Combining Testing and Runtime Verification Techniques*

Kevin Falzon[1] and Gordon J. Pace[2]

[1] European Center for Security and Privacy by Design
kevin.falzon@ec-spride.de
[2] Department of Computer Science, University of Malta
gordon.pace@um.edu.mt

Abstract. Testing is an established and integral part of the system design and development process, but incomplete coverage still leaves room for potential undiscovered bugs. Runtime verification addresses this issue by integrating verification oracles into the code, allowing for reparatory action to be taken in case of system failure after deployment. Despite the complementarity of the two approaches, the application of the two approaches at different stages in the development and deployment process results in much duplication of effort. In this paper we investigate the combination of the two approaches, by showing how one can use testing oracles to derive correct runtime verification monitors. We show how this can be achieved using QuickCheck and Larva, and apply the resulting framework to Riak, a fault-tolerant distributed database written in Erlang.

1 Introduction

As the need for more dependable systems increased concurrently with the complexity of the systems, validation and verification techniques are becoming integrated within the development process. In particular, testing, has developed from a largely *ad hoc* and *a posteriori* process to a structured approach playing a primary role throughout the whole design and development process. One of the major shifts in perspective has been the importance of building testing artefacts independently of the system, making them more applicable to future iterations of system refinement and extension. Such an approach enables, for instance, reusing much of the effort for a suite of features appearing in a product, to future versions so as to ensure that the addition of new features do not break the old ones. However, despite that that testing is being pushed beyond its traditional confines, the time of actual deployment is typically considered beyond its active scope.

A largely orthogonal approach to address system dependability is that of runtime verification, which addresses the application of formal techniques to ensure that individual runtime traces satisfy particular properties. The approach has been applied in a variety of ways, from verifying each step of deployed systems in a synchronous fashion, to *a posteriori* verification of runtime-generated logs.

* The research work disclosed in this publication is partially funded by the German Federal Ministry of Education and Research (BMBF) within EC-SPRIDE, and the Strategic Educational Pathways Scholarship Scheme (Malta), which is part financed by the European Union European Social Fund.

One aspect of testing that is usually beyond the scope of runtime verification, is that one also considers test case generation with which the specification can be checked before deployment. Different forms of automated test case generation can be found in the literature, but one common approach is a model-based one, where an abstract representation of execution traces of interest is used to generate concrete test cases [3]. For instance, one frequently finds automata used to represent the language of traces of interest, from which actual traces are generated, and along which properties can be verified to hold. In the domain of runtime verification, one finds similar artefacts of automata (or other abstract language specifications) with properties attached to states or system configurations. However, despite their structural similarities, the two domains use these descriptions in a distinct manner.

The major difference is that, while in testing the language description is used for both generation and classification, in runtime verification it is used to check against for language membership — a duality similar to the one found in the natural language processing domain, where natural language generation and parsing use similar (or identical) descriptions, but use different techniques. Although the artefacts may be syntactically similar, they maintain distinct semantics. However, many testing approaches also include means of specifying undesirable traces, which corresponds closely to monitors one would like to instantiate at runtime.

In this paper, we study the relationship between these language descriptions and investigate how the two can be related together. In particular, we look at the automata used in the testing tool QuickCheck [10], used to describe traces of testing interest, and dynamic automata used in the runtime verification tool Larva [5]. We show how the testing automata can be converted into runtime monitors, keeping the same semantics. To illustrate and evaluate the applicability of the approach, we have implemented the approach for Erlang [7] and applied it to Riak [2], an open-source distributed key store, incorporating many different testing scenarios on which properties can be verified. Properties are designed to examine various aspects of the translation, such as its generality and its performance when transforming input properties testing the program at different granularities.

The paper is organised as follows. In Section 2, we present the semantics of both QuickCheck automata and automata used by Larva for runtime verification. In Section 3, we present a theoretical framework to relate the two formalisms' traces, and prove that the automata are equivalent with respect to this relation. This result is proved using a construction which was implemented. The approach is evaluated in Section 4, and compared to related work in Section 5. In Section 6, we summarise and conclude.

2 QuickCheck and Larva

In this section, we summarise the semantics of QuickCheck automata used for testing and a subset of Dynamic Automata with Timers and Events (DATEs) used by the runtime verification tool Larva.

2.1 Function Invocation and Control-Flow Observation

Both monitoring and automated testing use references to function invocation and execution. While monitoring of a system requires awareness of events such as the moment of invocation and the moment of termination of a function, testing requires references to the invocation of a function, where control is relinquished to the system until termination of the call. The following defines the notation we will use in the rest of the paper.

Definition 1. *Given a set Φ of function names, and type X of parameters which may be passed to these functions, we write Φ_X to denote the set of possible function names tagged with the parameters passed:*[3] $\Phi_X = \{f_x \mid f \in \Phi,\ x \in X\}$. *We will use variables f, g, h to range over Φ, x, y and z to range over X and $\hat{f}$, $\hat{g}$, $\hat{h}$ over Φ_X.*

Given an alphabet Σ, we define the set of entry event names in Σ to be $\Sigma^{\downarrow}$, the set of exit events $\Sigma^{\uparrow}$ and the set of invocations over Σ to be $\Sigma^{\odot}$. The sets are defined to be the elements of Σ tagged by $\downarrow$, $\uparrow$ and $\odot$ respectively e.g. $\Sigma^{\downarrow} \stackrel{def}{=} \{a^{\downarrow} \mid a \in \Sigma\}$. The set of observable events $\Sigma^{\updownarrow}$ is defined to be $\Sigma^{\downarrow} \cup \Sigma^{\uparrow}$. We use variable α, β and γ to range over observable function entries and exits $\Phi^{\updownarrow}$ and $\hat{\alpha}$, $\hat{\beta}$ and $\hat{\gamma}$ to range over $\Phi_X^{\updownarrow}$.

In the rest of the paper, we will be using this mode tagging symbol approach to be able to refer, for instance, to both the event fired whenever a function f is entered (with any parameter): $\Phi^{\downarrow}$, and for the event which fires when f is invoked with particular parameters: $\Phi_X^{\downarrow}$. We will allow subscript tagging of mode-tagged function names e.g. $(f^{\downarrow})_x$, which is taken to be equivalent to $(f_x)^{\downarrow}$.

Definition 2. *To reason about testing and monitoring of systems, we will assume that we have a system semantics which determines how the invocation of a function changes the state of the system. If the type of system states is Θ, we assume we have the semantics defined as the function run $\in \Phi_X^{\odot} \times \Theta \to \Theta$. The return value of the function is assumed to be made accessible in the system state.*

Using this notation, we can now formalise the notion of testing and monitoring automata.

2.2 QuickCheck Testing Automata

QuickCheck [10] is a random test case generation and execution tool, which automatically instantiates inputs of a form defined by a *generator function* and checks an Erlang program's behaviour under these inputs against a user-defined specification, or *property*. The random component stems from the generating function, which, on invocation, returns a random input. When testing a property, one would typically generate and verify a batch of inputs, the number and size of the batches being subject to the computing resources available and the criticality of the system under test. While not systematic, the simpler analysis involved with this technique allows for large volumes of tests to be executed quickly on the concrete system.

[3] In this paper we gloss over the issue of types — for the sake of this paper, we may assume that f_x is only defined if x is of the type as expected by f.

QuickCheck Finite State Automata Other than the generation of inputs for individual Erlang functions,[4] QuickCheck also allows the generation and testing of sequences of function calls, or *traces*, using *QuickCheck Finite State Automata* (QCFSA) [10]. QCFSAs are automata with arcs that correspond to function calls within the system under test. By randomly traversing a QCFSA, one generates a sequence of function calls. At every stage of the traversal, one may restrict the generation of certain traces through *preconditions* on transitions, which must evaluate to true for an outgoing transition to be taken. The QuickCheck engine then verifies the trace by executing each function in sequence whilst simultaneously traversing the automaton using this sequence as an input. With every transition, QuickCheck verifies that the *postcondition* defined on each arc holds. Every transition may also include an action which is to be executed if the postcondition is satisfied, before proceeding. This is typically used to keep track of information for the test oracle.

QCFSAs incorporate two aspects of model-based testing, namely the generation of valid system traces and their verification with respect to a property, into a single construction. Thus, a QCFSA is simultaneously a *model* and a *property*. The purpose of a QCFSA is to describe sequences of function calls and properties that should hold over each function's execution.

Example 1. Figure 1 shows a QuickCheck automaton — note that transitions consist of four expressions separated by bars, which correspond to (i) a precondition; (ii) a function's signature; (iii) a postcondition; and (iv) an action to be executed (for which we use • to denote no action).

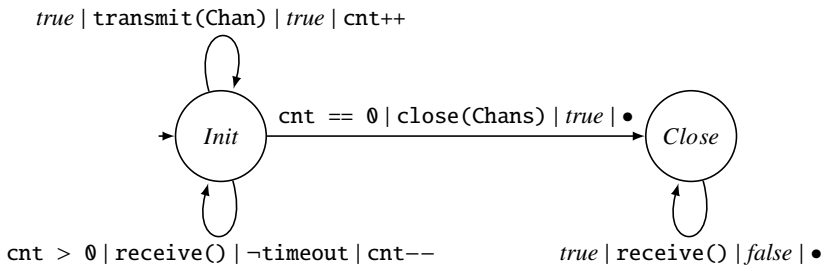


Fig. 1. A QuickCheck automaton testing `transmit` and `receive` operations over channels

This automaton generates test sequences made up of `transmit` operations, which send data over a channel `Chan` in `Chans`. The `cnt` variable keeps track of the number of outstanding acknowledgements. If `cnt` is not zero, then the automaton may accept a `receive` event with an acknowledgement of a sent message. A `close` event indicates that the channel is closed. The correctness criteria states that (i) any received `receive`

[4] In the rest of the paper, we will refer to these (as typically done in the literature) simply as *functions*, despite the inappropriateness of the term, given that they can modify their, and other entities', state.

operation may not time out, indicated over predicate `timeout`; and (ii) no acknowledgements are to be received after a `close` command.

Formalising QuickCheck Automata QuickCheck automata enable one to describe both a set of traces to check and which of these traces result in a violation. We start by defining QuickCheck automata and the languages they characterise.

Definition 3. *A QuickCheck automaton M over an alphabet of function names Φ, and a system with states ranging over Θ, is a tuple $\langle Q,\ q_0,\ t\rangle$, where Q is the set of states, $q_0 \in Q$ is the initial state and t is the transition relation labelled by (i) the precondition which should hold for a transition to be triggered; (ii) a function invocation which will be executed upon taking the transition; (iii) a postcondition which determines whether the property was violated; and (iv) a system action which is executed if the postcondition is not violated. The precondition, postcondition and action are parametrised over the parameters passed to the function invocation: $t \subseteq Q \times 2^{X\times\Theta} \times \Phi \times 2^{X\times\Theta} \times (X \times \Theta \to \Theta) \times Q$. We will write $(q,\ \text{pre},\ f,\ \text{post},\ a,\ q') \in t$ as $q \xrightarrow{\{pre\}f\{post\}}_a q'$.*

To ensure determinism, QuickCheck assumes that the preconditions on outgoing transitions from a particular state that are labelled with the same function name are pairwise disjoint [10]. Formally, for any two distinct transitions $q \xrightarrow{\{pre\}f\{post\}}_a q'$ and $q \xrightarrow{\{pre'\}f\{post'\}}_{a'} q''$ we assume that $\text{pre} \cap \text{pre}' = \emptyset$.

We can now define the semantics of a QuickCheck automaton:

Definition 4. *We define the semantics of QuickCheck automata as a relation $\Rightarrow \subseteq (Q \times \Theta) \times \Phi_X^{\odot} \times (Q \times \Theta)^{\perp}$, such that $(q,\theta) \overset{f_x^{\odot}}{\Longrightarrow} (q',\theta')$ will mean that when the automaton is in state q and the system is in state θ, the test can be extended by invoking $f_x^{\odot}$ resulting in automaton state q' and system state θ'. We also allow for transitions to $\perp$ to denote failing tests.*

The transition $q \xrightarrow{\{pre\}f^{\odot}\{post\}}_a q'$ denotes the transition from state q to q' which is taken with invocation $f_x^{\odot}$ provided that the system was in state θ and: (i) the precondition is satisfied: $\text{pre}(x,\theta)$[5]*; (ii) the postcondition is satisfied:* $\text{post}(x, run(f_x^{\odot},\theta))$. *After this, action a is invoked, leaving the system in state $a(x, run(f_x^{\odot},\theta))$. If the postcondition is satisfied, we allow for: $(q,\theta) \overset{f_x^{\odot}}{\Longrightarrow} (q', a(x, run(f_x^{\odot},\theta)))$.*
If the postcondition is not satisfied after executing f_x we add a failing transition which goes to the failure configuration $\perp$: $(q,\theta) \overset{f_x^{\odot}}{\Longrightarrow} \perp$.

$$\frac{q \xrightarrow{\{pre\}f\{post\}}_a q'}{(q,\theta) \overset{f_x^{\odot}}{\Longrightarrow} (q', a(x, run(f_x^{\odot},\theta)))}\ \text{pre}(x,\theta),\ \text{post}(x, run(f_x^{\odot},\theta))$$

[5] Although we encode predicates as the set of values which satisfy the predicate, we will abuse notation and write $pre(x,\theta)$ to mean $(x,\theta) \in pre$.

$$\frac{q \xrightarrow{\{pre\}f\{post\}}_a q'}{(q,\theta) \overset{f_x^{\odot}}{\Longrightarrow} \bot} \ \mathrm{pre}(x,\theta),\ \neg\mathrm{post}(x, run(f_x^{\odot},\theta))$$

We write $c \overset{s}{\Rightarrow}_* c'$ (with s being a string over $\Phi_X^{\odot}$) to denote the reflexive and transitive closure of $\Rightarrow$, starting in configuration c and ending in configuration c'.

It is important to observe that these automata play a dual role. On one hand, if we ignore the postconditions, they act as generators — specifying the language of traces which wants to generate test cases from. On the other hand, considering the postconditions, they also act as language recognisers — recognising the language of traces leading to a violation.

Definition 5. *The language of testable traces of a QuickCheck automaton $M = \langle Q, q_0, t\rangle$ for a system starting in state $\theta_0 \in \Theta$, written $\mathcal{T}_{\theta_0}(M)$, is defined as follows:*

$$\mathcal{T}_{\theta_0}(M) \overset{def}{=} \{s \mid \exists c \cdot (q_0,\theta_0) \overset{s}{\Rightarrow}_* c\}$$

The language of bad traces, written $\mathcal{N}_{\theta_0}(M)$, is defined to be the set of strings leading to a violation of a postcondition:

$$\mathcal{N}_{\theta_0}(M) \overset{def}{=} \{s \mid (q_0,\theta_0) \overset{s}{\Rightarrow}_* \bot\}$$

This dual role of these automata can be explained from a computational perspective — while it is easy to generate arbitrary traces in the set of testable traces, in general, it is not possible to generate traces in the set of bad traces algorithmically in an efficient way. QuickCheck uses the fact that the latter is a subset of the former, to restrict the search space when trying to find members of the language of bad traces.

2.3 Runtime Monitors

In runtime monitoring [4], the actual behaviour of the system at runtime is checked for compliance with a set of properties, or a model of the ideal behaviour. Traces are obtained though instrumentation operating either at the code or the binary level — typically performed automatically so as to maintain consistency and reduce errors. The specification of the ideal behaviour is generally done through the use of a logic or the use of automata, to enable the exact identification of bad traces. In our work we use DATEs, a class of automata used in the runtime verification tool Larva [5].

Larva and DATEs The runtime verification tool Larva [5] uses Dynamic Automata with Timers and Events (DATEs) — a form of replicating symbolic automata — to model the properties which are to be monitored. We will be using a constrained version of these automata, omitting the timers and dynamic spawning of new automata at runtime since they are not necessary in our context, and will simplify the presentation considerably.

As an event-based formalism, DATEs will be used to specify languages over an alphabet of system events which the monitor will be able to intercept. Unlike the function

names used in QuickCheck automata to *invoke* their execution, references to function names in DATEs are used to match against *observed* system behaviour, and we distinguish between the moment of entry and exit to a function. For this reason, monitoring automata will be tagged by event observations such as $f^{\downarrow}$ and $f^{\uparrow}$ and not invocations such as $f^{\odot}$. For example, a transition labelled by event $f^{\downarrow}$ will be triggered whenever the system control enters function f (no matter what parameters it receives). Note that in the case of a recursive function f, a single invocation $f^{\odot}$ may trigger such transitions multiple times.

Definition 6. *A symbolic event-based automaton over function names Φ and running with a system with state Θ is a quadruple $\langle Q, q_0, t, B\rangle$ with set of states Q, initial state $q_0 \in Q$, transition relation t and bad states $B \subseteq Q$.*

Transitions are labelled by: (i) the event *in $\Phi^{\updownarrow}$ which triggers it; (ii) a* guard condition *— corresponding to predicate over the parameter passed to the function and the system state and which determines whether the transition can be followed: $2^{X\times\Theta}$; (iii) an* action *(also parametrised over the values passed to the function as parameters) which may change the system state: $X \times \Theta \to \Theta$. The transition relation t thus satisfies: $t \subseteq Q \times \Phi^{\updownarrow} \times 2^{X\times\Theta} \times (X \times \Theta \to \Theta) \times Q$.*

It is assumed that bad states are sink-states, and thus do not have any outgoing transitions, and that there is an implicit total ordering on the transitions.

In this exposition, we assume that the function return values reside in the system state space Θ, which may also include information used by monitoring (e.g. to keep track of a counter), but which will not interact directly with the system.

Example 2. Consider a system which should ensure that if a user logs in using an account with priority level of 3 or less, he or she may not delete records. We will assume that logins occur using a function *login* which takes takes a parameter *plevel*, and record deletion happens upon executing function *delete*.[6] A DATE which verifies this property is shown in Figure 2.

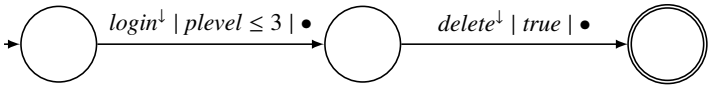


Fig. 2. Monitoring for unauthorised deletion

Note that each transition in the diagram is tagged by three bar-separated expressions identifying (i) the triggering event; (ii) the guard; and (iii) the action to be taken, respectively. Whenever no action is to be taken we still tag the transition with • so as to aid comprehensibility. Bad states are annotated by using a double circle node.

[6] We also abuse the predicate notation here and write predicates as expressions rather than as the set of parameter and system state pairs which satisfy the guard condition.

Example 3. As a more complex example, consider a system in which, after blocking a port p (using the function *block*(*port*)), no data transfer may occur on that port (using function *transfer*(*port*)). Since the function to block a port and to transfer data may be concurrently accessed, we will enforce that only once the *block* function terminates, *transfer* may not be entered. Figure 3 shows how such a property may be monitored.

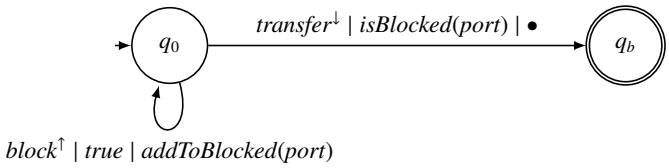


Fig. 3. Monitoring for transfers over a blocked port

There are different ways of encoding this property. The approach illustrated uses just two states — an initial one q_0 and bad state q_b. As long as the monitor is in state q_0, event $block^{\uparrow}$ (with no guard) performs an action which adds the port appearing as parameter to a set of blocked ports and goes back to state q_0. On the other hand, any *transfer* in which the port given as parameter appears in the set of blocked ports will take the monitor to state q_b:

Definition 7. *The configuration C of a symbolic event-based automaton $M = \langle Q, q_0, t, B\rangle$ is a monitor-state and system state: $C \in Q \times \Theta$. We write $(q, \theta) \xrightarrow{e_x}_M (q', \theta')$ (with e being of the form $f^{\downarrow}$ or $f^{\uparrow}$) if: (i) there is a transition from q to q' with event e: $(q, f, \text{cond}, \text{action}, q') \in t$; (ii) whose guard is satisfied on parameter x:* $\text{cond}(x, \theta)$*; (iii) the transition is the one with the highest priority with a matching event and satisfied condition; and (iv) changes the state to θ':* $\text{action}(x, \theta) = \theta'$.

Note that the total ordering on the transitions, used to choose the one with the highest precedence, ensures that the automaton is deterministic.

Definition 8. *The language of bad traces of a symbolic event-based automaton $M = \langle Q, q_0, t, B\rangle$ for a system starting in state $\theta_0 \in \Theta$, written $\mathcal{B}_{\theta_0}(M)$, is defined to be the set of strings over $\Phi_X^{\updownarrow}$ such that $\hat{\alpha}_1\hat{\alpha}_2 \ldots \hat{\alpha}_n \in \mathcal{B}_{\theta_0}(M)$ if and only if there are intermediate configurations such that: $(q_0, \theta_0) \xrightarrow{\hat{\alpha}_1}_M (q_1, \theta_1) \xrightarrow{\hat{\alpha}_2}_M \ldots \xrightarrow{\hat{\alpha}_n}_M (q_n, \theta_n)$, and $q_n \in B$.*

3 From Validation to Verification Automata

Even at a syntactic level, the automata used for testing and those used for runtime verification differ: while references to functions in testing automata are prescriptive, identifying which functions are to be *invoked*, references in the monitoring automata act as guards which trigger upon invocation or termination. The difference arises from

the fact that QuickCheck Finite-State Automata fulfil a dual role of both generators (of the testable traces) and recognisers (or violation oracles) of function call sequences. On the other hand, runtime monitoring automata act only as recognisers. In this section we show how the two approaches are related, and how monitors can be automatically derived from testing automata.

3.1 Relating Prescriptive and Observational Traces

The key issue in relating testing traces and monitoring ones is that a single function invocation may generate multiple monitorable events. At a bare minimum, a terminating invocation $f_x^{\odot}$ would generate two events $\langle f_x^{\downarrow},\ f_x^{\uparrow}\rangle$, but may generate longer traces if f invokes further functions from within it. For instance, if f is a recursive implementation of the factorial function, the trace generated by invocation $f_2^{\odot}$ would be $\langle f_2^{\downarrow},\ f_1^{\downarrow},\ f_0^{\downarrow},\ f_0^{\uparrow},\ f_1^{\uparrow},\ f_2^{\uparrow}\rangle$.

Matching $f_x^{\downarrow}$ with the exit from f with the same parameter $f_x^{\uparrow}$ does not work, since our system may use a global state, thus having the possibility of a function eventually invoking itself with the same parameter and still terminate. Counting open invocations to a function with particular parameters, and finding the corresponding exit event also fails to work since concurrency on the system side may create 'superfluous' exit events which we do not care for.

The solution we identified is that of ensuring that all functions have an additional identifying parameter which is guaranteed to be different for each invocation of that function. For instance, a function $f(n)$ which takes an integer parameter n would be enriched with an additional invocation identifier *id* to obtain $f'(id, n)$ which the system ensures that the value is different upon every invocation. In practice, programs do not follow this design pattern, but one can easily automate the transformation of a program to enforce that every function f works in this manner by changing it as follows:

```
function f(x) {
    f'(getFreshId(),x);
}

function f'(id,x) { ...}
```

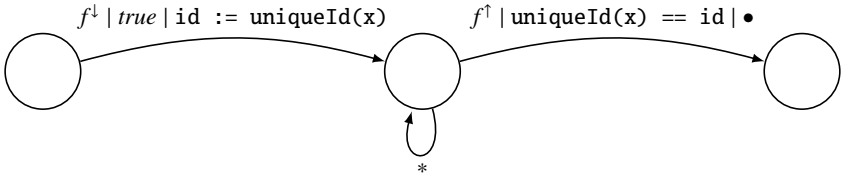
Once we can assume that we have these unique identifiers in place in the code, the relationship between invocation and event traces can be formalised in the following manner:[7]

Definition 9. *An invocation $f_x^{\odot}$ is said to be compatible with string w over events $\Phi_X^{\updownarrow}$ if: (i) $w = f_x^{\downarrow}\alpha_1\alpha_2\ldots\alpha_n f_x^{\uparrow}$; and (ii) none of events α_i are of the form $f_y^{\uparrow}$ with* uniqueId(x) = uniqueId(y). *The notion of compatibility is extended over strings of invocations over $\Phi_X^{\odot}$, ensuring that the event string can be split into substrings, each of which is compatible with the respective invocation.*

The uniqueness of the identifying parameter corresponds to the assumption that observing a system starting from state θ produce a sequence of events compatible with $f_x^{\odot}$ will necessarily end up in state run$(f_x^{\odot}, \theta)$.

[7] We assume that we can access the unique identifier of $x \in X$ as *uniqueId*(x).

The invocation identifier allows us to monitor the behaviour of the exact duration of a function in the following manner:



The reflexive transition on the middle state marked by $*$ corresponds to the set of transitions in which either (i) an event other than $f^{\uparrow}$ is received; or (ii) event $f^{\uparrow}$ is received but the condition `(uniqueId(x) == id)` does not hold. In either case, no action is performed.

Provided that the system does not change the value of the variable `id`, it is possible to prove that all strings going from the leftmost to the rightmost state in this diagram are compatible with $f^{\circlearrowright}$.

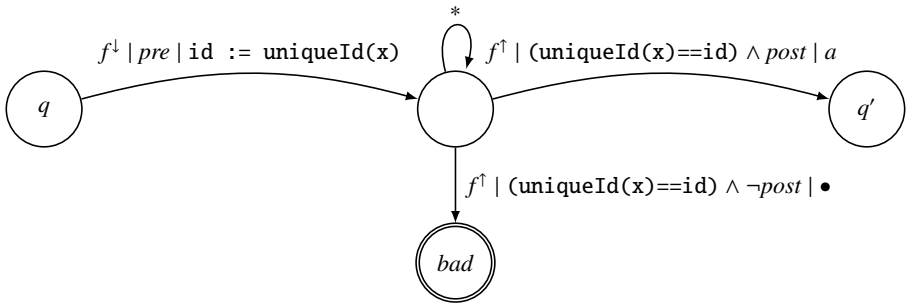
3.2 Monitors from Testing Automata

We are now in a position to formalise and prove the claim that given a testing automaton, we can construct a monitor which captures exactly the bugs which are identified by the testing automaton. The proof works by construction, which allows us to extract such automata automatically.

Theorem 1. *Given a QuickCheck automaton M, there exists a DATE M' such that for any initial system state θ: (i) given a QuickCheck bad trace $w \in \mathcal{N}_\theta(M)$, any compatible event trace w' is captured by the monitor: $w' \in \mathcal{B}_\theta(M')$; and (ii) given an event bad trace $w \in \mathcal{B}_\theta(M')$, any compatible invocation trace w' is a bad trace in M: $w' \in \mathcal{N}_\theta(M)$.*

The proof of this theorem follows from the construction of M'. The monitoring automaton M' will have the same states as M, but with (i) two additional states *idle* and *bad*; and (ii) an additional state for each transition in M. Only state *bad* is a bad state, and the initial state is the same one as in M.

The key to the construction is the translation of transition $q \xrightarrow{\{\text{pre}\}f\{\text{post}\}}_a q'$ into the following DATE fragment:



Furthermore, the transition relation in M' is finally made total by adding transitions for any uncatered for event and condition, into the idle state.

The correctness of the construction is straightforward. Given an invocation trace w which takes M from a state q to a state q', it is easy to show, using induction on the length of w, that any compatible trace w' also takes DATE M' from q to q'. This implies that for any QuickCheck bad trace $w \in \mathcal{N}_\theta(M)$, any compatible event trace w' is captured by the monitor $w' \in \mathcal{B}_\theta(M')$. On the other hand, for any event trace w going from state q to q' in M', with both states q and q' appearing also in M, the unique compatible invocation trace w' takes M from q to q'. This result allows us to prove that given an event bad trace $w \in \mathcal{B}_\theta(M')$, the compatible invocation trace w' is also a bad trace in M: $w' \in \mathcal{N}_\theta(M)$.

Example 4. Figure 4 shows the result of applying the transformation on the QCFSA illustrated in Example 1. Crucially, the transformation entailed (i) the creation of intermediate states Tx', Rx', $Close'$ and Rx'_{Cl}; (ii) the splitting up of transitions in the original QCFSA, with preconditions being checked on arcs to intermediate states, and postconditions being verified on the outgoing arc; and (iii) the creation of a *bad* state towards which failed postconditions lead. Self-loops in the input QCFSA manifest themselves as cycles between an intermediate state and the original state. For example, the self-loop over *Init* for `receive` operations resulted in a cycle being formed between *Init* and Rx'. Other outgoing arcs, such as that between *Init* and *Close* for the `close` operation, resulted in the source and target state being different, with an introduced intermediate state $Close'$.

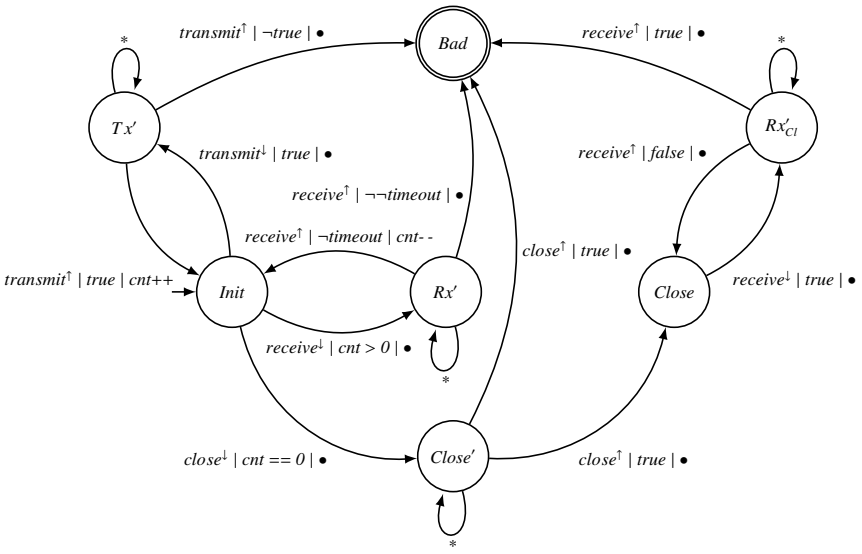


Fig. 4. The DATE obtained from the QCFSA from Example 1

Although the resulting automaton may seem complex, it can be generated in an automatic way from the testing automaton. In the next section we will look the application of this transformation to allow testing and monitoring from a single description.

4 Application on a Fault-Tolerant Distributed Database

The procedure of transforming QCFSAs into DATEs was evaluated using a real-life application, namely *Riak* [2], a distributed database implemented in Erlang. Several QCFSA properties were specified, each verifying different aspects of the system, and were automatically translated into DATEs and deployed in a custom-built runtime verification framework based on LARVA.

4.1 Riak: A Fault-Tolerant Distributed Database

Riak[2] is a distributed, noSQL database written in Erlang. Conceptually, it can be regarded as an associative array distributed over a cluster of nodes. An object, identified by a unique *key*, is inserted into the database by transmitting it to a set number of nodes, which proceed to persist that object locally. In addition to keys, Riak allows multiple concurrent key spaces, or *buckets*, to exist within the same database instance. Buckets can be seen as separate dictionaries managed by the same infrastructure. Each element in the data store can thus be identified by a $\langle bucket, key \rangle$ pair, which maps to the stored value. Consequently, while keys within the same bucket must be unique, keys need not be unique across all buckets. The mapping between a given bucket and key pair and a data element is defined using a *Riak object* data structure. Riak achieves resilience through replication, cloning objects onto multiple nodes.

To determine which nodes are involved with a particular object, each node is assigned partitions of a range of 2^{160} values, with partitions being disjoint and all values being covered by some partition. The partitioning and allocations form a structure called a *ring*. The object's key is then hashed, and nodes are chosen based on the partition within which the hashed value lands. For the purposes of replication, Riak transmits the object to **N** partitions consecutive to the first identified partition. As the cluster may change, and as database operations may be initiated from any node, Riak must communicate the state of the ring between all of its nodes. This is performed through a *gossip* protocol, which attempts to establish a system-wide, *eventually-consistent* view of the ring.

4.2 Specifications for Validation and Verification

To evaluate the approach, different QCFSAs were created and translated into runtime monitors. Broadly, these properties can be classified as being coarse- or fine-grained *control-flow* properties or properties over data structures. In addition, a QCFSA property which was packaged with Riak was converted into a runtime monitor though the described method, so as to gauge the method's utility on arbitrary third-party properties.

The *Vector Clock* Data Structure Vector clocks (or *vclocks*) are used in Riak as part of the mechanism that enforces coherence between objects residing over different nodes [2]. They are used to maintain a partial ordering amongst object update operations, keeping track of the nodes involved and the timestamp at which each operation was performed. Using vector clocks, Riak can determine which version of a given object is the most recent, and can reconcile different objects through *merging*.

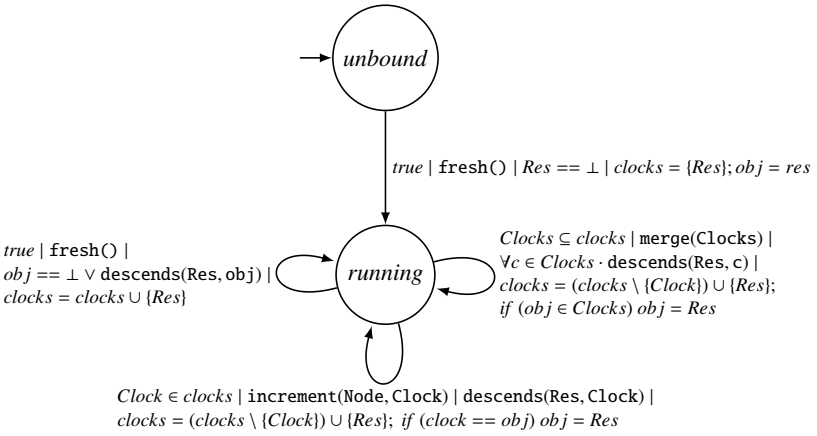


Fig. 5. Vclock QCFSA property. *Res* contains the result of an operation.

Figure 5 describes one of the properties that were investigated. It ensures that any vclocks created after or derived from a given vclock v are also its *descendants*. More specifically, the property checks that:

1. incrementing v will result in a vclock that descends from v; and
2. merging v with another vclock will result in a vclock that descends from v.

The `fresh` function returns a new vclock, which the property adds to an internal list, or pool, of vclocks *clocks*, as well as in *obj*. Beyond the creation of the initial clock, the property creates and adds further vclocks, incrementing and merging members of the vclock pool at random. The automaton's state data also contains an initial seed value for the randomization function so that the pool's size varies deterministically between generation and execution phases.

Figure 6 illustrates the DATE obtained on translating the QCFSA. The automaton is structurally similar, with transitions split via an intermediate state. While the monitor can recognize an equivalent set of bad traces, it should be noted that the automaton expects a single event sequence, with events arriving in the correct order. Consider the case where multiple vclocks exist concurrently within the system. Each monitored operation on a vclock will fire a corresponding event. While interleaving each stream into

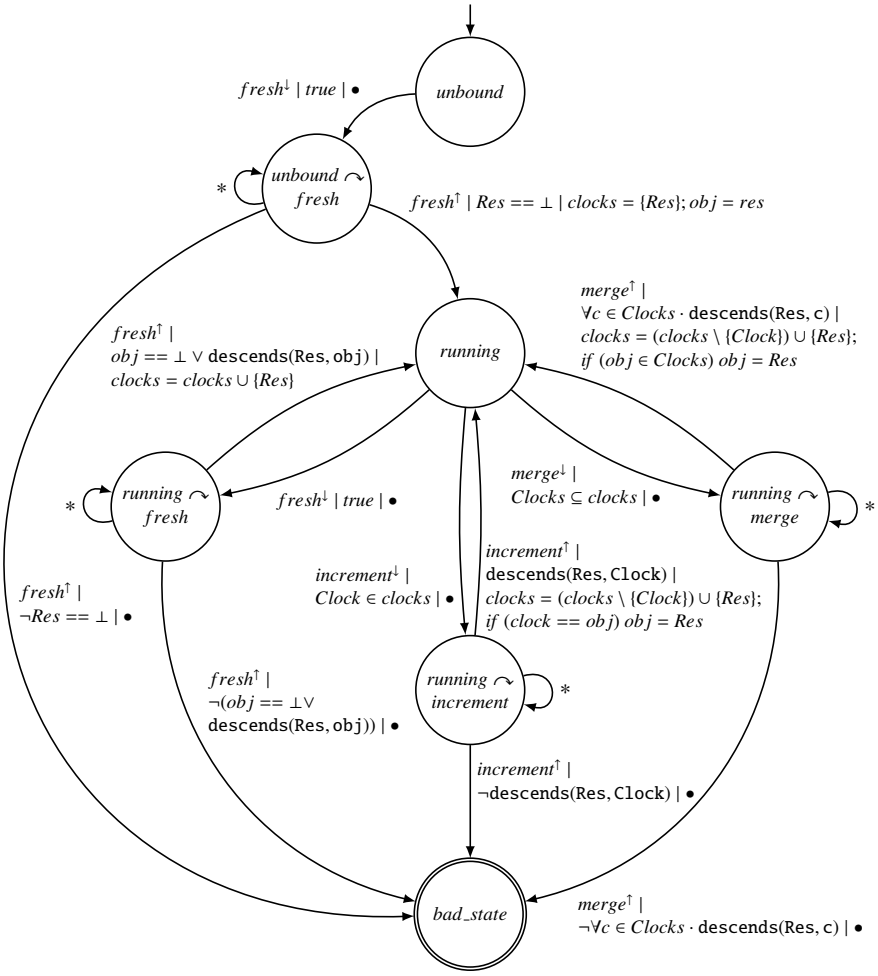


Fig. 6. DATE monitoring vclock operations.

a single event stream may work for some properties, this is often not the case, particularly when operations are being performed in parallel. Thus, the monitoring system must typically isolate each event stream and employ some monitoring policy, such as allocating a monitor to each stream or using a single monitor and interleaving streams deterministically.

The issue of uniquely identifying event sources is further complicated by Erlang's data types. For an object-oriented language, one may identify the subject of a method call as implicitly being that object on which the method is taking place. In addition, objects may be allocated an immutable identifier that is preserved across calls, simplifying the recognition of that object. In contrast, when monitoring in Erlang, one may only in-

fer the subject of an operation based on the arguments to a function, and persistent and unique identifiers cannot be attached to values in a straightforward manner.

In our monitoring framework, we considered several schemes for identifying an event-generating value. One may encapsulate each relevant value within a server process, which would then be uniquely identifiable by its process ID. This scheme, while valid, would require significant modifications to the system under test. A derivative approach is to separate streams based on the ID of their originating process, yet this is only valid if processes have at most one such monitored object, and if events that should be transmitted to a monitor do not originate from multiple sources. An alternative approach is to first annotate the system under test at the instrumented function points to also transmit the object being monitored. The runtime verification framework then maintains a mapping of objects to monitors, spawning a new monitor whenever a event originating from a hitherto-unseen object is received. Since the source object may change over time, each monitor is in charge of updating its local copy of the object with which the comparison is made. In our implementation, this is handled within each transition's `next_state_data` function, with the new value typically being copied from an input argument to a function. Thus, for example, the previous example maintains an *obj* variable in its state data, which contains the vclock to which the monitor should be tied. While this scheme works well in several scenarios, it requires detailed knowledge of the system under test, and complicates the creation of QCFSA properties, as they must incorporate the object-preservation logic.

Verification of Coarse-Grained Insertion and Retrieval When writing properties, it is apparent that the level of abstraction, or equivalently, the *granularity* of the operations being examined, directly influences their complexity. The property in Figure 7 checks the overall functioning of Riak by inserting and retrieving objects using its high-level database operations, namely `put` and `get`, which accept an object and a bucket and key pair, respectively. The property stores local copies of objects which have been inserted into the database via `put` in a dictionary *ObDict*, and verifies that objects retrieved from the database match their local versions. Since the property makes use of an internal model of the Riak database, it will be unable to verify the retrieval of objects which have been inserted by other processes, unless their insertion functions are also being monitored and update the property's model.

When operating as a QuickCheck automaton, it is necessary that the pool of objects be defined at initialisation, so as to ensure that the preconditions and state data transformations produce matching results during the generation and execution passes. To avoid hard-coding the initial state, the QCFSA is passed a set of randomly-generated objects on initialisation via the property harness, so as to lead to more varied tests.

The property is *coarse-grained* in terms of the level of detail at which the control flow is being analysed, as it concerns itself with high-level operations without testing the intermediate steps taken when executing them. The granularity at which the system is examined depends largely on the instrumentation points available and the property that must be verified, as will be seen shortly.

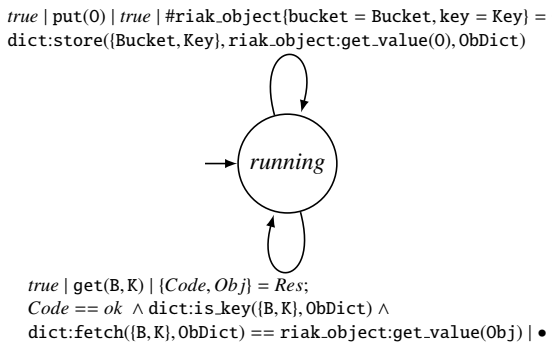


Fig. 7. QCFSA for coarse-grained object insertion and retrieval

Fine-Grained Verification of Insertion Protocol While the previous property may be simple to grasp, treating operations as monolithic blocks hinders the ability to isolate failure points should a property be violated. Points of failure can be localised to a greater degree by moving towards finer-grained properties that consider an operation's internal states. By decomposing a high-level operation's control flow, an automaton can verify that individual steps or sequences conform to a property.

Figure 9 describes a QCFSA which verifies the protocol used when committing an object to Riak, verifying that the number of positive acknowledgments received by nodes taking part in the object's persistence matches or exceeds the defined quota.

Within Riak, the protocol is implemented as a *Generic Erlang Finite State Machine*, or `gen_fsm` [7]. Broadly, the automaton goes through three stages, namely: (i) *initialisation*, where parameters such as the acknowledgment quota and the target node list are set; (ii) *transmission*, where the object in question is sent to the identified nodes; and (iii) *confirmation*, where the transmitting `gen_fsm` waits until the expected number acknowledgments are received.

The QCFSA described directly invokes the functions that implement the object insertion routine. The other alternative would have been to use the QCFSA to generate stimuli which would then be forwarded to an instantiated **gen_fsm** automaton, which would manage the actual invocation of the relevant functions. Such a property, while being valid, would not translate well into a runtime monitor, because its events would not be correlated directly to the implemented functions and would not be monitored. It would also have limited the granularity of the QCFSA's tests, as one would only be able to interact with the automaton through the defined event interface, whereas by deconstructing the automaton, one gains finer control over what can be tested at the expense of test complexity.

The QCFSA property is primarily concerned with the validity of the transmitting process's implementation, rather than the examination of network effects. Thus, when operating as a QCFSA, the system initialises a field *q_replies* within the state data structure with a stream of acknowledgment messages that matches that expected by the transmitter, which are then consumed by the automaton's receive loop within the automaton. This obviates the need to emulate the `gen_fsm`'s blocking nature within the

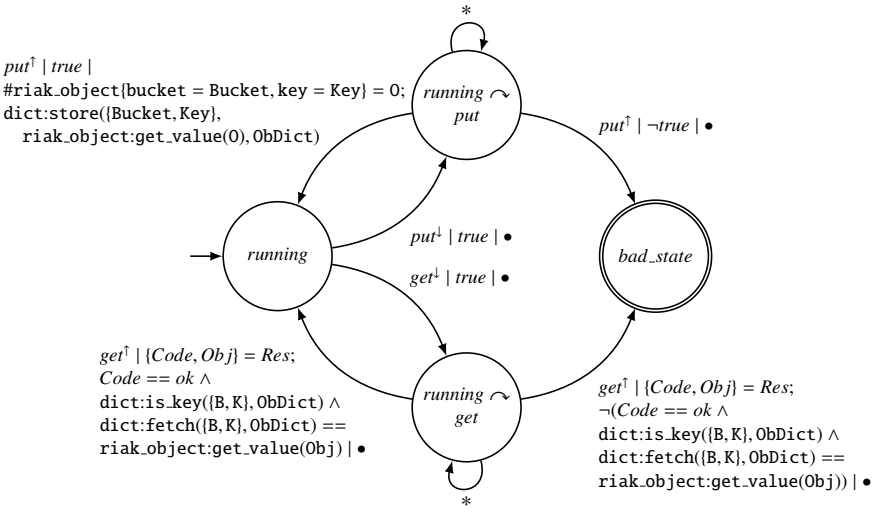


Fig. 8. DATE for coarse-grained object insertion and retrieval

property, foregoing the need to implement a mechanism for harvesting replies. Nevertheless, when executing the runtime monitor on a live system, the target nodes will send actual reply messages to the system under test, which then trigger the monitored receive functions. As the property only tests single object insertions at a time, the monitor spawning policy would be to allocate a monitor for each process initiating a `put` operation.

4.3 Results and Discussion

Overall, the use of our techniques on Riak shows the automated generation of runtime monitors from test models can be both feasible and effective. While the translation procedure preserves the original QCFSA's semantics, guaranteeing correctness of the monitors with respect to the testing automata, the study also identified a number of limitations that our approach has.

It was observed that the quality of the generated monitors can vary with the input property. For example, QCFSA properties which base their verdicts on an internally-updated state may produce an invalid verdict when deployed in a runtime scenario, unless the property ensures that its local state matches that of the system. For instance, the reply quota value used in the property defined in Figure 9 should be obtained from a live value such as a function's arguments list, rather than be set through a hard-coded value within the QCFSA's state data.

When writing properties, it was also found that certain properties that would normally be very easy to verify using DATEs can be hard to implement using QCFSAs — implying that some properties could more easily be expressed directly as monitors rather than extracted from QCFSAs. One root cause is QuickCheck's use of symbolic variables during the trace generation phase. By replacing function return values with

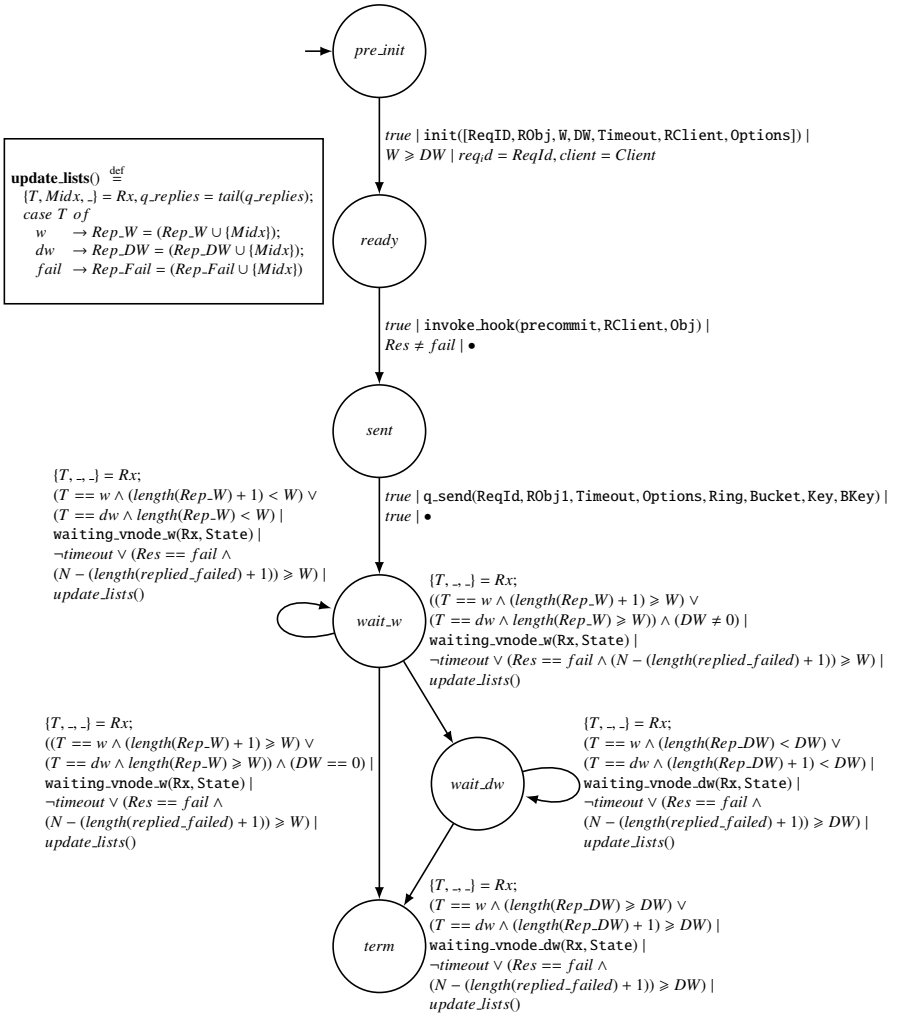


Fig. 9. QCFSA property that generates a command sequence for inserting objects into Riak. W and DW contain the quotas on writes and durable writes, while $Rep\_W/DW/Fail$ contain the hitherto processed replies, sorted by type.

symbolic variables, the system prevents properties from manipulating or directly inspecting a function's return value within the automaton's state data transformations, with [10] recommending that results should be handled as immutable black boxes. This may constrain properties to only examining abstract program behaviours rather than individual low-level operations.

5 Related Work

The language in which models are expressed often varies across different verification techniques, based on their aims and mode of operation. While this work uses QCFSAs as its base property logic, other works focus on writing properties using languages which are inherently amenable to multiple verification scenarios. *Input Output Symbolic Transition Systems* (IOSTS), proposed in [11], extend Labelled Transition Systems (LTS) by allowing the use of symbolic parameters and variables over transitions, which can facilitate static analysis. Similarly, the discrete temporal logic EAGLE [1] has been used as an input for several verification techniques. EAGLE is expressive in that it allows other temporal logics, such as LTL, to be embedded within it, whilst keeping the computational cost associated with the verification of more expressive properties to a minimum set by the complexity of the encapsulated logic and the property's size. An input property can then be used either as a test case generator or an event observer. In the former case, properties are used as inputs to the *Java PathFinder* [9] model checker, which is extended with symbolic execution capabilities. For runtime verification, EAGLE properties are used to derive monitors that examine parametrised events.

While this project focused primarily on combining testing and runtime monitoring, there are other facets of the testing process which can also be unified. The *ProTest* project [6] adopts a holistic view towards testing and verification in Erlang by integrating and automating the steps involved in creating and verifying properties. Of note, the project investigated the translation of UML specifications into QuickCheck properties and the use of QuickSpec to automatically derive a set of likely invariant properties which could then be tested. Other research on offline analysis on log files using the *Exago* tool, which extracts abstract representations of system events from logs and verifies the traces against a defined finite-state model of the system, was conducted. The *Onviso* tool was subsequently created for online event tracing across multiple nodes, and also contributed to the *PULSE* user-level thread scheduler, which can be employed within QuickCheck for testing scenarios involving concurrency. Finally, the project produced a method for efficiently converting LTL to Buchi automata, employing LTL rewriting, translation and automaton reduction. Such an automaton could then be used to derive a runtime monitor for verifying temporal properties, as also described in [8].

6 Conclusions

In this paper we have presented a technique which enables one to extract monitors from a testing specification for QuickCheck. The transformation has been proved to be correct, in that (i) if any bad trace that would have been caught by the testing oracle were to happen at runtime, the monitor would also catch it; and (ii) any violation caught

by the monitor will have also been caught by the testing oracle (if the trace were to be generated by the testing tool). The approach has been implemented in a prototype tool, which has been applied to a fault-tolerant distributed database.

Although we have studied this approach for two particular technologies — LARVA and QuickCheck, we believe that it can be extended to many other similar technologies with minor changes. In contrast, certain testing approaches can get in the way of monitoring. For instance, when using using partition testing and reducing the test case generation space (for instance, through the use of stronger preconditions), the monitors induced may be too weak since they would only be able to capture the characteristic bugs in the reduced testing search space. The interaction of partition testing and monitoring requires a deeper analysis to make our approach in such cases more effective.

We are currently looking into the use of runtime monitors to extract test case generators and oracles. Although monitors can be used as test oracles, identifying information to generate test cases is not straightforward. Through the use of a combinaton of monitoring and invocations, we hope to be able to also perform this transformation.

References

1. Artho, C., Barringer, H., Goldberg, A., Havelund, K., Khurshid, S., Lowry, M., Pasareanu, C., Rosu, G., Sen, K., Visser, W., Washington, R.: Combining test case generation and runtime verification. Theoretical Computer Science 336, 209–234 (May 2005)
2. Basho: The riak wiki. http://wiki.basho.com/ (last accessed 9 July 2012) (March 2011)
3. Broy M., Jonsson B., K.J.P.L.M., A., P.: Model-based testing of reactive systems. Lecture Notes in Computer Science, vol. 3472 (2005)
4. Colin, S., Mariani, L.: Model-Based Testing of Reactive Systems, chap. 18 Run-Time Verification, pp. 525–555. Springer (2005)
5. Colombo, C., Pace, G.J., Schneider, G.: Dynamic event-based runtime monitoring of real-time and contextual properties. In: Formal Methods for Industrial Critical Systems (FMICS). Lecture Notes in Computer Science, vol. 5596, pp. 135–149. L'Aquila, Italy (2008)
6. Derrick, J., Walkinshaw, N., Arts, T., Earle, C.B., Cesarini, F., Fredlund, L.Å., Gulias, V., Hughes, J., Thompson, S.: Property-based testing: the protest project. In: Proceedings of the 8th international conference on Formal methods for components and objects. pp. 250–271. FMCO'09, Springer-Verlag, Berlin, Heidelberg (2010), http://portal.acm.org/citation.cfm?id=1939101.1939123
7. Ericsson: Erlang reference manual user's guide version 5.7.5. http://www.erlang.org/doc/reference_manual/users_guide.html (last accessed 9 July 2012) (February 2010)
8. Giannakopoulou, D., Havelund, K.: Automata-based verification of temporal properties on running programs. In: Proceedings of the 16th IEEE international conference on Automated software engineering. pp. 412–. ASE '01, IEEE Computer Society, Washington, DC, USA (2001), http://portal.acm.org/citation.cfm?id=872023.872506
9. NASA: Java pathfinder. http://babelfish.arc.nasa.gov/trac/jpf (last accessed 9 July 2012) (April 2012)
10. Quviq AB: QuickCheck Documentation Version 1.26.2 (June 2012)
11. Rusu, V., Bousquet, L.D., Jeron, T.: An approach to symbolic test generation. In: Proceedings of Integrated Formal Methods. pp. 338–357. Springer Verlag (2000)