5 while others were gradually being restored.[7] Similarly the GC tests found outdated documentation for the `Gc.control` record. In this case the tests identified 3 record fields which were not supported on OCaml 5 and hence constantly zero, as well as an outdated default value for `stack_limit`.[8]

A fifth bug found would trigger crashes under OCaml's bytecode compiler. This one turned out to be caused by dangling bytecode fragments. Under the right circumstances these would cause later values reusing the same heap location to be garbage collected, despite being live, and hence wrecking havoc.[9]

Lastly, we observed that the explicit GC test containing `Gc.compact` could trigger segmentation faults or deadlocks, initally on macOS ARM64-based test machines. More investigation revealed that a central precondition for GC compaction could be invalidated if a domain would terminate simultaneously with a major heap compaction. This was caused by the terminating domain asynchronously adding orphaned shared heap pools to a global `pool_freelist`, which

[2] https://github.com/ocaml-multicore/multicoretests#issues
[3] https://github.com/ocaml/ocaml/pull/12540
[4] https://github.com/ocaml/ocaml/issues/11934
[5] https://github.com/ocaml/ocaml/pull/13370
[6] https://github.com/ocaml/ocaml/issues/13090
[7] https://github.com/ocaml/ocaml/pull/13424
[8] https://github.com/ocaml/ocaml/pull/13440
[9] https://github.com/ocaml/ocaml/pull/13549

would risk not getting updated during compaction, and thus leave an invalid heap with broken pointers.[10]

5.2 Bug Discussion

Having outlined our findings, we now take a step back and investigate the nature of the found defects. The 8 defects distribute as follows:

- 2 were out of date documentation,
- 2 were related to code generation (of PPC64 and bytecode) and its interaction with the runtime system,
- 1 was due to a missing GC root registration,
- 1 was an assertion error, and
- 2 were due to race conditions

Half of the above used sequential tests to trigger, and the other half required parallel testing to trigger. The latter four all required rare conditions to trigger, leading an OCaml compiler core developer to characterize the approach as "manufacturing bad luck on an industrial scale".[11]

Interestingly the compaction race condition was not a *data race*, i.e., two uncoordinated accesses to the same memory location, with at least one of them being a write, meaning that it would not have been caught using a developer tool such as ThreadSanitizer [34]. Rather the race was logical: The global `pool_freelist` was intended to be empty for compaction to function correctly, and changes to it were coordinated via a protecting mutex.

Of these 8 issues, 1 was known beforehand and reproducible, 1 was known and had not been reproduced. As such, the tests have indeed successfully targeted OCaml 5's runtime system and found previously unnoticed errors.

5.3 Challenges and Limitations

We have repeatedly triggered issues causing a deadlock or an infinite loop, causing a test suite run to make no progress. In such situations, a timeout on our CI workflows has helped. Some platforms are however slower than others, which means we use a higher timeout for, e.g., Windows, that additionally requires a Cygwin installation, and generally take longer to complete the test suite.

When an issue is fixed, e.g., in the development branch of the OCaml compiler, it may still continue to trigger in the latest official release, e.g., 5.3.0, until the next official version is released with the fix. This adds noise, as any failure from a CI run, warrants investigation of the cause. Once such issues have been settled, we have therefore developed a practice of disabling the test features triggering them for all affected OCaml versions.

An issue we have found repeatedly, is that a runtime system crash takes down the test runner too. To solve this, we developed a combinator `Util.fork_prop_with_timeout : int -> ('a -> bool) -> 'a -> bool` accepting a timeout bound (in seconds) and a property, which will run the property in a child process (thus protecting the parent from crashes), and fail if the property does not complete within the desired bound.

The `Util.fork_prop_with_timeout` combinator has been helpful in some cases to trigger proper test case shrinking despite hitting hard errors or crashes, e.g., for the GC `stat` assertion error. In the most challenging cases however, such as the compaction race condition, we were not able to trigger the crash when run with `Util.fork_prop_with_timeout` in a forked child process, due to the specific shared heap setup required. Such situations has thus called for other strategies, such as studying the target's source code or manually reducing the generator, e.g., to omit particular commands.

There is a clear limitation as to the kind of GC issues one can find by running relatively shortlived PBTs. For example, our tests will not be able to catch memory leaks, which require both longer runs as well as memory usage observations. Finally, as useful as PBT is, triggering a test error is only the first step. Subsequently one needs to analyze and understand the involved code and conditions causing an issue, and only then potentially develop a fix.

6 Related Work

The term *property-based testing* was initially coined for a dedicated white-box testing approach that included a domain-specific language for writing property specifications along with a static dataflow analysis to guide test executions [16]. Claessen and Hughes [6] introduced QuickCheck, as two embedded domain-specific languages (EDSLs) for Haskell: one EDSL for expressing testable properties, and another combinator EDSL of type-based generators for constructing samplable test case distributions. The QuickCheck approach has since been ported to numerous other programming languages [37]. Test input generation is not limited to random sampling. SmallCheck [33] uses an alternative approach based on generators that instead enumerate test inputs up to some bound, typically in an order from smaller to larger test inputs. QuickCheck and SmallCheck inspired systems have since come to be known collectively under the common *property-based testing* name.

To test imperative, monadic Haskell code, Claessen and Hughes [7] proposed a range of approaches, including algebraic specifications, model-based specifications, and pre- and post-conditions. Koopman and Plasmeijer [22] then proposed model-based state machine specifications in the Gast testing library for Clean. Erlang's `eqc_commands` module was first described by Arts et al. [3] and put into use to test the implementation of a telecommunication protocol. A later revision is described by Hughes [18]. Koopman et al. [21] conduct an experiment comparing the error-finding ability

[10] https://github.com/ocaml/ocaml/pull/14025
[11] https://github.com/ocaml/ocaml/pull/13549

of logical properties compared to state machines. They conclude that the latter is stronger and that such specifications are also easier to write.

The extention of using model-based state machines to test for race conditions is initially described briefly by Hughes [18], Sec. 8, which mentions running 2 parallel command sequences and a property searching for an interleaving. The approach is further refined with a custom randomized scheduler and process visualizer by Claessen et al. [8]. Hughes and Bolinder [20] present a later case study applying the approach to Erlang's `dets` term storage module, underlying a larger database system. In doing so, they find five race conditions with relatively little effort, while avoiding the need for a custom randomized scheduler.

Only a few prior works have investigated testing of a garbage collector, as in this paper's biggest case study. Polito et al. [31] used an approach of generating random heap events targeting the Pharo Virtual Machine, to fuzz test its sequential stop-the-world generational GC. Their approach directly targets the GC, thereby circumventing a JIT compiler. In contrast, we run tests under OCaml's native compiler generating operations that will invoke OCaml's GC. Similarly they use assertions and heap invariant checks to drive the GC fuzz test, and, like us, they were able to trigger both ephemeron and GC compactor bugs. Whereas Pharo supports concurrent programming through green threads, it's runtime system is not multicore like OCaml 5's.

Zheng et al. [39] develop event-coverage of GC events (e.g., minor and major heap changes) as coverage criteria for guiding fuzz testing of JVM GCs. They implement *GCFuzz*, based on 4 GC-related code generation rules, the output of which is injected into programs selected from a pool of seed programs. Based on this approach they are able to detect a range of previously unknown bugs in the Hotspot and OpenJDK JVMs. Overall Zheng et al.'s approach is based on grey-box fuzzing, whereas we use a black-box PBT approach. The former optimizes for reaching a higher coverage, whereas the latter is more light weight and thus more broadly applicable. Furthermore, whereas grey-box fuzzing is effective at locating bugs, our previous work has revealed that the approach is also sensitive to the seed programs, and may therefore miss bugs found by a black-box generation approach [30].

Within the research community, there have been several works on testing language processors: Purdom [32] developed early techniques for testing parsers and context-free grammars. McKeeman [25] later used the term *differential testing*, to characterize an approach for testing compilers against each other. His approach involved a stochastic grammar associating weights to each production.

To pass a strong static type check, Pałka et al. [29] proposed to generate well-typed lambda terms following a goal-directed reading of the typing rules. They used the approach to test the GHC Haskell compiler's strictness analyzer.

Yang et al. [38] devised *CSmith* — a custom generator of C programs free of C's undefined behaviour and used it to differentially test the output of different C compiler against each other. Le et al. [24] introduced *equivalence modulo inputs* (EMI), an alternative approach to test compilers: With EMI one generates pairs of programs that are equivalent on the same input. This approach unveiled another significant class of C compiler bugs. Donaldson et al. [15] later utilized EMI to test graphics shader compilers, where semantics-preserving transformations should result in similarly rendered images.

Midtgaard et al. [26] present a variant of Pałka et al.'s generator that produce well-typed, evaluation-order independent OCaml programs, which they used for differential testing of OCaml's native code and byte code compiler backends. Perényi and Midtgaard [30] present a stack-driven generator of type correct WebAssembly programs, following the stack-discipline of the specification's typing rules.

A separate line of work developed means to test static and dynamic analyzers: Hriţcu et al. [17] investigated different generation strategies to test a dynamic non-interference analysis of a simple stack machine, finding that some custom generation is required to reliably find counterexamples to the most tricky analysis bugs. Cuoq et al. [9] utilized the CSmith C program generator to test the Frama-C static analyzer for various properties, incl. stress testing it for crashes, turning the approximate value analysis into a non-approximating interpreter, and extracting invariants into checkable assertions. Midtgaard and Møller [27] utilized PBT to test abstract domains for lattice-related static analysis properties, and developed DSLs to write such tests concisely. The properties from the latter were later reused by Bugariu et al. [4] to drive grey-box fuzzing of a number of existing abstract domains implementations.

7 Conclusion

We have described how property-based testing can successfully be utilized to corner subtle bugs in a multicore runtime system. Our case studies illustrate that the model-based state machine approach extends well to also handle the intricacies of this setting. We have found 8 bugs across the 3 case studies covered in this paper, all of which have now been fixed over the course of OCaml releases 5.0.0 to 5.4.0. Our effort has thus contributed to strengthening the still relatively new OCaml 5 runtime system.

The test suite is open source under a 2-clause BSD license, and is available from the following URL:

https://github.com/ocaml-multicore/multicoretests

Acknowledgments

The author would like to express his sincere gratitude to Olivier Danvy, as a lecturer, advisor, and friend. Olivier opened the author's eyes to the academic world and to PL research, offering a nutritious and enlightening diet of Calvin

and Hobbes, science fiction, functional programming, interpreters, abstract machines, and continuations, among others. After 25 years the mantra "returning means applying the continuation" still sticks. That one can utilize the computer science skills taught and investigated by Olivier throughout his career to drive tests of a complex multi-threaded runtime system is testament to their relevantness.[12] Finally, Olivier's influence via presentation and writing advise, e.g., learning to avoid the passive voice, is highly appreciated. Congratulations Olivier—and thank you.

The full `multicoretests` suite was built with the help of Samuel Hym, Nicolas Osborne, Olivier Nicole, Charlène Gros, and Naomi Spargo. The author is also grateful to the anonymous reviewers for constructive feedback and to Miod Vallat and the OCaml compiler maintainers for their help fixing the reported bugs.

References

[1] Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. 2003. A Functional Correspondence between Evaluators and Abstract Machines. In *PPDP'03: Proc. of the 5th ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming* (Uppsala, Sweden), Dale Miller (Ed.). 8–19. https://doi.org/10.1145/888251.888254

[2] Mads Sig Ager, Olivier Danvy, and Jan Midtgaard. 2005. A Functional Correspondence between Monadic Evaluators and Abstract Machines for Languages with Computational Effects. *Theoretical Computer Science* 342, 1 (2005), 149–172. https://doi.org/10.1016/j.tcs.2005.06.008 Extended version available as the technical report BRICS RS-04-28.

[3] Thomas Arts, John Hughes, Joakim Johansson, and Ulf Wiger. 2006. Testing Telecoms Software with Quviq QuickCheck. In *Proceedings of the 5th ACM SIGPLAN workshop on Erlang, Portland, Oregon, September 16, 2006*. https://doi.org/10.1145/1159789.1159792

[4] Alexandra Bugariu, Valentin Wüstholz, Maria Christakis, and Peter Müller. 2018. Automatically Testing Implementations of Numerical Abstract Domains. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18)*. 768–778. https://doi.org/10.1145/3238147.3240464

[5] Guillaume Bury, Simon Cruanes, Vincent Hugot, and Jan Midtgaard. 2013. QCheck: QuickCheck Inspired Property-Based Testing for OCaml. GitHub. https://github.com/c-cube/qcheck

[6] Koen Claessen and John Hughes. 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Proc. of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP'00)*, Philip Wadler (Ed.). 53–64. https://doi.org/10.1145/351240.351266

[7] Koen Claessen and John Hughes. 2002. Testing Monadic Code with QuickCheck. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell (Haskell 2002)*. 65–77. https://doi.org/10.1145/581690.581696

[8] Koen Claessen, Michał Pałka, Nicholas Smallbone, John Hughes, Hans Svensson, Thomas Arts, and Ulf Wiger. 2009. Finding Race Conditions in Erlang with QuickCheck and PULSE. In *Proc. of the 14th ACM SIGPLAN International Conference on Functional Programming (ICFP'09)*, Graham Hutton and Andrew P. Tolmach (Eds.). 149–160. https://doi.org/10.1145/1596550.1596574

[9] Pascal Cuoq, Benjamin Monate, Anne Pacalet, Virgile Prevosto, John Regehr, Boris Yakobowski, and Xuejun Yang. 2012. Testing Static Analyzers with Randomly Generated Programs. In *NASA Formal Methods - 4th International Symposium, NFM 2012. Proc. (LNCS, Vol. 7226)*. 120–125. https://doi.org/10.1007/978-3-642-28891-3_12

[10] Olivier Danvy. 1996. Type-directed Partial Evaluation. In *Proc. of the 23rd Annual ACM Symposium on Principles of Programming Languages*, Guy L. Steele Jr. (Ed.). St. Petersburg Beach, Florida, 242–257. https://doi.org/10.1145/237721.237784

[11] Olivier Danvy. 1998. Functional Unparsing. *Journal of Functional Programming* 8, 6 (1998), 621–625. https://doi.org/10.1017/S0956796898003104

[12] Olivier Danvy. 2000. Formalizing Implementation Strategies for First-Class Continuations. In *Proc. of the Ninth European Symposium on Programming (LNCS, Vol. 1782)*, Gert Smolka (Ed.). Springer-Verlag, 88–103. https://doi.org/10.1007/3-540-46425-5_6

[13] Olivier Danvy and Karoline Malmkjær. 1988. Intensions and Extensions in a Reflective Tower. In *Proceedings of the 1988 ACM Conference on LISP and Functional Programming (LFP '88)*. 327–341. https://doi.org/10.1145/62678.62725

[14] Olivier Danvy and Kevin Millikin. 2008. On the Equivalence between Small-Step and Big-Step Abstract Machines: A Simple Application of Lightweight Fusion. *Inform. Process. Lett.* 106, 3 (2008), 100–109. https://doi.org/10.1016/j.ipl.2007.10.010

[15] Alastair F. Donaldson, Hugues Evrard, Andrei Lascu, and Paul Thomson. 2017. Automated Testing of Graphics Shader Compilers. *PACMPL* 1, OOPSLA (2017), 93:1–93:29. https://doi.org/10.1145/3133917

[16] George Fink and Matt Bishop. 1997. Property-Based Testing: A New Approach to Testing for Assurance. *SIGSOFT Softw. Eng. Notes* 22, 4 (1997), 74–80. https://doi.org/10.1145/263244.263267

[17] Cătălin Hriţcu, John Hughes, Benjamin C. Pierce, Antal Spector-Zabusky, Dimitrios Vytiniotis, Arthur Azevedo de Amorim, and Leonidas Lampropoulos. 2013. Testing Noninterference, Quickly. In *Proc. of the 18th ACM SIGPLAN International Conference on Functional Programming (ICFP'13)*, Greg Morrisett and Tarmo Uustalu (Eds.). 455–468. https://doi.org/10.1145/2500365.2500574

[18] John Hughes. 2007. QuickCheck Testing for Fun and Profit. In *Practical Aspects of Declarative Languages, 9th International Symposium, PADL 2007 (LNCS, Vol. 4354)*. 1–32. https://doi.org/10.1007/978-3-540-69611-7_1

[19] John Hughes. 2016. Experiences with QuickCheck: Testing the Hard Stuff and Staying Sane. In *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday (LNCS, Vol. 9600)*, Sam Lindley, Conor McBride, Philip W. Trinder, and Donald Sannella (Eds.). Springer-Verlag, 169–186. https://doi.org/10.1007/978-3-319-30936-1_9

[20] John Hughes and Hans Bolinder. 2011. Testing a Database for Race Conditions with QuickCheck. In *Proceedings of the 10th ACM SIGPLAN workshop on Erlang, 2011*. 72–77. https://doi.org/10.1145/2034654.2034667

[21] Pieter W. M. Koopman, Peter Achten, and Rinus Plasmeijer. 2012. Model Based Testing with Logical Properties versus State Machines. In *Implementation and Application of Functional Languages - 23rd International Symposium, IFL 2011, Revised Selected Papers (LNCS, Vol. 7257)*, Andy Gill and Jurriaan Hage (Eds.). Springer, 116–133. https://doi.org/10.1007/978-3-642-34407-7_8

[22] Pieter W. M. Koopman and Rinus Plasmeijer. 2005. Testing Reactive Systems with GAST. In *Revised Selected Papers from the Fourth Symposium on Trends in Functional Programming, TFP 2003, Edinburgh, United Kingdom, 11-12 September 2003. (Trends in Functional Programming, Vol. 4)*, Stephen Gilmore (Ed.). Intellect, 111–129.

[23] Pieter W. M. Koopman and Rinus Plasmeijer. 2011. Testing with Functional Reference Implementations. In *Trends in Functional Programming - 11th International Symposium, TFP 2010. Revised Selected Papers (LNCS, Vol. 6546)*, Rex L. Page, Zoltán Horváth, and Viktória Zsók (Eds.). Springer, 134–149. https://doi.org/10.1007/978-3-642-22941-1_9

[12] This is a geeky reference and it is not a typo.

[24] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler Validation via Equivalence Modulo Inputs. In *Proc. of the ACM SIGPLAN 2014 Conference on Programming Languages Design and Implementation, PLDI'14*, Michael F. P. O'Boyle and Keshav Pingali (Eds.). 216–226. https://doi.org/10.1145/2594291.2594334

[25] William M. McKeeman. 1998. Differential Testing for Software. *Digital Technical Journal* 10, 1 (1998), 100–107.

[26] Jan Midtgaard, Mathias Nygaard Justesen, Patrick Kasting, Flemming Nielson, and Hanne Riis Nielson. 2017. Effect-Driven QuickChecking of Compilers. *PACMPL* 1, ICFP (2017), 15:1–15:23. https://doi.org/10.1145/3110259

[27] Jan Midtgaard and Anders Møller. 2017. Quickchecking Static Analysis Properties. *Software Testing, Verification and Reliability* 27, 6 (2017). https://doi.org/10.1002/stvr.1640 A preliminary version was presented at the 8th IEEE International Conference on Software Testing, Verification and Validation (ICST'15).

[28] Jan Midtgaard, Olivier Nicole, and Nicolas Osborne. 2022. Multicoretests — Parallel Testing Libraries for OCaml 5.0. In *OCaml Users and Developers Workshop*. https://github.com/ocaml-multicore/multicoretests

[29] Michał H. Pałka, Koen Claessen, Alejandro Russo, and John Hughes. 2011. Testing an Optimising Compiler by Generating Random Lambda Terms. In *Proc. of the 6th International Workshop on Automation of Software Test, AST 2011*. 91–97. https://doi.org/10.1145/1982595.1982615

[30] Árpád Perényi and Jan Midtgaard. 2020. Stack-Driven Program Generation of WebAssembly. In *18th Asian Symposium on Programming Languages and Systems, APLAS 2020, Proceedings (LNCS, Vol. 12470)*, Bruno C. d. S. Oliveira (Ed.). 209–230. https://doi.org/10.1007/978-3-030-64437-6_11

[31] Guillermo Polito, Pablo Tesone, Nahuel Palumbo, Stéphane Ducasse, and Jean Privat. 2023. Heap Fuzzing: Automatic Garbage Collection Testing with Expert-Guided Random Events. In *2023 IEEE Conference on Software Testing, Verification and Validation (ICST)*. 107–116. https://doi.org/10.1109/ICST57152.2023.00019

[32] Paul Purdom. 1972. A Sentence Generator for Testing Parsers. *BIT* 12, 3 (1972), 366–375. https://doi.org/10.1007/BF01932308

[33] Colin Runciman, Matthew Naylor, and Fredrik Lindblad. 2008. SmallCheck and Lazy SmallCheck: Automatic Exhaustive Testing for Small Values. In *Proc. of the 1st ACM SIGPLAN Symposium on Haskell, Haskell 2008*, Andy Gill (Ed.). 37–48. https://doi.org/10.1145/1411286.1411292

[34] Konstantin Serebryany and Timur Iskhodzhanov. 2009. ThreadSanitizer: Data Race Detection in Practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications (WBIA '09)*. 62–71. https://doi.org/10.1145/1791194.1791203

[35] Ilya Sergey. 2016. Experience Report: Growing and Shrinking Polygons for Random Testing of Computational Geometry Algorithms. In *Proc. of the 21st ACM SIGPLAN International Conference on Functional Programming (ICFP'16)*, Jacques Garrigue, Gabriele Keller, and Eijiro Sumii (Eds.). 193–199. https://doi.org/10.1145/2951913.2951927

[36] KC Sivaramakrishnan, Stephen Dolan, Leo White, Sadiq Jaffer, Tom Kelly, Anmol Sahoo, Sudha Parimala, Atul Dhiman, and Anil Madhavapeddy. 2020. Retrofitting Parallelism onto OCaml. *Proc. ACM Program. Lang.* 4, ICFP, Article 113 (Aug. 2020), 30 pages. https://doi.org/10.1145/3408995

[37] Wikipedia. 2025. Wikipedia QuickCheck page. https://en.wikipedia.org/wiki/QuickCheck Accessed: 2025-06-04.

[38] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *Proc. of the ACM SIGPLAN 2011 Conference on Programming Languages Design and Implementation, PLDI'11*, David Padua (Ed.). 283–294. https://doi.org/10.1145/1993498.1993532

[39] Kai Zheng, Zan Wang, Yingquan Zhao, Junjie Chen, Hanmo You, Haoyu Wang, Yiheng Du, and Tianchang Gao. 2025. Exploring JVM Garbage Collector Testing with Event-Coverage. *ACM Trans. Softw. Eng. Methodol.* (May 2025). https://doi.org/10.1145/3733598

Received 2025-06-11; accepted 2025-07-31