# A Simplified Model Of QuickCheck Automata*

Kevin Falzon
University of Malta
kfal0002@um.edu.mt

Gordon J. Pace
University of Malta
gordon.pace@um.edu.mt

## ABSTRACT

Placing guarantees on a program's correctness is as hard as it is essential. Several approaches to verification exist, with *testing* being a popular, if imperfect, solution. The following is a formal description of *QuickCheck finite state automata*, which can be used to model a system and automatically derive command sequences over which properties can be checked. Understanding and describing such a model will aid in integrating this verification approach with other methodologies, notably *runtime verification*.

## 1. INTRODUCTION

Verifying a program's correctness is often non-trivial, and sometimes outright impossible. Several approaches towards verification have been adopted over time, including *testing* [1] and *runtime verification* [3]. The aforementioned techniques are concerned with the verification of a partial or full program with respect to a given *property*, where a property characterises a set of expected behaviours in some manner, as will be discussed shortly.

Testing always involves the *observation* and *classification* of a sample of executions [2]. One would typically provide a series of test inputs, possibly generated automatically, which are then executed and evaluated. Any discrepancies between the result of an execution and that which was expected indicate that there is a fault. The expected output can be characterised by a property expressed in several ways, such as through a list of input-output pairs or an automaton describing the system's behaviour. A common shortcoming of testing is that the test sample used is rarely exhaustive, leaving certain code paths untested and leading to the persistence of latent errors.

*QuickCheck* [5] is a automatic test case generation and execution tool developed by QuviQ, which facilitates the testing of *Haskell* or *Erlang* programs. In broad terms, QuickCheck ensures that a *property* (a code block that returns true on success) holds over a set of test cases. Test values are initialised using *generator* functions, which allow a user to specify the structure of a property's input data. The randomness is derived from the distribution of values produced by the generators. QuickCheck also allows different *quantifiers* to be used. For example, one could easily check a property $P(n)$ such that $\forall n : \mathbb{N} \cdot P(n)$. Naturally, only a subset of the possible values of $n$ will be checked, meaning that the property will not be tested exhaustively over all inputs, yet confidence in the system will increase as the number of analysed test cases grows.

QuickCheck also allows one to model a system as a *finite-state automaton*, from which different program execution paths can be derived and tested automatically. This paper will focus on the definition and use of such automata, starting with a an informal description of their behaviour illustrated by a simple example, followed by a formal definition of a simplified version of the automaton.

## 2. QUICKCHECK AUTOMATA

QuickCheck supports the creation of a class of finite-state automata [5], henceforth abbreviated as *QCFSA*. A QCFSA describes some aspect of the system by modelling a set of possible function call sequences, or *program traversals*. Each QCFSA transition corresponds to a function call. QuickCheck traverses the automaton, choosing arcs at random, and generates a sequence of calls (a *trace*). The trace can then be executed, and its result may be checked within a QuickCheck property.

Every arc has an associated *precondition* and *postcondition* function, which returns a boolean value. The precondition function is used to limit the generation of certain traces, with a path not being chosen whenever its precondition fails. A state may have multiple outgoing arcs for the same function call, provided that not more than one precondition returns true at that point in the traversal for that call. The postcondition function is checked while a trace is being run, following the execution of the transition's associated function. If the postcondition returns anything other than *true*, then the trace is considered to be a failing test.

A QCFSA may also store *state data*, which can be updated during a transition by associating a *next state data* function. This function is invoked after a postcondition evaluates to true, and returns the next state's data based on the current state data and the result returned by the function once it terminates. Additional auxiliary functionality, namely the ability to weight transitions, will not be considered in this exposition.

### 2.1 An Example

Consider a simple controller operating a light, which can either be **on** or **off**. The light may be manipulated by calling one of two functions, `switch_on()` and `switch_off()`, which return the light's new state.

Assuming that the light is initially off, this system could be modelled using a two-state automaton, as illustrated in Figure 1. This automaton could be written as a QuickCheck automaton as follows:

*The research work disclosed in this publication is partially funded by the Strategic Educational Pathways Scholarship (Malta). The scholarship is part-financed by the European Union - European Social Fund (ESF) under Operational Programme II - Cohesion Policy 2007-2013, "Empowering People for More Jobs and a Better Quality of Life"

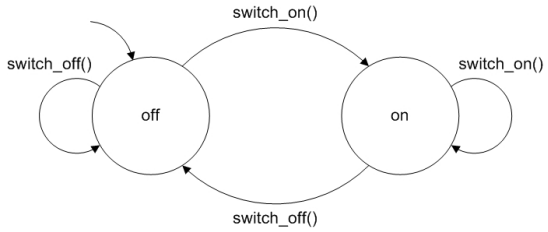


**Figure 1: Light Switch FSA**

- The *initial state* is *off*, and is specified as a return value

```
initial_state() ->
  off.
```

- For clarity, the *initial state data* is defined as a tuple, relating the current light's status to a token. The state data could be any valid Erlang data structure.

```
initial_state_data() ->
  {light, off}.
```

- Transitions are defined as functions of the form

$$S(D) \rightarrow [\{S', \{call, Module, Function, Args\}\}].$$

where $S$ and $S'$ are the current and next state, respectively. On performing a transition, the system will invoke `Function(Args)` within the specified `Module`. The transition function returns a list, as one may define multiple outgoing transitions from a given state. $D$ is the current state data, and references to $D$ may only be made within the $Args$ parameter. In the example under consideration, the transitions are defined as

```
off(_D) ->
  [{on,  {call, ?LIGHT_MODULE, switch_on,  []}},
   {off, {call, ?LIGHT_MODULE, switch_off, []}}].

on(_D) ->
  [{on,  {call, ?LIGHT_MODULE, switch_on,  []}},
   {off, {call, ?LIGHT_MODULE, switch_off, []}}].
```

where `?LIGHT_MODULE` is the name of the module implementing the switching functions.

- A precondition from state $S$ to $S'$ performing a $Call$ with state data $D$ has the general signature

$$precondition(S, S', D, Call) \rightarrow \mathbb{B}$$

As all paths must be generated, including self-looping state transitions, the precondition is set to return true for all possible transitions. Thus,

```
precondition(_, _, _, _) ->
  true.
```

- A postcondition from state $S$ to $S'$ performing a $Call$ with state data $D$ has the general signature

$$postcondition(S, S', D, Call, Result) \rightarrow \mathbb{B}$$

Postconditions can be expressed in several ways. One can opt to specify the correct behaviours and default to false for undefined behaviours. Conversely, one may define exceptions for incorrect behaviours, as in the following

```
postcondition(on, off, {light, on}, _, R) ->
  R == off;
postcondition(off, on, {light, off}, _, R) ->
  R == on;
postcondition(_, _, _, _, _) ->
  true.
```

The postcondition ensures that the result returned matches the value expected for that light-switch transition. The preconditions as defined above fail to consider the result of executing a light-changing function whilst remaining in the same state. Also, as only the function's return value is relevant, the state data clause may be omitted. The precondition should thus be generalised to consider any incoming transition, as follows

```
postcondition(_, off, {light, _}, _, R) ->
  R == off;
postcondition(_, on, {light, _}, _, R) ->
  R == on;
postcondition(_, _, _, _, _) ->
  true.
```

- Similarly, the next state data function has a general signature of

$$next\_state\_data(S, S', D, Result, Call) \rightarrow D'$$

In this case, on a successful transition, the next state should simply be updated with the light's new status. Thus,

```
next_state_data(_, _, _, R, _) ->
  {light, R}.
```

## 2.2 Running Tests

The light switch model was tested extensively using the automaton as defined previously. QuickCheck did not succeed in finding an error-inducing counterexample, which suggests (but does not prove) that the program correctly mirrors the model. So as to induce an error, an invalid implementation of the `switch_on()` function was tested. Instead of returning an **on** state, the function was set to return an **error**. QuickCheck succeeded in finding a failing trace, represented symbolically as

```
[{set,{var,1},{call,light,switch_on,[]}},
 {set,{var,2},{call,light,switch_off,[]}}]
```

This was then minimised to a single call, namely

```
[{set,{var,1},{call,light,switch_on,[]}}]
```

# 3. FORMALISING QCFSA

The following section presents a formal description of a simplified model of QCFSAs. The model is simplified in that it assumes that all operations are free from side-effects. It also assumes that functions are invoked without any parameters, which in practical terms would require that arguments are passed through a global state data structure.

Throughout the text, the term "event" refers to a symbolic call corresponding to an implemented function in the system under test. The automata being considered are assumed to be *deterministic*, using the definition provided in Section 3.1.1.

## 3.1 The Model

If $\Theta$ represents a *system state*, a QCFSA $\mathcal{Q}$ is described by a tuple $\langle Q, q_0, \Sigma, \theta_0, run, \rightarrow \rangle$, where

$Q$ is a set of states

$q_0 \in Q$ is the initial state

$\Sigma$ is an alphabet of events representing functions in the system under test

$\theta_0$ is the initial global system state containing all of the program's state variables, including $\mathcal{Q}$'s local state data

$run \in \Sigma \rightarrow \Theta \rightarrow \Theta$ is a function which returns the updated system state on executing a given event's associated function with the specified system state data

$\rightarrow \subseteq Q \times (2^{\Theta} \times \Sigma \times 2^{\Theta} \times \Theta \rightarrow \Theta) \times Q$ is a transition relation

The transition relation $\rightarrow$ serves to associate the preconditions, postconditions and state-changing actions to be performed on calling an event. Pre- and postconditions are characterised by sets of valid system states. Thus, a precondition is satisfied if the current system state is a member of the defined set. Similarly, a postcondition is satisfied if the system state entered after executing an event's associated function is a member of the set of valid postcondition states. As mentioned previously, a transition may specify a state-changing action, which is primarily used for updating the automaton's private state data. This action is performed whenever a postcondition executes successfully.

Moving from state $q$ to $q'$ on event $a$ with system state sets $pre$ and $post$ and executing state-changing action $\alpha$ is denoted by

$$q \xrightarrow[\alpha]{\{\text{pre}\}\ a\ \{\text{post}\}} q' \stackrel{\text{def}}{=} (q, (pre, a, post, \alpha), q')$$

which is simply a more elegant way of representing a transition.

### 3.1.1 Determinism

The model describes automata that are deterministic. Given any pair of transitions emanating from a state, either

- the transitions share the same state changing action, pre- and postconditions and lead to the same state, implying that the transitions are identical, or
- the transitions have non-intersecting preconditions, meaning that for each event, only at most one transition can be pursued

Formally,

$$\begin{array}{l}\forall a, q', q_1, q_2, pre_1, pre_2, post_1, post_2, \alpha_1, \alpha_2 \cdot \\ \quad ((q', pre_1, a, post_1, \alpha_1, q_1) \in \rightarrow \wedge \\ \quad (q', pre_2, a, post_2, \alpha_2, q_2) \in \rightarrow) \Rightarrow \\ \quad\quad ((pre_1 = pre_2 \wedge post_1 = post_2 \wedge \alpha_1 = \alpha_2 \wedge q_1 = q_2) \vee \\ \quad\quad (pre_1 \bigcap pre_2 = \emptyset))\end{array}$$

**Example 3.1.1**
The light-switch model described in Section 2.1 can be formalised as

$$\begin{array}{l}\langle\ Q, off, E, \{(light, off)\}, run, \\ \quad \{(S, (pre, e, \{post\}, \lambda t \cdot post), S') \mid \\ \quad\quad pre \in 2^{\Theta}, \theta \in 2^{\Theta}, S \in Q, S' \in Q, e \in E, \\ \quad\quad post = run(e, \theta) \wedge \{(light, S')\} = post\}\rangle\end{array}$$

where
$Q = \{off, on\}$,
$E = \{switch\_on, switch\_off\}$,
$run = \lambda e, t \cdot$
$\quad \mid (e = switch\_on) \rightarrow \{(light, on)\}$
$\quad \mid (e = switch\_off) \rightarrow \{(light, off)\}$

## 3.2 Configurations

### 3.2.1 Single Events

The *configuration* of a QCFSA consists of a pair $(q, \theta)$, where $q$ is the current state and $\theta$ is the global system state at that point in the execution. Performing a step from a configuration $(q, \theta)$ to a *valid* configuration $(q', \theta')$ on executing an event $a$ is denoted by $(q, \theta) \stackrel{a}{\Rightarrow} (q', \theta')$, and is possible if

$$\begin{array}{l}\exists pre, post, \alpha, \theta_m \cdot \\ \quad q \xrightarrow[\alpha]{\{\text{pre}\}\ a\ \{\text{post}\}} q' \quad \wedge \\ \quad \theta \quad \in pre \quad\quad \wedge \\ \quad \theta_m = run(a, \theta) \quad \wedge \\ \quad \theta_m \in post \quad\quad \wedge \\ \quad \theta' \quad = \alpha(\theta_m)\end{array}$$

A transition leads to a bad configuration if its postcondition is violated once an event is executed. As described earlier, a postcondition is violated if the new state is not part of the valid system state set. Moving to a bad configuration under event $a$ is thus defined as

$$\begin{array}{l}(q, \theta) \stackrel{a}{\Rightarrow} \otimes \stackrel{\text{def}}{=} \\ \quad \exists pre, post, \alpha, q' \cdot \\ \quad\quad q \xrightarrow[\alpha]{\{\text{pre}\}\ a\ \{\text{post}\}} q' \wedge \\ \quad\quad \theta \in pre \quad\quad \wedge \\ \quad\quad run(\alpha, \theta) \notin post\end{array}$$

### 3.2.2 Event Sequences

Trivially, a configuration does not change if no event is executed. Thus, if $\epsilon$ is the null event,

$$(q, \theta) \stackrel{\epsilon}{\Rightarrow} (q', \theta') \stackrel{\text{def}}{=} q = q' \wedge \theta = \theta'$$

Given a sequence of events $a : s$, where $s$ can itself be another sequence, moving from one configuration to another valid configuration whilst executing the sequence can be defined recursively as

$$\begin{array}{l}(q, \theta) \stackrel{a:s}{\Rightarrow} (q', \theta') \stackrel{\text{def}}{=} \\ \quad \exists q'', \theta'' \cdot (q, \theta) \stackrel{a}{\Rightarrow} (q'', \theta'') \wedge (q'', \theta'') \stackrel{s}{\Rightarrow} (q', \theta')\end{array}$$

This result assumes that any two consecutive steps can be combined into a single step over a sequence. Conversely, a sequence can be decomposed at any point into two sub-sequences. This can be described formally using the equality $\stackrel{a_1}{\Rightarrow}; \stackrel{a_2}{\Rightarrow} = \stackrel{a_1 a_2}{\Rightarrow}$.

A system cannot move to a bad configuration on an empty event, that is,

$$(q, \theta) \stackrel{\epsilon}{\not\Rightarrow} \otimes$$

Moving to a bad configuration given a sequence of events is defined as

$$(q, \theta) \stackrel{s+\not\Rightarrow<a>}{\Longrightarrow} \otimes \stackrel{\text{def}}{=} \exists q', \theta' \cdot (q, \theta) \stackrel{s}{\Rightarrow} (q', \theta') \stackrel{a}{\not\Rightarrow} \otimes$$

with the last step leading to a bad configuration.

**Example 3.2.1**
With reference to the light-switch model described in Section 2.1, the following expressions hold

$$(off, \{light, off\}) \stackrel{switch\_on}{\Longrightarrow} (on, \{light, on\})$$

If `switch_on()` is implemented incorrectly and does not return **on**,

$$(off, \{light, off\}) \stackrel{switch\_on}{\Longrightarrow} \otimes$$

## 3.3 Describing Traces

Using the aforementioned definitions, one may characterise two important sets related to the traversal of an automaton $\mathcal{Q}$, these being the set of *negative traces* $N(\mathcal{Q})$ and the set of *testable traces* $T(\mathcal{Q})$.

$N(\mathcal{Q})$ is the set of traces which, starting from the automaton's initial configuration, lead to a bad configuration. Thus,

$$N(\mathcal{Q}) \stackrel{\text{def}}{=} \{w : \Sigma^* \mid (q_0, \theta_0) \stackrel{w}{\not\Rightarrow} \otimes\}$$

$T(\mathcal{Q})$ refers to the set of traces with which the automaton is concerned, that is, the traces which it tests. Thus,

$$T(\mathcal{Q}) \stackrel{\text{def}}{=} \{w : \Sigma^* \mid \exists q, \theta \cdot (q_0, \theta_0) \stackrel{w}{\Rightarrow} (q, \theta)\} \bigcup N(\mathcal{Q})$$

The task of checking whether or not a trace $w$ is valid with respect to a property described by an automaton $\mathcal{Q}$ can thus be reformulated into a membership problem. Trace $w$ would be valid as long as $w \notin N(Q)$. In general, a program could be considered as being valid with respect to $\mathcal{Q}$ if it can never produce a trace in $N(Q)$ when starting from the initial conditions set by $\mathcal{Q}$.

# 4. QUICKCHECK AND RUNTIME VERIFICATION

By introducing the concept of the set of negative traces for an automaton $\mathcal{Q}$, parallels can be drawn between QuickCheck automata and *runtime verification* monitors. In simple terms, runtime verification is concerned with checking whether a given trace (partial in the case of online monitoring) is a member of the set of valid program behaviours, defined by one or more monitors [4]. Often, a monitor describes the invalid program behaviours, and thus one checks for non-membership. Thus, to translate a QCFSA $\mathcal{Q}$ into a runtime monitor, one would have to define a monitor which recognises $N(\mathcal{Q})$.

Providing the ability to translate between QCFSA and runtime monitors (and possibly back) would aid verification by allowing testing to continue beyond the system's development phase. It would also allow one to test critical parts of a system exhaustively whilst delaying the verification of non-essential components until runtime. Alternatively, one may switch between testing and verifying a component at runtime based on the computing resources available, increasing confidence in a system by testing it during lulls in processor demand. The translation could also incorporate details or statistics obtained during testing in order to guide runtime verification and minimise the number of unnecessary or repeated checks.

Based on preliminary efforts in translating between QCFSA and runtime verification monitors, it appears that most of the issues encountered stem from the fact that QuickCheck operates as a generator, whilst runtime verification listens on a stream of events that is generated by an external entity. This may lead to problems related to non-determinism and the ordering of operations related to a transition.

# 5. CONCLUSIONS AND FUTURE WORK

This paper has presented an overview of QuickCheck automata, illustrating their operation through an example and providing a formal description of a restricted automaton. It also defined the notion of an automaton's set of *testable* and *negative* traces.

The model detailed in this text is restricted primarily in that it is deterministic and assumes that functions have no side-effects. A global system state serves in preserving the model's computational capabilities whilst mitigating the problems that arise from a lack of function arguments. Some of the model's restrictions may have to be lifted in order to facilitate the integration of this approach with other testing techniques.

# 6. REFERENCES


[1] Paul Ammann and Jeff Offutt. *Introduction to software testing*. Cambridge University Press, Cambridge, UK, 2008.

[2] Antonia Bertolino. Software testing research: Achievements, challenges, dreams. In *Future of Software Engineering, 2007. FOSE '07*, pages 85–103, Washington, DC, USA, 2007. IEEE Computer Society.

[3] Séverine Colin and Leonardo Mariani. *Model-Based Testing of Reactive Systems*, chapter 18 Run-Time Verification, pages 525–555. Springer, 2005.

[4] Christian Colombo. Practical runtime monitoring with impact guarantees of java programs with real-time constraints. Master's thesis, University Of Malta, 2008.

[5] Quviq AB. *QuickCheck Documentation Version 1.191*, March 2010.