A Simple State-Machine Framework for Property-Based Testing in OCaml

Jan Midtgaard
The Maersk Mc-Kinney Moller Institute
University of Southern Denmark

1. Introduction
Since their inception [3, 1] state-machine frameworks have proven their worth by finding defects in everything from the underlying AUTOSAR components of Volvo cars to digital invoicing systems [2]. These case studies were carried out with Erlang’s commercial QuickCheck state-machine framework from Quviq, but such frameworks are now also available for Haskell, F#, Scala, Elixir, Java, etc. We present a typed state-machine framework for OCaml based on the QCheck library and illustrate a number of concepts common to all such frameworks: state modeling, commands, interpreting commands, preconditions, and agreement checking.

2. Property-based testing with QCheck
QCheck is a property-based testing library for OCaml. Consider the following example:

```ocaml
open QCheck
let t =
  Test.make (list small_nat)
  (fun xs -> List.of_seq (List.to_seq xs) = xs);
QCheck_runner.run_tests ~verbose:true [t];
```

Test.make expects a generator producing test input and a property that each test input should satisfy. We use QCheck’s combinators to build a generator of integer lists and then test that each list survives a round trip conversion through sequences.

Generators in QCheck are of type ‘a arbitrary. This is defined as a composite record type of “full” generators, which includes both an underlying “pure” generator of type ‘a Gen.t, an optional print function, and an optional shrinker:

```ocaml
type ‘a arbitrary = {
  gen : ‘a Gen.t;
  print : (‘a -> string) option;
  shrink : ‘a Shrink.t option;
  (* ... *)
}
```

The submodule Gen also offers a combinator library for building composite pure generators.

3. Testing hashtables
As an example, consider the hashtable implementation from the standard library. We recall a minimal selection of the hashtable interface in Fig. 1. If we are to test this imperative interface using property-based testing, one option is to generate an arbitrary sequence of (symbolic) hashtable operations and ensure that the outcome of each operation is as expected. A common method to phrase expectation in this context is by using a model: an idealized, declarative specification of the imperative API.

```ocaml
val create : ?random:bool -> int -> (‘a, ‘b) Hashtbl.t
val add : (‘a, ‘b) Hashtbl.t -> ‘a -> ‘b -> unit
val remove : (‘a, ‘b) Hashtbl.t -> ‘a -> unit
val find : (‘a, ‘b) Hashtbl.t -> ‘a -> ‘b
```

Figure 1: Selected operations from the Hashtbl interface

3.1 Commands and command generators
OCaml’s hashtables are polymorphic. To test them we need to choose concrete key and value types. Somewhat arbitrarily, we choose char as our key type and int as our value type. With this type choice in mind, a symbolic hashtable operation can now be represented as an algebraic datatype:

```ocaml
type cmd =
  | Add of char * int
  | Remove of char
  | Find of char @(deriving show)
```

Here we utilize a ppx-deriving preprocessor to automatically derive a printer show_cmd : cmd -> string.

Based on the data type definition we can now write a straightforward generator of commands. The generator chooses between each of the three commands and is phrased in terms of a character generator char_gen that generates arbitrary characters:

```ocaml
let gen_cmd =
  Gen.map2 (fun k v -> Add (k,v))
  gen_cmd char_gen Gen.small_nat;
Gen.map (fun k -> Remove k) char_gen;
Gen.map (fun k -> Find k) char_gen;
```

When combined with the printer show_cmd we can now form a full generator of arbitrary commands:

```ocaml
let arb_cmd =
  Gen.make ~print:show_cmd arb_cmd Gen.t
```

3.2 Model and model interpretation
We can model a hashtable with character keys and integer values by an association list of char * int pairs:

```ocaml
type state = (char * int) list
```

This type can naturally model the internal state of a hashtable, in the form of a collection of char keys and associated int values. Based on this model, it is straightforward to write an interpreter:

```ocaml
let next_state c s =
  match c with
  | Add (k,v) -> Add (k,v)::s
  | Remove k -> List.remove_assoc k s
  | Find _ -> s
```

Interpreting an Add command adds the key-value pair to the association list, whereas Remove deletes the first occurrence of key k using List.remove_assoc. This faithfully models how adding an entry with an existing key shadows any previous entries. In the Find case the state is returned unmodified since the operation has no effect on a hashtable’s internal state.

3.3 Interpreting commands and verifying the output
We still need to interpret the symbolic commands over the actual system under test (sut) and to verify that any output returned is as expected. We perform these two tasks with a function run_cmd:
type sut = (char, int) Hashtbl.t

(* run_cmd : cmd -> state -> sut -> bool *)
let run_cmd c s h = match c with
  | Add (k,v) -> Hashtbl.add h k v; true
  | Remove k -> Hashtbl.remove h k; true
  | Find k -> List.assoc_opt k s
     = (try Some (Hashtbl.find h k)
           with Not_found -> None)

Since Add and Remove have return type unit, there is no output to verify and we therefore simply return true. In the Find case we verify that the output agrees with the corresponding operation over the model's association list. We do so by relying on assoc_opt from the List module.

4. From commands to command lists

So far, we have combined three types: (1) a type of commands, (2) a system under test (hashables), and (3) a model of the system's state (association lists) with operations for interpreting a command over the model and interpreting a command over the system under test and ensuring agreement. We now consider a common interface for phrasing such state-machine tests:

module type StmSpec =
  sig
  type cmd
  type state
  type sut
  val arb_cmd : state -> cmd arbitrary
  val init_state : state
  val next_state : cmd -> state -> state
  val init_sut : unit -> sut
  val cleanup : sut -> unit
  val run_cmd : cmd -> state -> sut -> bool
  val precond : cmd -> state -> bool
  val precond : cmd -> state -> bool
  val next_state : cmd -> state -> state
end

The operation arb_cmd returns a full command generator. It accepts a state parameter to enable state-dependent cmd generation. It is furthermore phrased as a full generator, to allow an optional cmd printer and shrinker to be provided. For example, using this setup we can revise char_gen to increase the chance of generating an existing key to addr, remove, or find. We do so by choosing an existing key from s with probability \(\frac{1}{2}\):

(* gen_cmd : state -> cmd Gen.t *)
let gen_cmd s =
  let char_gen =
    if s = []
    then Char.gen
    else
      let keys = List.map fst s in
      Gen.choice [Gen.oneof keys;
                   Gen.char]
  Gen.choice
      (* ... (unchanged) *)
    (* arb_cmd : state -> cmd arbitrary *)
    let arb_cmd s =
      QCheck.make ~print:show_cmd (gen_cmd s)

The init_state and next_state represent the model's initial state and an operation for interpreting a command over the model, respectively. Finally there are three operations concerned with the system under test: init_sut for initializing it, run_cmd for interpreting a command, and cleanup for resetting the system under test. We include the full example in Appendix A. As an additional operation, the signature requires precond for expressing preconditions for a command. This is useful, e.g., to prevent the command list shrinker from breaking invariants when minimizing counterexamples.

The framework is phrased as a functor QCSTM.Make. When passed a module satisfying the StmSpec interface it returns a module with the following signature:

sig
  val arb_cmds : state -> cmd list arbitrary
  val interp_agree : state -> sut -> cmd list -> bool
  val agree_test : ?count:int -> name:string -> Test.t
end

The arb_cmds represents a state-dependent command list generator and interp_agree represents an agreement checker for command lists. The operation agree_test lets us easily build an agreement test. Compared to writing a model out explicitly, the framework saves us from repeatedly writing a recursive agreement checker and a state-dependent command list generator.

The example comprises a state machine with only a single state as illustrated in Fig. 2. Additional states and precond come into play when modeling a protocol, e.g., if Queue.pop should only be invoked on a non-empty queue.

5. Other examples

To ensure that the design holds water, we have written tests of 4 modules from Stdlib (including a larger Hashtbl test with 9 commands) along with examples from the property-based testing literature. Collectively these span both tests of OCaml and C code called via the C types library. The examples are summarized below and are all available from https://github.com/jmid/qcstm

<table>
<thead>
<tr>
<th>name</th>
<th>type</th>
<th>#cmds</th>
<th>LOC</th>
<th>ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>counter</td>
<td>int ref</td>
<td>4</td>
<td>41</td>
<td>10.3</td>
</tr>
<tr>
<td>water jug</td>
<td>puzzle</td>
<td>6</td>
<td>43</td>
<td>7.2</td>
</tr>
<tr>
<td>Queue</td>
<td>Stdlib</td>
<td>5</td>
<td>66</td>
<td>13.2</td>
</tr>
<tr>
<td>Stack</td>
<td>Stdlib</td>
<td>7</td>
<td>79</td>
<td>11.3</td>
</tr>
<tr>
<td>Buffer</td>
<td>Stdlib</td>
<td>8</td>
<td>86</td>
<td>10.8</td>
</tr>
<tr>
<td>Hashtbl (minimal)</td>
<td>Stdlib</td>
<td>3</td>
<td>48</td>
<td>16.0</td>
</tr>
<tr>
<td>Hashtbl</td>
<td>Stdlib</td>
<td>9</td>
<td>97</td>
<td>10.8</td>
</tr>
<tr>
<td>put-get</td>
<td>C</td>
<td>2</td>
<td>42</td>
<td>21.0</td>
</tr>
<tr>
<td>circular buffer</td>
<td>C</td>
<td>4</td>
<td>92</td>
<td>23.0</td>
</tr>
<tr>
<td>stdio</td>
<td>C</td>
<td>5</td>
<td>152</td>
<td>30.4</td>
</tr>
</tbody>
</table>

Generally there is some overhead in settings things up, e.g., to define types and apply the functor as illustrated by comparing the two Hashtbl counts. Disregarding the minimal one, OCaml code requires 10–13 lines of test code per command. For testing C code the ratio is clearly higher.

6. Conclusion

We have presented the design of qcstm, a typed state-machine framework for OCaml based on the QCheck QuickCheck library. The framework is available via OPAM: opam install qcstm

References

A. A complete example

open QCheck

module HConf =

struct
  type state = (char * int) list
  type sut = (char, int) Hashtbl.t
  type cmd =
    | Add of char * int
    | Remove of char
    | Find of char
      [@@deriving show]

  (* gen_cmd : state -> cmd Gen.t *)

  let gen_cmd s =
    let char_gen =
      if s = []
      then Gen.char
      else let keys = List.map fst s in
        Gen.oneof [Gen.oneofl keys; Gen.char]
    in
      Gen.oneof
        [ Gen.map2 (fun k v -> Add (k,v)) char_gen Gen.small_nat;
          Gen.map (fun k -> Remove k) char_gen;
          Gen.map (fun k -> Find k) char_gen; ]

  let arb_cmd s = QCheck.make ~print:show_cmd (gen_cmd s)

  let init_state = []

  let next_state c s = match c with
    | Add (k,v) -> (k,v)::s
    | Remove k -> List.remove_assoc k s
    | Find _ -> s

  let init_sut () = Hashtbl.create ~random:false 42

  let cleanup _ = ()

  let run_cmd c s h = match c with
    | Add (k,v) -> begin Hashtbl.add h k v; true end
    | Remove k -> begin Hashtbl.remove h k; true end
    | Find k ->
      List.assoc_opt k s = (try Some (Hashtbl.find h k)
          with Not_found -> None)

  let precond _ _ = true
end

module HT = QCSTM.Make(HConf);

QCheck_runner.run_tests ~verbose:true
  [HT.agree_test ~count:500 ~name:"Hashtbl-model-agreement"]