A Note on State-Machine Frameworks for Property-Based Testing

Jan Midtgaard

Version 3.1, February 28, 2020

1 Introduction

State-machine frameworks have proven their worth after property-based testing have located subtle bugs in everything from the underlying AUTOSAR components of Volvo cars to Google's LevelDB library. These two case studies were carried out with the commercial Erlang QuickCheck state-machine framework from Quviq, and the API design of this framework has subsequently been mimicked in the open source frameworks Proper and Triq.

In this note we reconstruct a typed state-machine framework for OCaml based on the QCheck library. In doing so, we illustrate a number concepts common to all such frameworks: state modeling, commands, interpreting commands, preconditions, and agreement checking. As such, the note is as relevant to understand state-machine frameworks in Erlang, Haskell, Python, or C++.

2 A direct example

As an example, consider the builtin hashtable implementation from the OCaml standard library. We recall a selection of the hashtable API in Fig. 1. The operation `create` lets us create a hashtable, whereas `add` and `remove` lets us insert and delete key-value entries to and from an existing hashtable. We consider two forms of hashtable queries: `find` looks up the value for a given key whereas `mem` simply asks whether there exists an entry with a given key.

```
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
val mem : ('a, 'b) Hashtbl.t -> 'a -> bool
```

Figure 1: Selected operations from the `Hashtbl` API

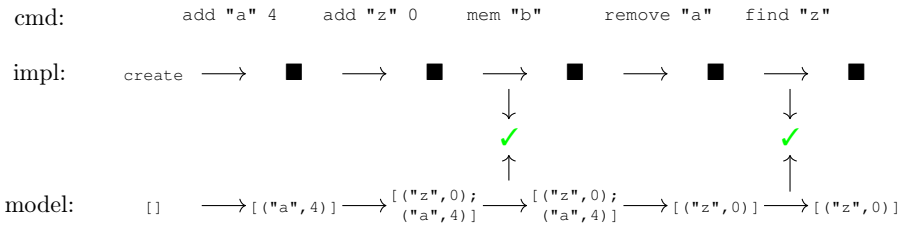

Figure 2: Checking agreement of a list of commands, diagrammatically

If we are to test this imperative API 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 (functional) specification of the imperative API.

Graphically, we can visualize our approach as in Fig. 2, where the top row represents an arbitrary sequence of commands, the middle row represents the operations over the system under test (as a black box), and the bottom row represents the operations over the model. Side-effecting operations (`add` and `remove` with return type `unit`) are interpreted over both rows without a comparison, whereas operations with an actual output (`mem` and `find`) cause us to compare it to the model's output.

2.1 Commands and command generators

The hashtables from OCaml's standard library are polymorphic: they work for multiple key types `'a` and value types `'b`. To test them however, we need to choose concrete key and value types. Somewhat arbitrarily, we choose `string` 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:

```
type cmd =
  | Add of string * int
  | Remove of string
  | Find of string
  | Mem of string [@@deriving show { with_path = false }]
```

Here we utilize a preprocessor (ppx-deriving) to automatically derive a printer (`show_cmd : cmd -> string`) from the type definition.

Based on the data type definition we can now write a straight-forward generator of commands. We phrase the generator in terms of a string generator `str_gen` that chooses between generating a short string and an arbitrary one:

```
(*  gen_cmd : cmd Gen.t  *)
let gen_cmd =
  let str_gen = Gen.oneof [Gen.small_string;
                           Gen.string] in
  Gen.oneof
    [ Gen.map2 (fun k v -> Add (k,v)) str_gen Gen.small_nat;
      Gen.map  (fun k   -> Remove k) str_gen;
      Gen.map  (fun k   -> Find k) str_gen;
      Gen.map  (fun k   -> Mem k) str_gen; ]
```

When combined with the printer `show_cmd` we can now form a full generator of arbitrary commands, and subsequently lift it to a full generator of arbitrary command lists:

```
(*  arb_cmd : cmd QCheck.arbitrary *)
let arb_cmd = make ~print:show_cmd gen_cmd

(*  arb_cmds : cmd list QCheck.arbitrary *)
let arb_cmds = list arb_cmd
```

With commands and command generation in place, we proceed to our model.

2.2 Model and model interpretation

We can model a hashtable with string keys and integer values by an association list of `string * int` pairs:

```
type state = (string * int) list
```

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

```
(*  next_state : cmd -> state -> state *)
let next_state c s = match c with
  | Add (k,v) -> (k,v)::s
  | Remove k  -> List.remove_assoc k s
  | Find _
  | Mem _     -> 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` from the standard library. This model faithfully models how adding an entry with an already existing key shadows (but does not overwrite) any previous entries. In the `Find` and `Mem` cases the state is returned unmodified since these two operations have no effects on the hashtable's internal state.

2.3 Interpreting commands and verifying the output

We still need to interpret the symbolic commands over the actual *system under test*, namely the hashtables from OCaml's standard library and to verify that any output returned is as expected. We perform these two tasks with a function `run_cmd`:

```
(*  run_cmd : cmd -> state -> (string, int) Hashtbl.t -> 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)
  | Mem k     -> List.mem_assoc k s = Hashtbl.mem h k
```

Since `Add` and `Remove` have return type `unit`, there is no output to verify and we therefore simply return **`true`**. In the `Find` and `Mem` cases, we verify that the output of the hashtable operations agree with the corresponding operations over the model's association list. We do so by relying on two functions from the standard library's `List` module:

```
List.assoc_opt : 'a -> ('a * 'b) list -> 'b option
List.mem_assoc : 'a -> ('a * 'b) list -> bool
```

2.4 Combining all the pieces

Based on the above, we can now write a recursive agreement checker that walks the list of symbolic commands:

```
(*  interp_agree : state -> (string, int) Hashtbl.t -> cmd list -> bool *)
let rec interp_agree s h cs = match cs with
  | [] -> true
  | c::cs ->
    let b = run_cmd c s h in
    let s' = next_state c s in
    b && interp_agree s' h cs
```

The function interprets a command first over the hashtable and then over the model. We remark that the recursion stops early in case of a disagreement, thanks to short-circuit Boolean evaluation of the conjunction `&&`.

Finally we can formulate the actual agreement test:

```
(*  agree_test : QCheck.Test.t  *)
let agree_test =
  Test.make ~name:"Hashtbl-model␣agreement" ~count:500
    arb_cmds
    (fun cs -> interp_agree [] (Hashtbl.create ~random:false 42) cs)
```

The property tests agreement between the model and the hashtable, starting from an empty association list and a newly created hashtable. With optional

parameters we furthermore specify a title for the test and a number of runs (500). We can now run the test:

```
QCheck_runner.run_tests ~verbose:true [agree_test]
```

which gives rise to the following output:

```
random seed: 106718569
generated error fail pass / total     time test name
[✓]  500    0    0  500 /  500     1.9s Hashtbl-model agreement
============================================================
success (ran 1 tests)
```

2.5 Interlude: Fault injection

When run, the property-based test suite finishes without errors after ensuring agreement over 500 arbitrary command sequences. How can we be sure that the test suite works as expected?

One way to "test the tester" is by *fault injection*, i.e., introducing a deliberate error to check whether it is caught by our test suite. Since there are several pieces of software in play this could be either (1) an error in the model or (2) an error in the hashtable implementation.

Suppose we change `run_cmd` as follows:

```
(*  run_cmd : cmd -> state -> (string, int) Hashtbl.t -> bool *)
let run_cmd c s h = match c with
  | Add (k,v) ->
    if String.length k <= 2
    then Hashtbl.add h k v
    else Hashtbl.add h k (v+1);
    true
  (* remaining cases left unmodified *)
```

This corresponds to injecting an error in the hashtable implementation, causing it to insert the value `v+1` rather than `v` when the key's string length is greater than 2.

Somewhat surprisingly, after recompiling and rerunning our test suite, it does not appear to catch the injected error:

```
random seed: 326199985
generated error fail pass / total     time test name
[✓]  500    0    0  500 /  500     1.9s Hashtbl-model agreement
============================================================
success (ran 1 tests)
```

Why is this so? We realize that our injected error would only be caught by the model-based tests, if we generate a list of commands that first `add`s a key, e.g., `"abc"` with some value, e.g., `41`, and later try to look up the very same key with

`find "abc"`.[1] Our first version simply leaves it up to the generator to produce such a command list, which unfortunately has a relatively low probability.

2.6 Revision: State-dependent key generation

Ideally, we would like to generate `add`, `remove`, `find`, and `mem` commands of existing keys with some reasonable probability, similar to how one would typically hand write unit tests for such cases. One way to achieve this goal is to take the model's state into consideration, in effect implementing *state-dependent command generation*.

Suppose we provide our generator of single commands with the model's state as a parameter, then we can choose between either (1) generating an existing key from the hashtable or (2) generating an arbitrary key with the string generator. The following implementation does so, by collecting the keys of the model and choosing among them with probability $\frac{1}{3}$:

```
(*  gen_cmd : state -> cmd Gen.t  *)
let gen_cmd s =
  let str_gen =
    if s = []
    then Gen.oneof [Gen.small_string;
                    Gen.string]
    else
      let keys = List.map fst s in
      Gen.oneof [Gen.oneofl keys;
                 Gen.small_string;
                 Gen.string] in
  Gen.oneof
    [ Gen.map2 (fun k v -> Add (k,v)) str_gen Gen.small_nat;
      Gen.map  (fun k   -> Remove k) str_gen;
      Gen.map  (fun k   -> Find k) str_gen;
      Gen.map  (fun k   -> Mem k) str_gen; ]
```

How can we now lift `gen_cmd` to generate command lists in a state-dependent manner? To do so, we will utilize the infix operator `>>=` from QCheck's `Gen` module:

```
val ( >>= ) : 'a Gen.t -> ('a -> 'b Gen.t) -> 'b Gen.t
```

This infix operation takes two arguments: a (pure) generator and a function, which is applied to the generated value, similar to a callback function in JavaScript. For example, we can write a generator of integer pairs, where the first component is less than the second:

```
let my_pair_gen =
  let open Gen in
  small_nat >>= fun i -> pair (int_bound i) (return i)
```

[1] `mem "abc"` or `remove "abc"` would not cause disagreement with the model as these operations do not inspect the corresponding value.

An inspection of `my_pair_gen`'s output indicates that it works as intended:

```
# Gen.generate ~n:4 my_pair_gen;;
- : (int * int) list = [(2, 3); (48, 518); (53, 87); (0, 4)]
```

We can utilize our newly discovered operation to express a fueled, state-dependent command list generator:

```
(*  gen_cmds : state -> int -> cmd list QCheck.Gen.t *)
let rec gen_cmds s fuel =
  let open Gen in
  if fuel = 0
  then Gen.return []
  else
    gen_cmd s >>= fun c ->
      (gen_cmds (next_state c s) (fuel-1)) >>= fun cs ->
         return (c::cs)
```

When we run out of fuel, we simply generate the empty command list. When there is still fuel in the tank, we first generate a single command using `gen_cmd` and bind it to the parameter `c`. We subsequently generate the tail of the command list by a recursive call from the state that `c` takes us to and bind the generated tail to the parameter `cs`. Finally we glue `c` in front of `cs` and return the result as a constant.

It is now straightforward to construct a full generator, by providing a printer and a (list) shrinker and initializing the (pure) generator with the empty association list:

```
(*  arb_cmds : cmd list QCheck.arbitrary *)
let arb_cmds =
  make ~print:(Print.list show_cmd) ~shrink:Shrink.list
    (Gen.sized (gen_cmds []))
```

We can now check that our injected error is caught as expected:

```
random seed: 136017840
generated error fail pass / total     time test name
[✗]     6    0    1    5 /  500     0.0s Hashtbl-model agreement

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

Test Hashtbl-model agreement 2 failed (10 shrink steps):

[(Add ("^\228\203P", 5)); (Find "^\228\203P")]
================================================================
```

The reported counterexample is minimal in that it contains only two commands: an `add` command followed by a `find` command. The string key is small (4 characters) but not minimal, as the minimal key length to trigger our injected error is 3 characters. Note that QCheck has not shrunk the individual list elements (the commands) since we have not provided a shrinker for them. As such it has only shrunk the list size.

3 Extrapolating from the direct example

In summary, we have combined three types:

- a type of commands
- a system under test (hashtables)
- a model of the system's state (association lists)

with operations for

- interpreting commands over the model
- interpreting commands over the system under test and assuring agreement
- ensuring agreement over a list of commands

In addition we have realized the benefit of state-dependent command generation.

Based on this realization, we consider a common module signature (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
end
```

The three types `cmd`, `state`, `sut` are as expected: They represent the type of commands, the type of the model's state, and the type of the system under test, respectively. The operation `arb_cmd` is 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.

The `init_state` and `next_state` represents the model in the form of its initial state and an operation for interpreting a command over the model, respectively. Finally there are three operations concerned with the system under test (abbreviated `sut`): `init_sut` for initializing it, `run_cmd` for interpreting a command, and `cleanup` for resetting the system under test.

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 as we shall see shortly.

The state-machine framework is formulated as a functor `QCSTM.Make`. When `QCSTM.Make` is passed a module satisfying the above signature in return it produces a module with the following signature:

```
sig
  (* some entries omitted *)
  val arb_cmds : Spec.state -> Spec.cmd list arbitrary
  val interp_agree : Spec.state -> Spec.sut -> Spec.cmd list -> bool
  val agree_test : ?count:int -> name:string -> Test.t
end
```

Note how `arb_cmds` represents a state-dependent command list generator and how `interp_agree` represents a recursive agreement checker. The operation `agree_test` lets us easily build an agreement test.

For example, we can instantiate the `QCSTM` framework to test OCaml's hashtables by creating a module `HConf` with the required signature and passing it to `QCSTM.Make` as illustrated in Fig. 3. After binding the resulting module to the name `HT`, we can run the QCheck test `HT.agree_test` as desired. Compared to writing a model out explicitly, a framework saves us from repeatedly writing a recursive agreement checker and a dependent command list generator.

4 Another example: Queues

As a second example, consider the selection of the standard library's `Queue` API in Fig. 4. The `create` operation lets us construct a new queue, `push` adds an element to the back, `pop` removes the front element, whereas `top` reveals the front element without affecting the underlying queue. Both the `pop` and `top` operations raise an exception when called with an empty queue:

```
# Queue.top (Queue.create ());;
Exception: Stdlib.Queue.Empty.
```

As such it is a *precondition* for `pop` and `top` that their `Queue` argument is non-empty. This means we have to be careful if we wish to generate and test only well-formed sequences of `Queue` operations.

4.1 Modeling queues

We can easily model a queue as a list: the top of the queue is the list's left end, the empty list represents the empty queue, and we push an element by appending it on the right. If we fix the type of queue elements to `int`, the three types for `QCSTM` are in place:

```
type state = int list
type sut = int Queue.t
```

```
open QCheck

module HConf =
struct
  type state = (string * int) list
  type sut   = (string, int) Hashtbl.t
  type cmd =
    | Add of string * int
    | Remove of string
    | Find of string
    | Mem of string [@@deriving show { with_path = false }]

  (*  gen_cmd : state -> cmd Gen.t  *)
  let gen_cmd s =
    let str_gen =
      if s = []
      then Gen.oneof [Gen.small_string;
                      Gen.string]
      else
        let keys = List.map fst s in
        Gen.oneof [Gen.oneofl keys;
                   Gen.small_string;
                   Gen.string] in
    Gen.oneof
      [ Gen.map2 (fun k v -> Add (k,v)) str_gen Gen.small_nat;
        Gen.map  (fun k   -> Remove k) str_gen;
        Gen.map  (fun k   -> Find k) str_gen;
        Gen.map  (fun k   -> Mem k) str_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 _
    | Mem _     -> s

  let init_sut () = Hashtbl.create ~random:false 42
  let cleanup _   = ()
  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)
    | Mem k     -> List.mem_assoc k s = Hashtbl.mem h k

  let precond _ _ = true
end
module HT = QCSTM.Make(HConf)
;;
QCheck_runner.run_tests ~verbose:true
  [HT.agree_test ~count:500 ~name:"Hashtbl-model␣agreement"]
```

Figure 3: A complete example of `QCSTM.Make`

```
val create : unit -> 'a Queue.t
val pop    : 'a Queue.t -> 'a
val top    : 'a Queue.t -> 'a
val push   : 'a -> 'a Queue.t -> unit
```

Figure 4: A selection of the `Queue` API operations

```
type cmd =
  | Pop
  | Top
  | Push of int [@@deriving show { with_path = false }]
```

Based on these types we can now implement the model:

```
let init_state = []
let next_state c s = match c with
  | Pop ->
    (match s with
      | []    -> []
      | _::s' -> s')
  | Top -> s
  | Push i -> s@[i]
```

Initially queues are empty, modeled with the empty list `[]`. The command interpreter `next_state` pattern matches on the given command. A `push` command alters the state by inserting a new element last, at the right. Since `top` will not alter the internal state, it just returns the state `s` unmodified. Finally `pop` needs a non-empty queue state to remove from, so in case the state is empty `[]` the model remains unmodified.

4.2 Interpreting and verifying queue commands

We can also hook up with the system under test, the `Queue` module:

```
let init_sut () = Queue.create ()
let cleanup _   = ()
let run_cmd c s q = match c with
  | Pop -> (try Queue.pop q = List.hd s with _ -> false)
  | Top -> (try Queue.top q = List.hd s with _ -> false)
  | Push n -> Queue.push n q; true
```

The initializer `init_sut` simply creates a new queue. As old queues will be garbage collected `cleanup` does not need to do anything. The command interpreter `run_cmd` again pattern matches on the given command. Since `push` commands have no return value, we simply perform the corresponding queue command and return **`true`** to signal success. Both `pop` and `top` perform the corresponding queue command and compare the result with the front of the list. Should either of `pop`, `top`, or `List.hd` raise an exception, we return **`false`** to signal a test failure.

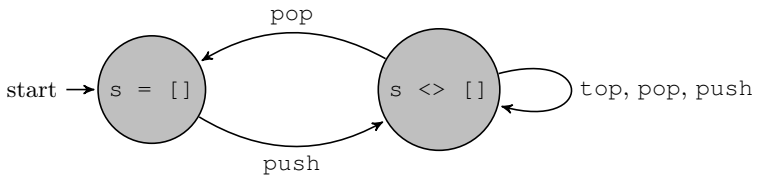

Figure 5: A state machine corresponding to the `Queue` command generator

4.3 Generating queue commands

We can now phrase a state-dependent command generator:

```
let gen_cmd s =
  let int_gen = Gen.small_nat in
  if s = []
  then Gen.map (fun i -> Push i) int_gen
  else Gen.oneof
         [Gen.return Pop;
          Gen.return Top;
          Gen.map (fun i -> Push i) int_gen]
```

From an empty queue (`s = []`) we can only generate `push` commands, whereas from a non-empty queue (`s <> []`) we can generate either `pop`, `top`, or `push` commands. This generator corresponds to the state machine in Fig. 5: In one configuration (`s = []`) only `push` commands are allowed, whereas a second configuration (`s <> []`) allows all three kinds of commands. For each command the state machine performs a transition from configuration to configuration. For example, upon encountering a `push` command in the configuration `s = []` the state machine moves to the second configuration `s <> []`. On the other hand, upon encountering a `push` command in the configuration `s <> []` the state machine stays in the configuration `s <> []`.

We can now lift the pure generator to a full, state-dependent generator by supplying the derived printer:

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

4.4 Preconditions and their effect on shrinking

It only remains to provide a precondition function `precond` to instantiate the `QCSTM.Make` functor. For now we simply provide a constant **`true`** function:

```
let precond _ _ = true
```

If we instantiate the `QCSTM.Make` functor with all the above and run the agreement test `agree_test` of the resulting module, everything seems to work:

```
random seed: 79522449
generated error  fail  pass / total     time test name
[✓] 10000     0     0 10000 / 10000     1.6s queue-model agreement
================================================================
success (ran 1 tests)
```

Now, let us inject an error, e.g., change the model to ignore insertions of 98:

```
let next_state c s = match c with
  | Pop ->
    (match s with
      | []    -> []
      | _::s' -> s')
  | Top -> s
  | Push i ->
    if i<>98 then s@[i] else s  (* an artificial fault in the model *)
```

Rerunning our tests, we encounter a problem:

```
random seed: 226748961
generated error  fail  pass / total     time test name
[✗]     9     0     1     8 / 10000     0.0s queue-model agreement

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

Test queue-model agreement failed (14 shrink steps):

[Pop]
================================================================
```

QCheck generated 9 well-formed command lists, found a mismatch between the implementation and the model on the ninth command list, and subsequently shrunk it down to just one command: `pop`! Such a command list clearly cannot result from our generator, but it arose from shrinking a well-formed command list. The `precond` function guards against this very situation, by expressing which commands are considered acceptable in a given state:

```
let precond c s = match c with
  | Pop    -> s<>[]
  | Top    -> s<>[]
  | Push _ -> true
```

This states that `pop` and `top` are acceptable only when the queue is non-empty. As such, it expresses the requirements from our state machine in Fig. 5 again. With our refined `precond` function, we confirm that counterexamples are now shrunk to well-formed command lists:

```
random seed: 508563729
generated error  fail  pass / total     time test name
[✗]     6     0     1     5 / 10000     0.0s queue-model agreement

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

Test queue-model agreement failed (10 shrink steps):

[(Push 98); (Push 8); Pop]
================================================================
```

This is a minimal counterexample: we need two `push` commands and the first to insert 98, in order to observe how the insertion has been ignored.

4.5 A consistency test

Since we now express command list requirements in two different ways: as a dependent generator and as a precondition function, there is a natural risk of a mismatch between the two. For this reason `QCSTM` offers a consistency test:

```
val consistency_test : ?count:int -> name:string -> Test.t
```

This test generates a number of command lists and checks that all contained commands satisfy the precondition `precond`. It accepts an optional `count` parameter and a test name as a labeled parameter `name`.

We include the complete `Queue` example in Fig. 6.

5 Conclusion

We have written a model-based test of the hashtables from OCaml's standard library and generalized the example to a general approach for testing imperative code. Secondly with a queue example we have illustrated how commands with a precondition can be expressed. We thereby avoid generating, testing, and shrinking to ill-formed command sequences.

The QCSTM framework can be installed via the OPAM package manager with the following command: `opam install qcstm`
The source code and additional examples are available from the following url:

`https://github.com/jmid/qcstm`

```
open QCheck

module QConf =
struct
  type state = int list
  type sut = int Queue.t
  type cmd =
    | Pop (* may throw exception *)
    | Top (* may throw exception *)
    | Push of int [@@deriving show { with_path = false }]

  let gen_cmd s =
    let int_gen = Gen.small_nat in
    if s = []
    then Gen.map (fun i -> Push i) int_gen
    else Gen.oneof
           [Gen.return Pop;
            Gen.return Top;
            Gen.map (fun i -> Push i) int_gen]

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

  let init_state = []
  let next_state c s = match c with
    | Pop ->
      (match s with
        | []    -> []
        | _::s' -> s')
    | Top -> s
    | Push i -> s@[i]

  let init_sut () = Queue.create ()
  let cleanup _   = ()
  let run_cmd c s q = match c with
    | Pop -> (try Queue.pop q = List.hd s with _ -> false)
    | Top -> (try Queue.top q = List.hd s with _ -> false)
    | Push n -> begin Queue.push n q; true end

  let precond c s = match c with
    | Pop    -> s<>[]
    | Top    -> s<>[]
    | Push _ -> true
end

module QT = QCSTM.Make(QConf)
;;
QCheck_runner.run_tests ~verbose:true
  [QT.consistency_test ~count:10_000 ~name:"queue␣gen-precond␣agreement";
   QT.agree_test       ~count:10_000 ~name:"queue-model␣agreement"]
```

Figure 6: The complete `Queue` example of `QCSTM`