Department of Computer Science
University of Malta

Combining Runtime Verification and Testing Techniques

Kevin Falzon

July 2011

Supervisor: Prof. Gordon Pace

Submitted in partial fulfilment of the requirements
for the degree of Master of Science

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.

Faculty of ICT

Declaration

Plagiarism is defined as "the unacknowledged use, as one's own work, of work of another person, whether or not such work has been published" (Regulations Governing Conduct at Examinations, 1997, Regulations 1 (viii), University of Malta).

I, the undersigned, declare that the dissertation entitled

Combining Runtime Verification and Testing Techniques

is my work, except where acknowledged and referenced.

Kevin Falzon
July 2011

Abstract

As the complexity of modern systems increases, so too does their disposition towards containing faults. Highly interconnected components can fail by interacting in unanticipated ways, and large or intricate systems are inherently error-prone. Many systems are being deployed in critical scenarios where even the slightest and subtlest of errors would potentially prove catastrophic.

Testing alone as a means of verifying a system is seldom comprehensive enough, and cannot guarantee correctness. *Runtime verification* complements testing through the use of runtime monitors, which observe a system as it executes and verify that its live behaviour is valid. Runtime monitors cannot always be made use of, as they may introduce unacceptable system overheads and only detect bad states once they are reached. Deploying both techniques on a single system would increase its reliability, as runtime verification could be used to guard against faults which escape detection during testing. Nevertheless, employing both techniques simultaneously is usually not a viable option, as test suites and properties require significant amounts of time and expertise to create. In addition, applying each technique separately may result in inconsistent verification.

Runtime verification operates through the use of monitors derived from *properties*, while testing typically relies on the execution of *test cases*. As creating and checking individual test cases takes time, *model-based testing* can be adopted to create models of a system from which input stimuli can be automatically derived. The system's behaviour under these generated inputs can then be compared with a property. Model-based testing and runtime verification are thus very similar, differing essentially in how program behaviours are produced.

This project aims towards uniting both verification techniques by investigating the translation of testing models described as *QuickCheck Finite State Automata* (QCFSAs) into *Dynamic Automata with Timers and Events* (DATEs), from which runtime monitors are then derived. Each logic is defined formally, and a translation procedure is presented. Using a simplified formal model of QCFSAs as the starting logic, the process is proven to preserve the original property's test purpose under translation. More specifically, it is shown that both properties will have identical *negative trace sets*, and will identify the same set of behaviours as being invalid. The formal translation is then implemented as a program which produces a DATE from a given QCFSA. Using the DATE, the program then instruments the system under test to emit events at the relevant program points. An Erlang module implementing the DATE as a monitor listening on the events is then generated, completing the runtime monitoring framework.

The implementation is evaluated by translating four QCFSA properties designed to verify *Riak*, an open-source distributed key-store for Erlang. The operation and behaviour of each monitor is analysed, and the results are compiled into a set of observations, providing guidelines on creating properties. Part of the evaluation is devoted to the investigation of *context* and property scope, and the task of relating multiple interleaved event streams to their respective monitors when simultaneously verifying more than one property of the same type at runtime.

Contents

Chapter 1

Introduction

1.1 Background

Ensuring that a system functions correctly, especially if the system being considered fulfils a critical role, is of the utmost importance. The repercussions of a system's failure can be dramatic, expensive, and even fatal. An oft-cited example of the consequences of software errors is the failure of the *Ariane V* rocket in 1996 [AO08, JM97], which exploded due to an uncaught exception thrown while downcasting a 64-bit integer to a 16-bit signed integer. The cost of this incident was estimated to have been around $500 million. Another prominent failure was the *Pentium FDIV* bug [AO08], which caused division operations on certain Pentium processors to occasionally produce invalid results.

When verifying systems, one often resorts to *testing*. In its most basic form, testing is performed by executing one or more *test cases*, these being pairs which relate a system's expected behaviour to a sequence of input stimuli. If the system's observed behaviour within the designed environment deviates from that specified by the test case, then the test has failed. A collection of test cases is called a *test suite*. The size of a test suite depends on the difficulty in creating tests for the target system and the resources allocated to the testing effort. For example, test suites created manually are limited by the amount of available manpower, while automatically-generated test suites depend on the amount of computational power available for their generation and verification. Consequently, the test suite's size tends to be very small when compared to the size of the set of possible system behaviours. The lack of exhaustive testing can be mitigated to a varying extent by identifying *partitions* within the system's set of behaviours. As several errors tend to stem from a single cause, one may often group such behaviours into homogeneous collections and only test single instances of that failure type, reducing the number of tests that need to be verified.

As testing is rarely exhaustive, it is generally employed so as to catch the most blatant or frequent errors as well as to increase confidence in the system. However, even the most subtle of errors could lead to a system failing catastrophically. In addition, testing can be an expensive undertaking, with [CR99, AO08, Ber07] stating that it typically consumes over 50% of the total development budget.

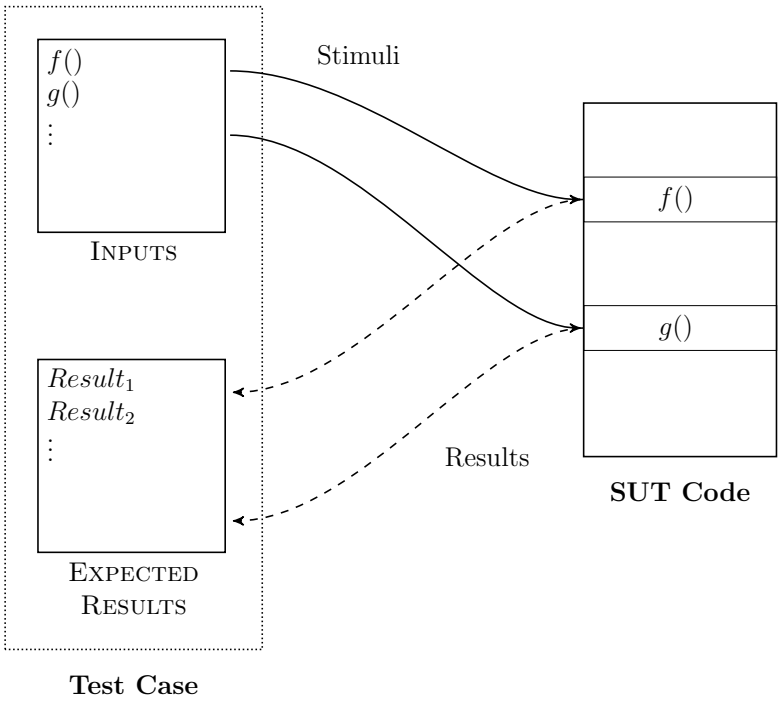

Figure 1.1: Verification using a test case

Testing is but one approach to improving a system's reliability. Another option is to use *runtime verification*. Runtime verification addresses testing's coverage issues by postponing verification until deployment. Given a formal description of a program's correct behaviour (a *property*), a runtime monitor inspects a system as it executes on live inputs and ensures that its behaviour conforms with the specification provided. This effectively bypasses the issue of coverage by only verifying relevant code paths, namely those which the system takes. By using runtime verification, one also avoids having to devise artificial input values to drive testing.

One drawback of runtime verification is that it cannot anticipate failures, and a monitor only recognises a bad state when the fault occurs. While in some cases one can design a property to recognise impending bad states, certain failures cannot be guarded against beforehand. In some instances, one can define compensatory corrective *actions* which can be made to execute on entering a bad state so as to revert to a valid system state, yet not all failures can be undone. This lack of anticipation makes certain temporal properties based on future events *unmonitorable*, as will be discussed later.

Testing can be performed on an offline system without impacting a deployed system's performance. On the other hand, runtime verification must be performed in parallel with the deployed system as it executes, potentially leeching its resources. *Offline verification* on recorded executions is possible, yet its use comes at a price, namely the fact that actions for mitigating failures can no longer be performed at runtime. Online verification has the advantage that a program's dynamic state can be analysed accurately, as the program's variables will be populated with real and concrete values. To offset the overheads incurred through monitoring, one could either increase the target system's computational capacity or make monitoring more efficient. By analysing a system's state and its evolution over

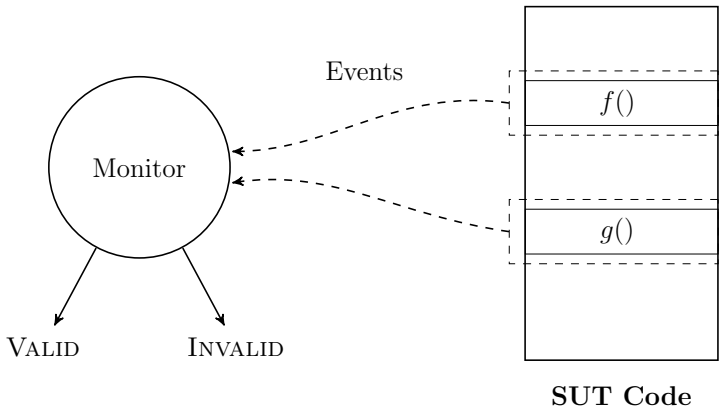

Figure 1.2: Verification using monitors observing events generated by instrumented functions

traces, the runtime verification engine can potentially adjust itself and change the degree of monitoring, and certain program behaviours which are known to be valid may be considered safe to operate unattended.

Testing and runtime verification share the same common goal, namely that of recognising incorrect system behaviours. The difference lies in that the former must also create stimuli to drive a system's execution and observe it, whereas the latter observes a live system directly as it executes. In an attempt to make the best use of both techniques it is thought worthwhile to experiment on their integrated deployment rather than their use in isolation. One approach to testing which shares significant common ground with runtime verification is *model-based* testing, which makes use of models of the system, often designed to generate test cases. These models typically contain information specifying the system's intended or implemented behaviour, or both.

In view of the effort required to create properties for runtime monitoring or models for testing, it is unlikely that both techniques will be employed to the same extent for the same project. Ideally, one would develop a single property logic which could then be transformed into an input for either technique. Removing the need to re-implement the same property twice would not only streamline the approach, but would also help ensure that the same property is being tested consistently in each mode, as errors could otherwise be introduced into the property itself during a rewrite. Changes in requirements would also be easier to perform, as alterations to the property would be immediately propagated to both techniques.

The choice of base logic dictates what can be verified. For runtime verification, a property must carry enough information to be translatable into a useful monitor. Similarly, a property used for test case generation may require additional information which a simple classifying property would not contain. For example, a runtime monitor which observes calls to a function might not require fully-specified function signatures within its property, whereas a test case generator may need additional information regarding each of the arguments' data type to direct the generation of input values. Yet a property logic which requires too much information may result unusable. For some instances, techniques have been developed to automatically derive certain properties or test cases

from a system, yet these mostly serve as aids for specifying properties and still contain a manual component. It is also essential that testing and runtime verification properties derived from the same root property will classify all system behaviours in an identical manner, producing equal verdicts. Any discrepancies between verdicts would imply that the properties being verified are not identical, leading to false positives or negatives being reported.

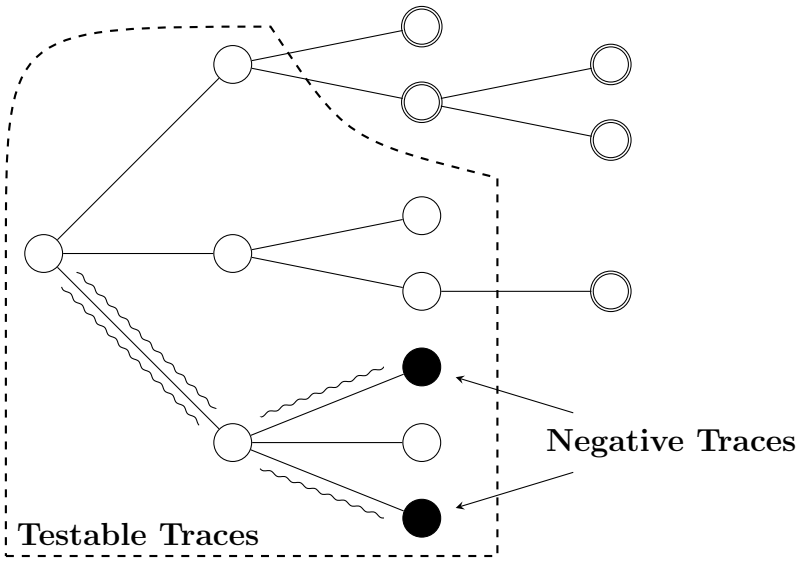

Figure 1.3: Classification of system behaviours into testable and negative trace sets

The preservation of verdicts can be demonstrated through the notions of a property's *testable* and *negative* trace sets. A property is designed to produce a verdict on a subset of a system's behaviours. The program paths which it can reason upon thus form part of the property's set of *testable traces*, these being the traces which it can test. Traces which the property classifies as leading to failure are partitioned into its *negative trace* set. As will be seen, *QuickCheck automata* incorporate both a model of the system and a property that classifies traces. The former serves to characterise the testable trace set, and determines which traces will be verified using the property, whereas the latter defines the negative trace set. Although a property may be able to produce a verdict on a trace which is not described by the model, such a verdict is unreliable, as the traces are not covered by the automaton's test purpose. When translating automata of this type, one must ensure that the result operates within the original automaton's parameters and produces consistent verdicts. Specifically, it must produce a negative verdict for all the traces within the intersection of the automaton's negative and testable trace sets. The theory behind negative and testable trace sets will be expounded upon further throughout this project, and its concepts are used extensively when analysing the transformation of models and properties into different structures.

1.2 Aim and Approach

This project investigates the integration of testing and runtime verification techniques for program correctness. As each technique requires an explicit notion of what constitutes a system's correct behaviour, the project focuses on devising a method for reusing specifications written for one technique as an input to the other. This would benefit the verification effort by removing the need to manually re-implement the same specification for each technique, saving time. An automatic translation also ensures that verification is being performed consistently, and that the same property is being verified when using testing and runtime verification. One can then opt to invest the time which would otherwise be spent re-targeting properties into developing more complete and useful properties, leading to better verification.

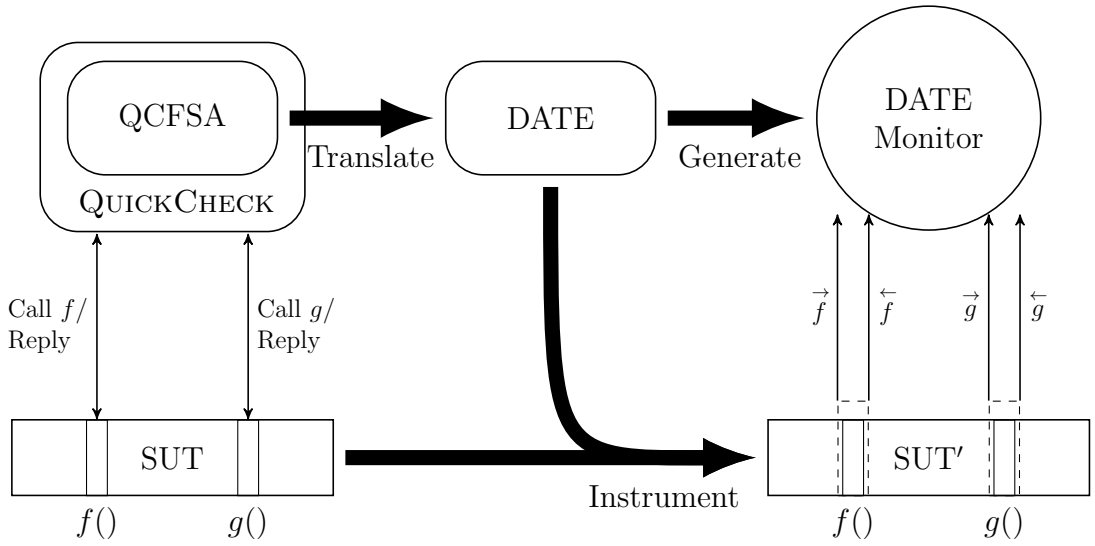

Figure 1.4: From QuickCheck automata to runtime verification

The investigation is carried out by examining the translation of testing automata into runtime monitors within the context of *Erlang* [Eri10c]. Erlang is a well-supported programming language used within industry. It focuses on reliability, fault tolerance and concurrency at a low computational cost, the latter making it an ideal medium for runtime monitoring. There also exists a significant body of work regarding the testing of programs written in this language. *QuickCheck* [Quv10] is a random test case generation tool developed for several languages, including Erlang, which supports the use of *QuickCheck Finite State Automata* (QCFSA). These automata incorporate constructs for the generation and classification of traces. After defining QCFSAs formally, a method for transforming them into *Dynamic Automata with Timers and Events* (DATEs) [Col08] is presented. The suitability of DATE as a good candidate for a target runtime verification logic is demonstrated, and QCFSA elements can be seen to be cleanly mappable onto its event and transition model. In addition, DATEs support the verification of temporal properties, leaving room for future inclusion.

Once the translation process has been defined formally for the QCFSA and DATE models, the project describes and analyses its implementation within Erlang. The trans-

lation is implemented as a tool which automatically converts QCFSAs into DATEs based on the formal constructions presented. A runtime monitoring framework for verifying DATEs within Erlang has been developed for this project. An additional tool has been developed to automatically translate DATEs into an Erlang runtime monitor designed to operate within the runtime verification framework. This also instruments the system under test, wrapping monitored functions with routines to generate and forward events. The tools are designed for interoperability, and can be chained as shown in Figure 1.4.

To evaluate the process, the project analyses the translation of four QCFSA properties written for *Riak* [Bas11], an open-source distributed key store written in Erlang. Riak incorporates 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 results produced during the evaluation stage serve as a basis on which the translation's applicability is determined. The primary result is that while the translation works for single event streams, care must be exercised when deploying multiple concurrent monitors. Complications that arise owing to QCFSAs' lack of explicit *context* make it difficult for the monitoring engine to extract the separate event streams from their interleaving. The translation may also be compromised should the QCFSA make use of its private state data when the locally stored values do not reflect the system's true state. The evaluation elaborates on these issues, and proposes a series of recommendations which should be followed when writing the initial QCFSA so as to ensure that the translation will preserve the original property's test purpose.

1.3 Document Structure

The document is structured as follows:

Chapter 2 concerns *testing*, describing its various forms and applications as well as its inherent pitfalls, notably its *coverage* issues and the difficulty in creating comprehensive test suites. Several approaches at automating the different parts of the testing process are covered. The concept of a property's set of testable and negative traces with respect to a program's possible set of behaviours is introduced. The chapter concludes with an overview of three Erlang testing frameworks, namely *EUnit*, *Erlang Common Test* and *QuickCheck*, with particular emphasis on the latter.

Chapter 3 provides an overview of *runtime verification*, defining its aims and modes of operation and issues, concluding with a description of DATEs.

Chapter 4 presents a formal model of QuickCheck automata and their translation into DATEs. As DATEs were originally conceived for use within Java, any discrepancies between their reference implementation and their Erlang counterparts are listed. A simplified translation is presented through an example, and its shortcomings are identified. A more complex and comprehensive translation is then described and applied to the example. The chapter concludes with recommendations on property writing by characterising the effects of preconditions on a property's testable

trace set, as well as the limitations imposed by QuickCheck's use of the state data construct.

Chapter 5 reviews the implementation of the property translation and runtime monitoring framework. Part of the discussion is devoted to describing several mechanisms for adding *context* to properties so as to allow concurrent event streams to be monitored.

Chapter 6 introduces *Riak*, the application on which the case study is conducted. Four properties are translated using the tools and concepts introduced in the preceding chapter, and the behaviours of the resulting monitors are examined. The properties attempt to quantify the translation's applicability when testing data structures and a program's control flow at different granularities. One of the properties formed part of the Riak package, and its translation was used to evaluate the procedure's performance when applied to third-party properties.

Chapter 7 draws results from the evaluation and attempts to determine the parameters within which the translation can be applied. This is followed by a review of other existing studies that are similar or related to this project.

Chapter 8 concludes the document, listing the project's core contributions and possible directions for future research.

Appendix A provides an overview of the pertinent Erlang constructs and concepts used within this document.

Chapter 2

Testing and QuickCheck

2.1 Introduction

Testing a system thoroughly can be arduous, yet in the absence of formal verification, it is often an inescapable requisite. Traditionally, testing has been and indeed often still is largely a manual procedure, requiring varying degrees of human insight and intervention. When testing, one must first decide *what* should be tested, and *how* the testing is to be performed. The former involves a full understanding of a system's required behaviour, while the latter concerns mechanisms for ensuring that the implemented system behaves as required. A program's expected behaviour can be characterised using a *property* written in a structured notation. Alternatively, one can employ *test cases*, which define the expected behaviour of individual executions. Each technique has its own implications on coverage, generality and ease of verification, and must be deployed using different verification mechanisms.

Testing can be performed at varying levels of abstraction and at different stages of the development process. The relationship between these two modalities is traditionally represented through the *V-model* hierarchy, which will be introduced later on in this chapter. Augmented versions of this model exist, including an alternative hierarchy which takes *components* as the fundamental functional units of a system. Components are often used in conjunction with *contracts*, which specify a component's behaviour under different scenarios.

As the process of manually crafting properties and test cases is involved and error-prone, there is much to be gained from automation. Certain aspects of testing, such as instrumentation and result monitoring, have been shown to be very amenable to automation, and are widely deployed. Several techniques for automatically inferring properties from a system or model have also been employed with varying degrees of success, as will be seen in subsequent chapters.

Simply creating a test suite or property is not sufficient, as the system must then be checked to ensure that it behaves as required. Consequently, several automated techniques were developed to check for *compliance* between a property or test case and the system

being implemented. This chapter describes three industry-level testing tools for *Erlang*[1], namely the *EUnit* and *Erlang Common Test* frameworks, and *QuickCheck*. While the former two are mostly aimed at automating the repetitive housekeeping tasks associated with the execution of test suites, the latter tool attempts to augment the testing process by randomly generating test cases from a description of the requirements. This facilitates the creation of arbitrarily large test suites, potentially leading to a greater degree of coverage.

2.2 Testing

Testing refers to a broad set of techniques which focus on comparing a system's observed behaviour with that which is expected, identifying malfunctions in the process. It is widely used in industry for *quality assurance*, and serves to provide a degree of confidence in the system being tested.

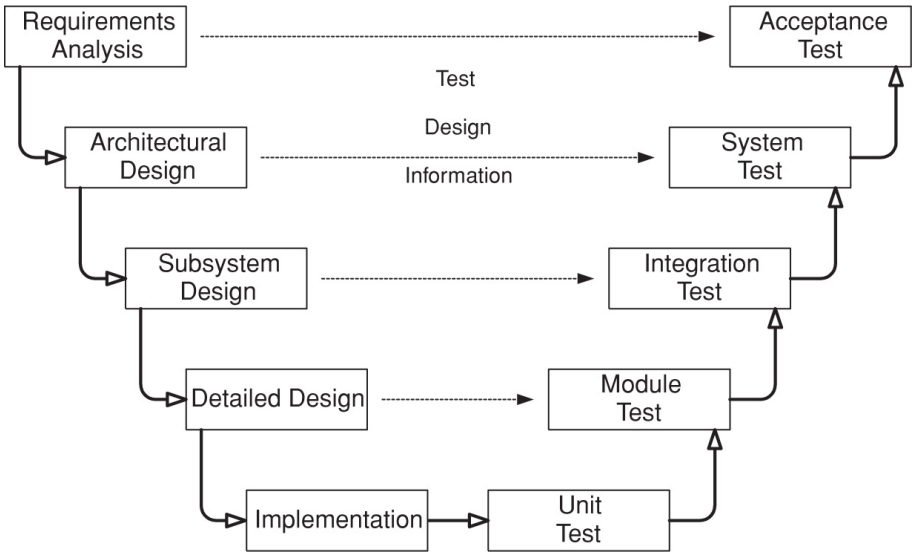

Figure 2.1: The *V-Model*, reproduced from [AO08, pg. 6]. Each development activity (left) has a corresponding test type (right).

Testing always involves the *observation* and *classification* of a sample of executions [Ber07]. A sample will define the limits of the test, and must be representative of the system's behaviour. While a larger test sample will encompass more of a system's behaviours, one cannot always test using very large samples due to the computational cost incurred from their execution. Thus, it is important that a judicious method of *test selection* is adopted, choosing samples that are comprehensive enough whilst remaining of manageable size. The degree to which a given sample encompasses a system's relevant behaviour is often quantified in terms of several different *coverage* metrics, as will be described in Section 2.3.4. Other factors influencing a sample's characteristics include the abstraction level at which the system is being observed, the system's testing environment and the point within the development process at which testing is being performed.

[1] Appendix A provides an overview of Erlang's primary language constructs.

It should be noted that there is a fundamental distinction between *validation* and *verification*. Validation is defined as the process of ensuring that a system fulfils its intended purpose, whereas during verification, one checks that the system's elements conform to their specification [AO08].

2.2.1 Types of Testing

Different phases of the development process have their own associated testing strategies. A popular model for relating development activities to their corresponding testing techniques is the *V-model* [AO08], illustrated in Figure 2.1. Activities generate information that drives the testing that is carried out at that level. Identifying a fault early on saves money, as it stops faults from propagating into subsequent levels.

Every level's testing strategy is designed to inspect different aspects of the system. Lower levels of the model test an implementation directly at a fine granularity. As one moves towards the upper layers of the model, testing becomes increasingly abstract, verifying a system in terms of larger, interacting components. Finally, the top levels check that the overall system fulfils its required function, and correctness is gauged in terms of utility and sustainability.

Table 2.1 lists the levels of the V-model and the objective of the verification methods used at each level, as described in [AO08]. Levels are presented in increasing order of abstraction.

Type	**Checks**	**Purpose**
Unit	Implementation	Test units[2] built during the implementation phase
Module	Detailed design	Checks the behaviour of individual modules[3] in isolation
Integration	Subsystem design	Assumes that modules are correct and verifies inter-module communication
System	Architectural design	Assumes that the individual system components work correctly and checks that the assembled system complies with its specification
Acceptance	Requirements	Checks that the completed software satisfies the customer's needs

One form of testing which can be applied at all levels is *regression testing*, which is performed after changes are made to a system so as to ensure that its previous functionality has not been invalidated.

[2] A *unit* is a single unit of work, such as a method or function [MH03]. What constitutes a unit of work is somewhat subjective, and varies between contexts.

[3] A *module* is defined as a set of related units [AO08].

2.2.2 Component-Based Development

Other variations on the V-model hierarchy levels have been proposed. For example, [JuRJBP07] describes a similar classification of the testing processes, replacing the *unit* with a *component* as the smallest testable system element. This is based on the principle of *Component-Based* (CB) software, which is built using prefabricated pieces of software (components). CB systems differ from *Object-Oriented* (OO) designs primarily in that OO focuses on implementation, whereas CB development is more concerned with the composition of conceptual entities. With reference to the V-Model, *Component-Based Development* replaces the *unit* and *integration* test levels with the *component* and *deployment* test levels. Component tests involve the analysis of the individual components' correctness, while deployment tests attempt to evaluate component interactions following their composition. Component tests can be carried out using traditional *black* or *white box* testing approaches. Regression tests can be performed whenever components are added. Part of the difficulty in testing CB systems lies in the degree of the components' *compositional predictability*, as the order and modes of interaction between integrated components cannot always be predetermined.

[JuRJBP07] lists five *metrics* which can be employed to determine how well a component can be tested, namely component

- *observability*, where a component consistently generates the same output when given the same input
- *controllability*, which refers to the difficulty of influencing the output through the interface
- *understandability*, or how well a component's behaviour can be understood given its interface description and possibly some additional metadata
- *traceability*, or the capability of monitoring component executions
- *test support capability*, which specifies how well components can be tested using automated testing tools

Contracts and Services

Component-based systems lend themselves to the *design by contract* paradigm, where each component specifies its interface and its explicit context dependencies using *metadata* in the form of a *contract* [JuRJBP07]. In *built-in testing*, test cases are built using contracts and verified with the aid of instrumentation, and are then packaged within components. This approach is convenient in that it is easy to maintain, yet it may leave unnecessary code in the final build. Other approaches include the use of *testable architectures*, where tests are provided as additional specifications. Built-in testing suffers from a lack of accountability, and the guarantees stated by the contracts must be certified by a trustworthy entity, possibly through an external audit.

Contracts can be defined at various levels of abstraction, with [BJPW99] identifying four, namely the *syntactic*, *behavioural*, *synchronisation* and *Quality Of Service* levels. Contracts at each level are worded using different constructs and techniques. Contract languages vary in expressivity and complexity, the most expressive being natural language contracts, which in turn affect the difficulty in contract enforcement as well as determining what can be monitored. Different aspects of a contract may also be expressed using different languages. For example, [LQS08] defines a service's properties using atomic propositions over system states, whereas the composition and coordination of the individual services is defined through XML. Contract languages may also be used to define how agents should interact with the service, specifying what data should be exchanged and which operations should be executed. They may also allow one to define compensatory actions for handling business exceptions or faults. Other languages may allow contracts to bind parties using criteria that can span across different platforms, executions or service lifetimes [LQS08].

The original notion of contracts set by [Mey92] may also be augmented to provide guarantees on *security* rather than robustness [HSRT08]. A *security policy* is a description of the set of allowable system executions [AN08]. A system adheres to a policy if the set of possible program executions is a subset or equal to the security policy. In order to create security contracts, one must first identify potential security violations, possibly through the use of UML *misuse cases* derived from the system's security requirements. These are then used to generate attack trees from which individual vulnerabilities are identified. Vulnerabilities are guarded against using assertions that are checked during execution. These assertions are inserted at *breakpoints* within the code, such as function entry points. Assertions consist of zero or more *rules*, representing safety or liveness properties of the program state which must remain true. These rules are checked using a corresponding monitor. Contract security modelling allows security properties to be added to a service without modifying its implementation. It also allows the detection of logic error exploits as well as the validation of parameters through pre- and postconditions, as with normal contracts.

Contract-based components that are accessed by clients can be seen as *services* having a *contract*, a *grounding* and an *implementation* [BBCF08]. A contract defines what a service does (service signature), its behaviour, and possibly details related to *Quality of Service* (QoS), such as time limits on service operations. The service signature, or *contract definition* [BJPW99], is a full description of the service being provided that avoids revealing its implementation. For each available service operation, it specifies its name and provides a description of its input and output parameters, stating their types and listing any possible exceptions that can be raised. The service behaviour can be described using a modelling language, and serves to define how the service will operate. A service grounding provides a link between the semantic and syntactic levels, describing how the service should be accessed. This may include what protocols should be used and how messages should be structured. A service may also provide a set of additional properties which describe the service in some way, acting as metadata.

Service types can be classified into a taxonomy [BBCF08], as shown in Figure 2.2. *State-less* services do not keep track of client interactions and thus can be invoked multiple times and in any order. Conversely, *state-full* services are sensitive to the order in

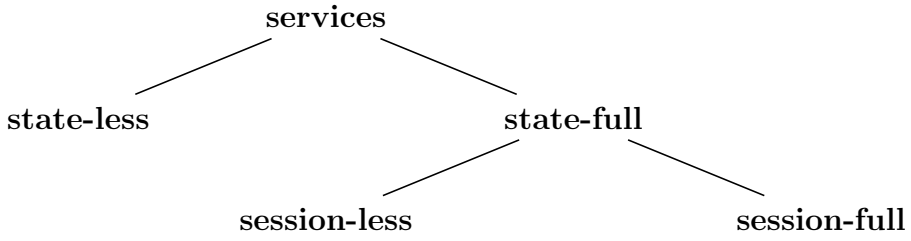

Figure 2.2: A taxonomy of service types

which interactions take place. State-full services are further classified into *session-less* and *session-full* services. Session-less services share a single virtual communication channel amongst all of the service's clients, whereas session-full services dedicate a separate channel to each client. A service's type affects its verification, as tests would have to take the service's state transitions into consideration. This is particularly relevant when monitoring contract violations. For example, while invocations of state-less services can be verified in isolation, state-full services would require that a history of the service's transactions be recorded.

It should be emphasised that contracts only describe obligations, and do not, by themselves, *ensure* a component's correctness. Similarly, [HSRT08] states that contacts guarding against security breaches may be subverted through side-channels. For example, while a contract may guarantee that all communication over a channel will be encrypted, it cannot protect against the theft of an endpoint's private key.

2.3 Automating Test Case and Property Synthesis

When testing, one typically resorts to the use of *test cases* or *properties*. A test case is a sequence of input values or calls that induces a particular behaviour from the system. The validity of this behaviour is determined through a *test oracle*, which compares the observed behaviour with that which was expected. Individual test cases are often grouped into a *test suite*. Test cases attempt to characterise a program's behaviour through enumeration, where every separate manifestation of an interesting system behaviour is verified through a test case. Testing a system sufficiently does not necessarily require that it be subjected to every possible test case which can be run on it. Instead, one can often identify partitions within the set of test cases whose members all produce the same form of observed behaviour, in which case one would only have to test one case from each partition. This is discussed in greater detail in Section 4.6.1.

Partitions aside, test cases tend to concern the verification of a single, specific system behaviour. For example, a test case for a simple square root function `sqrt`($\mathbf{N}$) that verifies the function's result with $\mathbf{N} = 1$ will make no assurances of correctness for other values of $\mathbf{N}$. Instead of enumerating $\mathbf{N}$ and running tests for every value, one could instead characterise the function's general expected behaviour and describe it as a *property*. Properties describe the behaviour of an aspect of the system, typically using some logic notation, as will be seen in Section 3.2. Using the previous example, one could define

a property which states that $\forall n \colon \mathbb{N} \cdot \texttt{sqrt}(n)^2 = n$. Given that a property has been identified, the verification problem is thus reduced to checking that a system's expected behaviour conforms to that property.

Verification is a two-stage process. One must first establish what should be verified and then carry out verification, with the former stage often dictating how the latter should be performed, as will be described in Section 2.5. The following concerns the creation stage, and considers possible avenues in automation.

2.3.1 Overview

Devising properties or comprehensive test suites for a non-trivial system can be a complex endeavour. The difficulty depends on several factors, including the amount and type of information at the tester's disposal (such as the availability of the system's source code) and the number and complexity of system behaviours that must be tested.

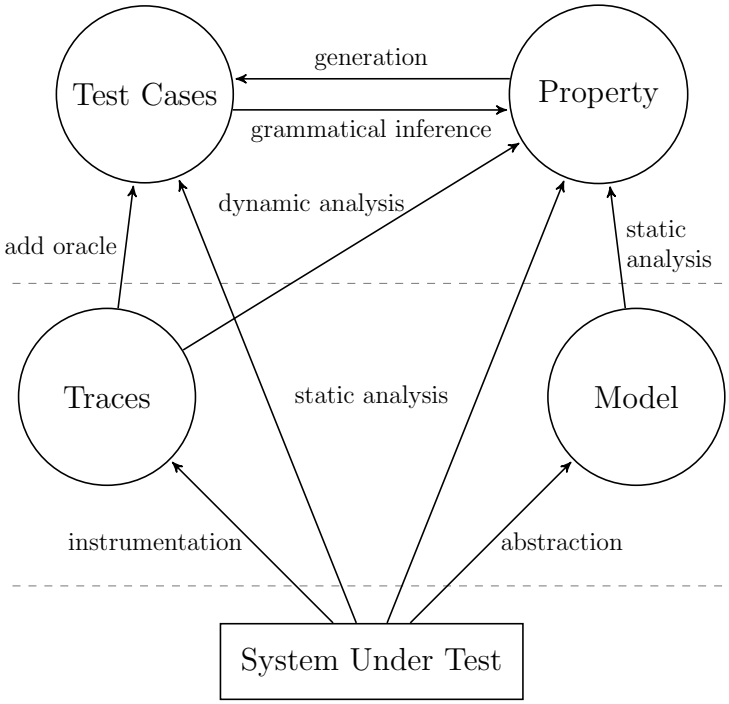

Figure 2.3: Common transformation and derivation techniques which lead to test cases and properties

Several attempts at automating the various steps of creating testing artefacts have been devised. Broadly, the approaches tend to focus on automatically inferring patterns and abstracting from the concrete implementation to more conceptual descriptions of the system. Figure 2.3 shows some of the possible transformations that can lead from an implemented system to abstract properties and test cases[4] by building on an intermediate

[4]In *test-driven development*, one would typically start from test cases or properties and move downwards towards a concrete implementation. In this case, automation could include the generation of abstract models describing the target system or even code stubs.

analysis stage. Given a live system and its source code, one can generate:

- **traces** of past system executions through instrumentation
- **a model** describing the system at a higher level of abstraction
- **test cases** which relate one or more system executions to their expected behaviour
- **properties** that model the correct behaviour of some aspect of the system

Traces and models are the products of *analysis*, and describe how the implemented system under test behaves. Conversely, properties and test cases attempt to describe how the system *should* behave. What constitutes correct behaviour depends entirely on what is required from the system. Thus, in the absence of a corresponding specification, heuristics are employed when inferring oracles directly from the system's behaviour. This leverages the fact that several common programming errors share similar patterns, and can produce a basis on which more elaborate testing artefacts can be built.

It should be emphasised that every transformation utilises its own techniques, and may require additional input information specific to that approach. These restrictions could impede the direct application of consecutive transformations. For example, inferring a property from a set of traces might require that a particular form of instrumentation be used when recording executions. Thus, paths through the hierarchy must often be performed using a single, integrated approach.

The following is an overview of some of the common techniques employed in moving from one testing artefact to another. In some of the cases considered, the creation of an artefact requires that a combination of transformations be applied. For example, certain property-generating techniques incorporate both static and dynamic analysis. In other instances, the transformation may streamline the analysis and synthesis processes into a single procedure, making use of internal intermediate representations and blurring the boundaries between artefact types.

2.3.2 Acquiring Traces

A *run* or *trace* is defined by [LS09] as a potentially infinite sequence of states or events, such as function calls. An *execution* is subsequently defined as a finite prefix of a trace. The process of recording executions from a live system depends on the environment within which it is executing, and can often be automated through *instrumentation*.

When recording traces, one must first identify which events are of interest and what information should be logged when they occur. Certain programming languages use virtual machines for executing programs, which can simplify the instrumentation process. For example, Erlang has in-built tracing mechanisms that can store executions which follow a set of specified patterns and rules without having to modify the system being analysed [Eri10c]. Similarly, Java allows the use of *reflection*, which can be used to insert logging code at the relevant points of interest directly into the program binaries at

runtime [Col08]. This also allows arbitrary actions to be triggered as soon as an event takes place. In lieu of system-supported instrumentation, one could opt to automatically inline instrumentation code into the system's source code prior to compilation, although this is subject to the source code's availability.

When instrumenting code, one introduces overheads that did not exist within the original system. These overheads may lead to the failure of certain time-critical applications by causing them to miss deadlines. In such cases, one must either compensate by increasing the system's computational capacity or by devising a non-intrusive probing mechanism. Similarly, the system must be able to accommodate any additional memory requirements imposed by the instrumentation code, and unless the traces are being consumed in an online fashion (Chapter 3), one must also dedicate memory for storing executions. In general, the performance impact imposed by instrumentation should be kept to a minimum. [CM05]

2.3.3 Generating a Model

When writing properties, it is often simpler to reason about a system in terms of its high-level functionality rather than the low-level operations involved. Models of a system provide more abstract descriptions of the system under test, allowing its behaviour to be understood more easily by a tester. In addition, [CDH+00] states that many verification tools and techniques operate on restricted models and languages such as finite-state machines, rather than directly analysing programs written in more general languages such as Java. For example, *model checking* (Section 2.3.6) demands that a model describing the aspect of the system that is to be investigated must be derived prior to verification, unless the model checker is designed to accept programs written in the language with which the system under test has been implemented. The latter approach limits the generality of the techniques employed as they become too language dependant, limiting the model checker's interoperability with other verification tools. The difference in abstraction levels offered by modelling and implementation languages is referred to as the *semantic gap*, and grows wider as the models become more restricted in relation to the language in which the system under test has been implemented. As will be seen in Section 2.5.4, models are also of great utility when coupled with a formal specification of the system's requirements, as they can lead to automated verification.

The process of creating system models manually is both time consuming and error-prone, as the system's implemented behaviour might not always be mirrored correctly by the model. Automation could simplify the process whilst ensuring a greater degree of consistency. A model serves to abstract away details which are irrelevant for the analysis being conducted. Thus, a model which describes a system at the same abstraction level at which it has been implemented is of little utility. To construct a useful model, one must first decide which aspects of the system are of interest. For example, the Bandera testing tool set [CDH+00] constructs a model of a Java program given a description of what is to be analysed. It employs *slicing*, omitting control points and data structures which are unrelated to the property being tested, leading to a reduced model which is more amenable to verification. Bandera's approach to slicing is affected by the structure

of the program, with a high degree of cohesion resulting in the removal of fewer slices. The model returned after slicing is reported to maintain the program's soundness and completeness with respect to the property being checked. Bandera also performs data abstraction to reduce a model's state space, provided that the property for which the model is being built does not depend on concrete program values but rather on the data's high-level properties. The abstraction is performed by replacing variables with symbolic representations of that data type, and operations on those values are replaced by their symbolic counterparts. As the loss of information on concrete values can cause certain tests to fail, the model also introduces a special *unresolved* symbol $\top$, which is modelled as a non-deterministic choice and delays the computation of a verdict. [CDH+00] states that due to the state explosion problem, there are greater performance gains to be had through minimising model sizes rather than by implementing complex model traversal algorithms.

Unlike the approach taken by Bandera, which requires prior knowledge of what is to be tested, [EFD05] builds models directly from Erlang programs by converting their source code into an equivalent representation expressed in a variant of μ-calculus. By developing a program using defined design patterns, the method can automatically convert the relevant module into its algebraic representation. For example, the Erlang generic client-server behaviour (Appendix A) behaviour can be automatically translated into a μ-calculus process, with `call` and `cast` operations converted into synchronous and asynchronous communication operations, respectively. Similarly, functions without side-effects are directly translated into stateless processes through rewriting, while functions with side-effects are integrated using process operators. Once an algebraic representation is obtained, one may either verify the model directly, or else derive a Labelled Transition System from it.

2.3.4 Deriving Test Cases

Test cases must fulfil a *test requirement* [AO08], that is, they must investigate some aspect of the system under consideration. A *coverage criterion* is subsequently defined as one or more test requirements that a set of tests must satisfy. Multiple tests in a suite can check the same property. A coverage criterion is said to *subsume* another if every test set that satisfies the former also satisfies the latter. Thus, for example, a branch coverage criterion can informally be seen as subsuming a statement coverage criterion, since if a test set traverses all of a program's branches, then it will have also covered all of its statements.

The number of combinations of input values (and consequently, states) that a program may accept is often enormous. This renders testing through full enumeration intractable. Thus, coverage criteria must be employed to choose which test inputs should be investigated. The *coverage level* of a set of test requirements with respect to a coverage criterion is the degree by which the test requirements satisfy the criterion. Choosing the right level of coverage is essential, as meeting a coverage criterion can be computationally expensive, and in some cases, undecidable.

Coverage criteria are chiefly employed in two scenarios [AO08], namely as:

- *generators*, where a test set is automatically derived from a coverage criterion
- *recognisers*, where a test set is created using some external method (such as manual test case creation) and is then checked for compliance with the coverage criterion

Coverage criteria tend to be used as recognisers, as the degree of automation required for generators is not always attainable. Recognisers suffer from the fact that one cannot accurately gauge whether the coverage level provided by the test set being investigated will suffice. There is no single method of evaluating the quality of a coverage criterion, and the choice of criterion depends primarily on the difficulty of generating tests and computing test requirements, as well as the efficacy of the tests in revealing faults.

In essence, a test case is a pair that relates a particular system behaviour to a verdict, or in general terms, an *execution* to a *test oracle*. In the absence of a complete pair, automation may be employed so as to infer or generate a matching component. For example, given just a set of executions, one may attempt to infer a test oracle that can classify them as failing or valid. Similarly, if a property has been identified, then it can be used as a test oracle, with the task then being to generate appropriate runs of the system on which the property should be verified. The following is a description of techniques used to infer complete test cases from traces and properties.

Test Cases from Traces

To convert a set of executions into test cases, one would have to relate testing oracles to them. In the absence of a formal specification of the program's intended behaviour, one would have to devise oracles manually. Alternatively, one may attempt to infer oracles directly from the set of observed behaviours. This approach is adopted by the *Eclat* [PE05] Java verification tool, which derives a set of JUnit[5] test cases from the system under test and a set of example program executions.

As the verifier lacks a test oracle description, Eclat uses Daikon (Section 2.3.5) to infer an operational model of the system under test. The system classifies the given executions into one of three groups based on their non-compliance with the inferred model. As the model is incomplete, a violation does not necessarily correspond to a fault. Executions are thus marked as probably being *illegal*, *normal* or *fault-revealing*, where:

- **illegal** traces are the result of presenting the system with inputs that it was not designed to handle
- **normal** traces constitute normal system behaviour, even though the derived model has been violated
- **fault-revealing** traces are those that are likely to indicate actual failures

Eclat attempts to identify partitions within the set of fault-revealing candidates, with traces following similar violation patterns being placed within the same partition. The

[5]Erlang implements a similar *EUnit* testing framework, covered in Section 2.5.1.

system attempts to reduce multiple reports stemming from the same error by only emitting a single test case per partition. This approach is limited by the inferred operational model, and does not cater for the verification of all aspects of the system. In the case of Eclat, [PE05] states that even if no faults are identified, the reduction phase will output a small set of inputs and thus the process would not hinder testing significantly.

Test Cases from a Property

If a property that must hold over some part of the system's execution has been identified, then one can create a set of test cases by generating input stimuli and relating them to an oracle implementing the property[6]. In this case, the task would be to generate a test set which operates at the correct coverage level, identifying executions which can lead to the property being violated. Automation helps to ensure that the generation process remains consistent across tests and can increase the chances that a test set comprehensively describes the property of interest.

[Ber07] describes three general approaches to test set generation, namely:

Model-based test generation, which involves the creation of a formal model that mirrors the behaviour of the system being examined. Once the system is formalised, a generator can examine the model's states and actions and derive test cases.

Random test generation, where inputs or stimuli are generated at random and fed into the system. A purely random search cannot ensure coverage, and random test generators are often directed or make use of a feedback mechanism.

Search-Based test generation treats the set of required test cases as elements of a more general search space. These approaches try to direct the search to relevant areas of the search space by employing *metaheuristics*.

Model-based test generation is centred primarily on model analysis and traversal, which are subject to the model's type and the information that it contains. An example of a model-based test generation approach is illustrated by [CR99] as follows. Initially, a model is converted into a high-level language. The test case generator then produces a *scenario tree*, with vertices representing system states and transitions consisting of tuples of the form

$$\langle \text{applied stimulus (input), generated responses (output), requirements} \rangle$$

Test cases are then extracted from the scenario tree using a modified Breadth-First Search traversal. Generating an entire scenario tree is inefficient due to its size. Thus, two heuristics are used to prune the generated tree, namely a greedy searching algorithm followed by a distance-measure based search for degenerate cases. Ideally, the generator would produce a minimal test set that maintains coverage with the smallest possible

[6] A property can also be seen as a model of the system's correct behaviour. In this context, the terms are used interchangeably.

number of test cases, yet producing such a set can be shown to be NP-complete through a reduction to the set covering problem.

Another approach taken by [BJ03] uses *Extended Finite State Machines* (EFSM) as models from which test cases are derived for conformance testing. The specification language used for writing EFSM is similar to Erlang. An EFSM consists of a set of states, a set of guarded transitions, state variables, constants and parametrised events. Invocations of system functions serve as incoming events, and application interactions are represented by outgoing events. EFSM automata maintain state across transitions through the use of a state data record structure which contains state variables, similar to Erlang *generic finite state machines* (`gen-fsm`, explained in greater detail in Appendix A.4.4). Unlike `gen-fsm` behaviours, EFSMs do not support concurrency and do not follow the single-assignment variable constraints imposed by Erlang, although this behaviour can be replicated through property rewriting. Transitions are guarded, and given that a condition on an event's parameter values is met, the automaton moves to the next state, performing a variable assignment operation and emitting an event. EFSM automata must be deterministic, that is, they cannot have overlapping guard conditions for the same event on transitions leaving a state.

Test cases are derived by traversing an EFSM from an initial node to a final state, and are assumed to be finite in length. As an automaton must be deterministic, a test case can be described by the automaton's starting configuration and the value parameters received. Thus, a *symbolic* test case consists of the control path from the initial EFSM configuration to a final state, an output function which determines the next output variable assignment given the earlier values, and a guard, resulting in a tree of possible configurations from which a system can evolve. Testing is driven by a client application that feeds in the initial user inputs and collects the output results.

Another example of model-based test case generation is presented in [HW05], using *Communicating Timed Automata* (CTA) to model real-time systems and their environment in order to generate test cases. CTAs are allowed a finite set of real-valued clocks and synchronisation channels. Transitions have triggering conditions and may specify a set of clocks to be reset on traversal. Symbolic executions of a CTA are mapped onto test cases, which consist of sequences of input events and expected outputs. The automaton traversal strategy can be changed by the tester. For example, one can opt to use breadth-first or depth-first traversals to generate traces, the latter leading to longer and possibly more effective test cases. The technique adopted also supports the estimation of various coverage metrics, such as

- **arc coverage**, which counts the number of transitions taken
- **region coverage**, which measures the number of reachable automaton states using region equivalence[7]
- **triggering condition coverage**, which uses a mixture of arc and region coverage

[7]Region equivalence is used since the state space is uncountably infinite, due to the use of real-valued clocks.

One issue endemic to model-based test case generation is that it can fall prey to the state-space explosion, leading to a lack of coverage. In the scenario described by [BJ03], the input variable domains were known to be small, typically restricted to two values, and traces were known to be short (2–4 steps), thus restricting the traversal's depth. Otherwise, the user would have to place constraints on the input domain with the help of a specially-developed tool. In [BHJP05], EFSMs are extended by allowing one to specify additional coverage criteria through the use of *observers*, which check that a generated test suite covers one or more given states or edges.

To combat the potential blow-up in the size of a generated test suite, [HBAA10] evaluates techniques for finding an interesting subset of a test suite with the greatest ability to detect faults. The *reduction* of a test suite is also NP-hard, and an attempt is made to produce an approximate result through heuristics. The heuristics investigated are the use of:

1. *random* or *semi-random* test case selection.
2. *coverage-based selection*, which attempts to maximise an identified subset's coverage following the assumption that coverage is correlated to error detection. For example, requiring that every transition be visited at least once during generation will result in a smaller, yet potentially less effective, test suite.
3. *similarity-based selection*, based on the assumption that a test suite with dissimilar test cases exposes more types of system behaviours.

The work presented in [HBAA10] focuses primarily on the evaluation of similarity-based approaches. Part of the difficulty lies in devising an adequate measure of similarity between traces. One approach is to use the *Identical Transitions* similarity measure, which counts the number of identical transitions between a given pair of traces and divides this value by their average length. Given the existence of a similarity measure, the selection algorithm must then identify a subset of test cases with the minimum total pairwise similarity measure. This task is a search problem, and several techniques such as clustering or greedy algorithms may be used. The exposition compares the use of genetic algorithms with greedy and random searches. The latter is performed primarily as a benchmark, as an effective search algorithm must be able to demonstrate a consistent improvement over a random search. In the investigated case study, the test suite derived through a genetic algorithm-based search is reported to have performed better than random and cover-based search techniques for equivalent sample sizes. One issue is that the search required the fine tuning of several input parameters specific to the problem being investigated. Other options, such as adaptive random searches and the use of different similarity measures such as *edit distance*, are also marked for future investigation.

The TEst Sequence generaTOR (TeStor) algorithm [PMBF05] is a component integration testing tool that attempts to address state space explosion issues by testing system components separately, rather than as a collection of components executing in parallel. Each system component's behavioural model is described as a UML state chart, which is then used for verification, allowing the testing of incomplete implementations. Additionally, a sequence diagram is used to specify which component interactions should

be tested. State machines are linearised, producing traces that are loop free and creating separate traces for each branching condition point. Traces that do not conform with the input specification's message ordering restrictions are rejected. While linearisation will reduce the search space through determinisation, it will lead to many valid system behaviours being left untested, although TeStor guarantees that every message of interest will be covered by at least one scenario.

The key benefit of TeStor is that it operates on artefacts that are used in industrial scenarios, namely UML charts, making itself amenable to widespread deployment. Similarly, the approach adopted by [HJK+11] uses UML automata for describing test requirements. One drawback of such notations is that the properties defined may be too abstract, limiting their capacity for deriving concrete test cases. *FShell Query Language* (FQL) is used to generate traces from the model whilst treating a UML model as a *Control-Flow Automaton* (CFA). Certain UML elements, such as states and edges, can be translated directly into their CFA counterparts, yet the test concretisation process requires the definition of relations between the edge values and their implementation. For example, edges may correspond to locations within the code or function names. Through FQL, the resultant automaton can be analysed, and paths can be generated based on the required coverage type (such as state or transition coverage) or their correspondence to specified regular expression patterns.

While tools such as *QuickCheck* (Section 2.5.3) require the use of constructs specific to test case generation, they avoid certain problems that arise during test concretisation by enforcing a more rigid operational model. As QuickCheck is designed for deployment within industrial settings, it aims towards being robust and general whilst maintaining simplicity, as will be seen shortly.

Test Cases from the SUT

Rather than using a model or analysing execution traces, one can generate test cases directly from the system under test through *static analysis*. One way of testing the system is to simply feed it randomly-generated inputs and check that a given property, such as termination within bounded time, will hold. While a purely random testing approach is simple to implement, it will fail to provide a sufficiently high level of coverage when the input space is large. Thus, random generators should also incorporate some form of feedback mechanism that directs the search. One such approach is taken by the *Directed Automated Random Testing* (DART) [GKS05] automated unit testing application. DART forsakes the use of model-based testing in favour of directed random test case generation combined with automated code inspection techniques. These include parsing the source code statically to extract interfaces, automatically generating a test driver which will utilise the aforementioned interfaces and dynamically analysing the program as it is subjected to random testing in order to direct future test inputs to execute alternate paths. DART begins by using a random *input vector* as an initial test case. A new input vector is subsequently derived by analysing the path taken on executing the previous vector and choosing input values which will cause the branches' logical conditions (path constraints) to evaluate differently. This is achieved by calculating a solution to the path constraint

with the last predicate negated. DART assumes that all program traces are finite and that functions do not have side-effects.

An advantage of random test generation is that it can test the artefact directly. This eliminates the need to derive a model, which, apart from requiring some effort to produce, has to represent the program's behaviour accurately. One issue with static analysis is that it can generate false negatives. A balance must be struck between only reporting high-confidence warnings, possibly missing other faults, and reporting all suspect faults.

[PPW+05] attempts to evaluate the performance of test suites generated using different techniques, namely:

- **fully manual** generation through interactive simulation
- **semi-random** (directed) test case generation using a model
- **undirected** random generation
- **manual generation** assisted by a model of the original system requirements

The main criteria on which a test suite is evaluated is its ability to detect errors and its degree of coverage. The tests produced aim towards uncovering errors in the model, badly formulated requirements and programming errors. Test suites generated through model-based techniques were found to be practically equivalent to manually-written tests in catching programming errors, yet the former were superior in finding requirement errors, as models provided a better understanding of the program's required behaviour. The evaluation reports that there was no correlation between the techniques employed and the gravity of the errors uncovered, and that no single technique managed to detect all of the errors within the system under test.

The model of coverage used was *condition/decision coverage*, which counts the number of different evaluations for each atomic action in a condition. Model-based techniques were found to produce suites with a higher coverage level for an equivalent number of random traces, yet [PPW+05] concedes that the models were designed by the same developer who built the system under test, which led to a biased test suite due to the tester's detailed knowledge of the implementation's behaviour. The evaluation also hints towards a correlation between coverage and error detection, yet this could not be generalised to all forms of test case generation.

2.3.5 Deriving Properties

Properties are essentially models which are taken to describe a system's correct behaviour. The primary mechanisms for deriving properties are static and dynamic analysis (or a combination of both), with the former operating directly on the system under test and the latter on executions. When employing these techniques, soundness and completeness are often an issue. In this regard, soundness refers to the ability of a property to classify valid executions correctly, whereas completeness is its ability to classify all valid executions.

Generalising executions derived from dynamic analysis may lead to unsound results, as individual runs will not necessarily encompass the behaviour of future executions [NE02].

[SC07] identifies two levels of soundness, namely *language-level* and *user-level* soundness. Language-level soundness is quoted in terms of the program's execution semantics. A property which is sound at the language level will only describe valid program traversals, whereas an unsound property will embody executions that in reality can never be performed. Static analysis techniques that fail to take context into consideration often fall prey to language-level unsoundness and generate false positives. User-level soundness is built over language-level soundness and refers to the correctness of the property with respect to the program's semantically-correct behaviour. User-level soundness cannot normally be inferred automatically from the program, as semantics tend to be represented informally. Alternatively, [SC07] states that soundness can be redefined in terms of *incorrectness*, arguing that an analysis is sound in terms of program correctness if it is complete in identifying incorrectness.

Properties from a Model

Throughout the discussion, a model is considered as a representation of the implemented system, whereas a property describes a desired behaviour with which the system should conform. From the perspective of automatic property generation, there is no true distinction between the methods used in inferring either description from the system under test, as both are ultimately characterising the concrete system. Consequently, when deriving properties automatically, models are used primarily as:

- intermediate artefacts that are inferred from the system under test and refined into a property
- user-supplied abstractions describing the system under test

In the first case, a model is built from the system through an appropriate abstraction, as described in Section 2.3.3. For a reduction to be effective, it requires a bias, such as a user-supplied property. In the second case, the user would typically provide a model of the expected behaviour rather than that which was implemented, implying that the model is a desired *property*. If the model provided is that of the concrete system, then one could infer a property from it rather than directly from the system under test, simplifying the analysis.

Properties from the SUT

The *Extended Static Checker* (ESC) is a compile-time static analysis tool that attempts to identify errors which are normally only caught at runtime, including null dereferences, invalid class casts and attempts to address elements beyond an array's bounds. ESC analyses source code directly, rather than a model of the system. ESC requires annotations similar to assertions within a program in order to guide verification. Internally, ESC uses a

model checker, and can check a program either as one whole or by analysing its constituent modules. [NE02]

ESC/Java[8] is an implementation of ESC that analyses Java programs. Java source code is compiled into a set of predicate logic formulae, expressing a method's characteristics. Methods are then checked in isolation, ensuring that their pre- and postconditions have not been violated [SC07]. ESC/Java is language-level unsound, as it allows users to define unverified and potentially false assumptions. It also disallows the use of arithmetic operations within annotations, and only analyses loops through at most one iteration. Failure to provide adequate annotations will also cause ESC/Java to be unsound, as the analyser will not factor in context. ESC/Java is criticised by [SC07] as not being cost effective due to the burden associated with annotation. Another drawback is that its unsoundness can lead to an excess of false positives being reported.

Properties from Traces

Daikon[9] is an invariant detector that tries to automatically deduce invariance properties through dynamic analysis. It can analyse programs written in several languages, including C and Java. It operates by instrumenting the system under test and analysing the apparent relationships that exist between local variables during the execution of a test suite. Variables begin with a set of associated candidate invariance properties, which are then excluded as testing progresses. Invariance is checked over a code block's entry and exit points. Daikon can identify several forms of invariance, including linear relationships and orderings between variables as well as a variable's likely domain. For example, Daikon can detect that a variable is never set to zero or that a list remains sorted throughout all of the examined executions. [NE02]

Since the number of executions in the test suite is typically much smaller than the number of possible paths through the system, Daikon may identify invariants which in reality do not hold but have not yet been refuted by a counterexample. Thus, Daikon ranks invariants using probabilities, with the likelihood of an invariant being correct increasing as the number of analysed test cases grows, giving a more accurate list of invariants.

Daikon uses offline verification (Section 3.1.1), meaning that traces must be recorded in order to facilitate analysis [NE02]. If the set of traces is too small, Daikon will be unable to infer meaningful invariance relationships as the probabilities will fail to indicate any clear candidates. On the other hand, having too many traces will cause the analysis to take longer and may require too much memory. Inheritance (in the case of Object-Oriented Programming languages) can also mislead local reasoning, as Daikon does not fully unravel object references. These issues would largely be resolved were verification to take place in an *online* manner. Another issue is that Daikon's rankings may be skewed based on what is being tested. While system tests are good candidates for invariant generation, unit tests tend to invoke methods in an unpredictable manner, misleading Daikon's statistical engine.

[8]Project website: `http://kind.ucd.ie/products/opensource/ESCJava2/` (last accessed July 2011)
[9]Project website: `http://groups.csail.mit.edu/pag/daikon/` (last accessed July 2011)

Similarly, *QuickSpec* [CSH10] is a tool that automatically produces a list of algebraic equations which appear to hold over a given set of functions written in Erlang or Haskell. These equations can serve as a basis for testing, and the tool's failure to discover laws which are known to hold may indicate that the implementation is faulty. Discovered equations are potentially unsound, as they are built through partial testing. Nevertheless, they are complete in that all syntactically valid equations which the system may satisfy can be derived from the discovered set. QuickSpec takes the compiled program under test, a list of function headers (complete with argument data types) and a set of test data generators which create instances of the data types used as inputs. A finite set of terms is generated and partitioned into equivalence classes, separating inputs which lead to varying behaviours into different partitions and relating the terms through equations.

As with Daikon, one drawback of QuickSpec is that it can produce far too much output to be useful. QuickSpec attempts to remove equations that can be derived from others within the set in order to minimise the result, and ranks equations using a simplicity measure which assumes that more complex equations are built from simpler elements. Derivability is, in general, undecidable, and one cannot produce a unique minimal set. As with Daikon, QuickSpec uses counterexamples to derive its results, yet unlike Daikon, it discovers relationships between arbitrary function input values rather than specified variables within a program.

Properties from Test Cases

Test cases are used to verify individual traces, each having an associated verdict. To convert this knowledge into a property would require that a single, overarching notion of correctness be inferred from the test suite.

Unlike the approach adopted by Daikon, where a property is derived from live executions, [AT10] uses grammatical inference to infer a finite state machine from an EUnit test suite (Section 2.5.1), which can then be used to generate more test cases or to serve as an oracle. Assuming that it has been constructed correctly, a test suite will always meet the test criteria for which it was designed. Tests in a test suite will consist of a series of assertions that compare the results of functions to their expected values. The system automatically replaces each assertion by a function (event), with traces consisting of sequences of such events. Traces are sorted into positive and negative examples based on the type of assertions employed, with exception-catching assertions symbolising negative traces. From the test suite, the system builds a minimal accepting finite state automaton describing a regular grammar that combines the individual tests into a property. [AT10] states that positive traces alone might not be sufficient for inferring a correct grammar. It is assumed that traces are closed on previous statements, that is, if a trace consisting of any **N** events is positive, then a prefix of **N-1** events will also be positive.

Once a finite state automaton has been derived, the system described by [AT10] augments the model by adding data type information to every event's arguments, which are derived from annotated function headers in the system under test. The updated model is converted into a *QuickCheck Finite State Automaton* (Section 2.5.4), with assertions translated into postconditions. The data type information is used to generate input values

when using the QCFSA as a test case generator. The stated advantage of the transformation into a QuickCheck property is that it allows a larger volume of test cases to be verified whilst remaining relatively compact. The approach is not entirely automatic. Any non-determinism introduced by the event abstraction process has to be resolved manually, as would the introduction of state preservation across multiple transitions of the resultant QuickCheck automaton.

2.3.6 Combining Analysis Techniques

To this point, static and dynamic analysis techniques have been analysed mostly in isolation. Intuitively, static and dynamic analysis differ in that the former is performed directly on a concrete or abstract representation of the system under test, whereas the latter operates on executions of the system. This distinction is not perfect, as static analysis may involve some amount of simulation of partial program traversals[10]. [SC07] suggests a different distinction, categorising techniques as belonging to static or dynamic analysis based on their concerns. A dynamic analysis technique is thus defined as one which is aimed towards observing *control flow*, whereas a static analysis technique is used to produce inferences on *data flow*.

Static and dynamic analysis have opposing characteristics. Static analysis can assure a greater degree of coverage when compared to dynamic analysis, which can only reason about individual executions. Static analysis is often impractical for very complex systems due to state space explosions. Dynamic analysis provides detailed information on local program states [CM08], as the evolution of snapshots of concrete variable and register values can be analysed over a trace. Conversely, static analysis often lacks context data [AB05], requiring analysers to either generate suitable values and approximate the set of possible program states or to simply avoid analysing unknown factors. The following section describes some of the numerous efforts at combining static and dynamic analysis for automating processes such as property generation and system verification.

Applying Analysis Techniques as Separate Phases

Several approaches apply different analysis techniques sequentially, using the output generated in one phase as an input to the next. The sequence and number of phases is dictated by the techniques in use.

Check 'n' Crash [SC07] is an example of a static-dynamic tool, which augments the output of ESC/Java through the application of a dynamic analysis phase. Using a constraint solver, it deduces variable assignments that should cause the program to crash based on ESC/Java's analysis. The assignments are then compiled into test cases, which are then executed using JUnit (Section 2.5.1). Static analysis is thus used to identify potential errors, while dynamic analysis attempts to confirm or disprove the errors' true existence.

[10] Observing a static system's behaviour under a set of defined inputs is still classified as static analysis, even though the program is, in a sense, being executed.

A tool related to Check 'n' Crash is the *Dynamic-Static-Dynamic (DSD) Crasher* [SC07]. DSD-Crasher performs an additional dynamic analysis step prior to invoking Check 'n' Crash. This attempts to address ESC/Java's user-level unsoundness through the use of heuristics by trying to characterise the program's intentions. It also employs Daikon in an attempt to infer invariants.

JNuke [AB05] performs *generic analysis*, using a context data structure to describe a program's states at different program locations. Static and dynamic analysis techniques are used in order to populate the context structures. Analysis is then performed on the intermediate abstract representation built using information harvested from the program. This allows analysis to be performed using the technique most suited for that particular task. For example, [AB05] states that thread-local properties should be analysed statically, as runtime verification would require too much memory.

Static and dynamic analysis can also be used to assist in the creation of an existing program's specification. The approach taken by [NE02] is similar to those mentioned previously. The method attempts to annotate a program using information gleaned from dynamic analysis. This is done using Daikon, inserting likely invariants as annotations which are then used to refine static analysis. The invariants are chosen based on the specification's purpose. This provides a static verifier with more context data, improving soundness. On the other hand, test suites generated through this method may vary significantly in size and quality, although even small suits may serve to uncover parts of the program's semantics. Additionally, it may be the case that while the correct specification has been inferred from a generalised behaviour, the program itself still violates the specification due to latent errors which escape the initial analysis phase.

Analysing Concurrent Programs

The behaviour of concurrently-executing elements can be very hard to analyse due to the many ways in which they can interact with each other. When verifying such systems, static and dynamic analysis can be used to different extents. For example, static analysis may be used to identify concurrent accesses to variables, notwithstanding problems brought about by *aliasing*[11] [CM08]. Aliases cannot always be identified statically if their reference is resolved at runtime, causing the verifier to miss detecting shared memory accesses, whereas a dynamic analyser can check the ultimate reference address prior to accessing the location.

Static analysis can be used to identify potential points of contention amongst processes, and dynamic analysis may be used in order to simulate different process interleavings. The tool presented by [CM08] consists of a set of static and dynamic analysis modules. The static module *Soot* generates call graphs and identifies relationships between program instructions, also detecting potential parallel interactions. The *Value Schedule Generator* (VSG) then observes the execution of random threads as they evolve through the system. Value schedules consist of pairs of conflicting variable accesses to shared variables and the

[11] Aliasing is a phenomenon whereby a shared variable is accessed through multiple handles, as when two pointers refer to the same memory location. The guises under which aliasing can manifest depend on the constructs supported by the programming language being used.

partial orderings between accesses. The VSG avoids exploring more than one interleaving for each generated partial ordering. Dynamic analysis is conducted using the *Java PathFinder*[12] (JPF) model checker, which determines whether the interleavings generated by the static checkers are feasible.

Employing Model Checking

Model checking can provide a more thorough verification of a program than unassisted dynamic analysis, as it examines the program's entire state space. *Model-driven verification* uses dynamic analysis in order to guide searches within the state space based on the behaviour reported by the program's instrumentation. Model checking can be used to make up for dynamic analysis' unsoundness when an exhaustive search is feasible, as when analysing certain localised execution paths.

The approach taken by [GJ08] uses a model of the system for verification. Apart from verifying properties, the proposed system attempts to check *modifies clauses*, which limit how a function may alter values. Modifies clauses are good candidates for dynamic evaluation, as sometimes the addresses of locations and variable values can only be evaluated at runtime. Dynamic analysis also facilitates the examination of programs at a fine granularity, as well as the evaluation of coverage metrics. Using dynamic analysis, the system tracks mutable system objects, such as variables, as they evolve over an execution. Traversals through the program are tracked by storing a trace's progress through defined program checkpoints. These are then used to create state abstractions and minimise the number of states that the checker must verify, whilst also allowing the model checker to recognise states that it has already visited.

The framework's instrumentation phase is carried out independently of the verifier, meaning that other techniques such as random test case execution can also be employed whilst maintaining the same data collection framework. This also allows analysis to be performed both during and after program deployment. Additionally, [GJ08] states that the system can be modified to infer properties.

Analysing Programs as Partitions

As mentioned earlier, different parts of a program may be easier to analyse using one technique rather than the other. This concept underpins the approach adopted by [JVSJ06], which attempts to partition a program into blocks which can then be analysed separately.

A program $\mathcal{P}$ is partitioned into two or more parts, with each part thus being a subset of the original program. Partitioning must preserve any errors that existed in $\mathcal{P}$. A program or partition can be described using a *Control Flow Graph* (CFG), which is a static representation of the program that describes all of the program's control flow. Nodes in a CFG correspond to instructions, with the start and end node representing a block's entry and exit points, respectively.

[12]Project website: `http://javapathfinder.sourceforge.net/` (last accessed July 2011)

Let G the CFG of program $\mathcal{P}$. Partition P_1 with CFG G_1 is said to be a *proper partition* of $\mathcal{P}$ if

- G_1 is a sub-graph of G
- G_1 contains the entry and exit nodes of G
- every node in G_1 lies on at least one possible path in G that goes from the entry to the exit node

Subsequently, if $\mathcal{P}$ is partitioned into parts P_1 and P_2 with CFGs G_1 and G_2, respectively, the partitioning is said to be *proper* if

- P_1 and P_2 are proper partitions
- $G = G_1 \cup G_2$
- the paths of G_1 and G_2 are disjoint

If a partition is proper, then it must be a valid program with behaviours that are a subset of the parent program. If a program is properly partitioned, then the partitions will cover all the code in the original program. Once partitions are determined, statically analysing individual partitions may become feasible, as their state space will be smaller than that of the original program. The difficulty lies in determining how a program should be partitioned. For a two-way partitioning, it is sufficient to determine only one partition, as the other can be derived as a complement by creating the smallest CFG that contains all the remaining nodes. Choosing the initial partition entirely at random may lead to false positives being reported, as impossible traversals may be generated. [JVSJ06] uses dynamic analysis to determine the initial partition. Usage statistics are collected at runtime, and executed traces are added to the initial partition. Once the size of the partition reaches a set threshold, the complementary partition containing transitions that have not yet been analysed will be derived. The initial partition should ideally be large, as otherwise the complementary partition's state space might not have shrunk to the point where static analysis becomes feasible.

2.3.7 Conclusion

While the methods illustrated can facilitate verification, fully automated verification of arbitrary programs is still unattainable, and for unrestricted languages, impossible. Automation serves as an aid, and inferred properties can be used as a basis on which subsequent tests are built.

When combining multiple verification techniques, one may find that there are mismatches between the information which is available and that which is required. This is often the result of techniques taking inputs of differing forms, with translations between inputs resulting in the loss of information. For example, ESC/Java annotations cannot express all of the invariant types derived using Daikon, limiting their interoperability. Thus, choosing the right representation and tool set is essential.

2.4 Testable and Negative Traces

The notion of test case coverage presented earlier can be extended to properties. In this case, coverage is related to the number of observable system traces that a property can correctly identify as failing, and is defined as follows.

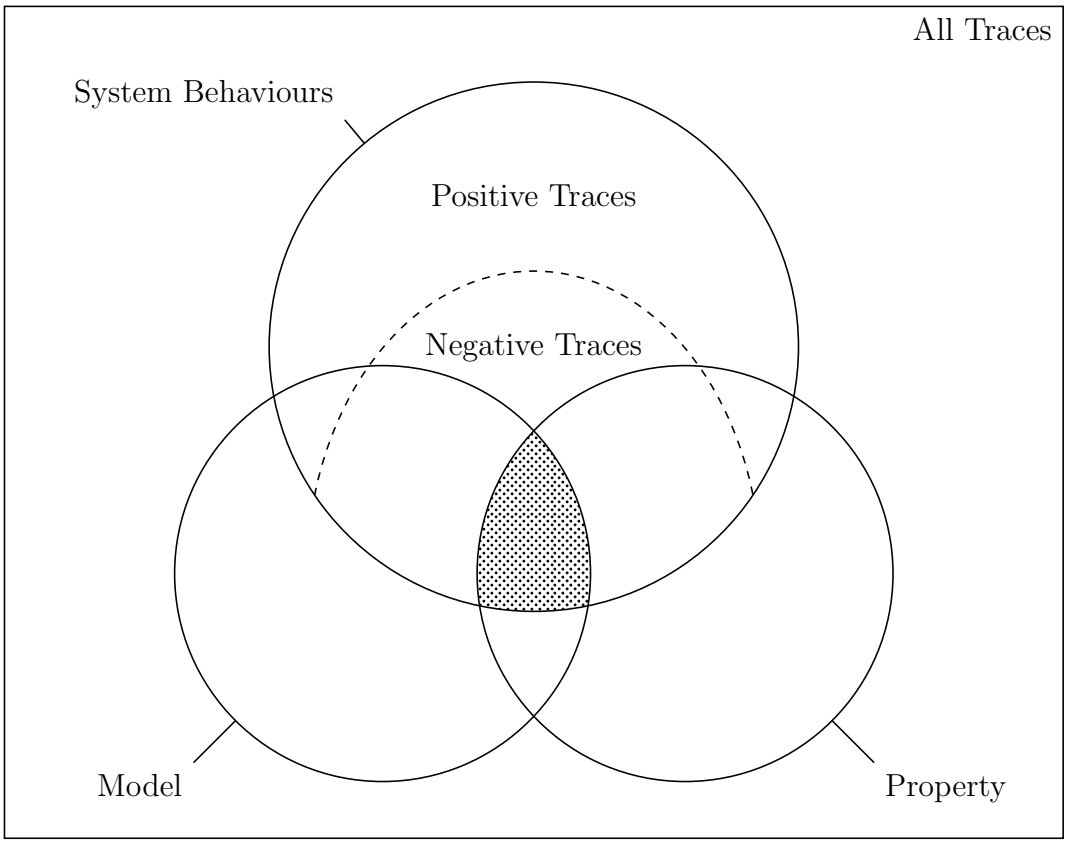

Figure 2.4: Partitioning of the set of all possible traces

Given that Σ is the set of system events with which a property to be verified is concerned, $\Gamma \stackrel{\text{def}}{=} 2^{\Sigma^*}$ is the set of all possible event interleavings. An implemented program $\mathcal{P}$ would typically only emit a subset of Γ, as its internal structure would restrict certain event sequences from being generated. Nevertheless, programs such as stateless server processes could indeed allow any arbitrary interleavings of Σ, in which case the set of legal system behaviours $\mathcal{P}_T$ would be equal to Γ. If the system is imperfect, then some of its behaviours will be invalid. Consequently, $\mathcal{P}_T$ could be partitioned into the set of allowable (*positive*) traces and the set of bad (*negative*) traces. A non-empty negative trace set would imply that the system contains faults.

The task of a property, and verification in general, is to identify a program's set of negative traces. A property is used to produce a verdict on some set of system behaviours, the size of which depends on the nature of the property and its quantification. For example, a property φ_1 stating that every valid trace must consist of at least 2 events is quantified over Γ, and can thus produce a verdict for all of $\mathcal{P}_T$. Conversely, certain properties may only be applicable to a subset of the system's behaviours. For example,

given that e and f are two events in Σ, a property φ_2 might require that every event e is followed by event f. Such a property would not be able to produce a negative verdict for traces that do not contain e events, excluding such traces from verification. Thus, the set of traces with which a property is concerned, or the behaviours which it can test, is defined as the set of *testable traces*. When presented with a trace outside of its testable set, a property must either implicitly assume that non-members are valid, or return an indeterminate verdict. A property's testable trace set may contain traces not in $\mathcal{P}_T$, in which case the property could produce a verdict on traces which never actually occur.

Models can be used to investigate specific interesting program behaviours. A model $\mathcal{M}$ can thus be seen as characterising a subset $\mathcal{M}_T$ of the traces in $\mathcal{P}_T$. If $\mathcal{M}_T \not\subseteq \mathcal{P}_T$, then the model is inaccurate and does not match the program's implementation, as it encompasses behaviours which the program never exhibits. A model can be used to direct the verification process. Instead of verifying a property over the entire set of $\mathcal{P}_T$, one may design a model which characterises the program's negative trace set and verify the property on this set, foregoing the checking of traces which are known to be valid. Ideally, only error-inducing traces are modelled, and often it is enough to model a few bad system behaviours rather than the entire negative set trace, as many bad traces may be the result of a single fault and fall within partitions (Section 4.6.1).

As will be seen in Chapter 4, QuickCheck automata use models to characterise the set of interesting traces and restrict the size of their testable trace set. These are then used in conjunction with properties that classify the individual traces as being valid or invalid. Thus, when verifying φ_2, a model would restrict the search to traces containing e and f events, which would then be checked against φ_2. The property's set of negative traces, that is, the traces which it would consider as failing, would all contain one or more $e \cdot f$ sequences. If the model generates and verifies traces which never occur, then the system may produce false negatives.

2.5 Verifying Properties and Test Cases using Erlang

Once a property or test suite has been devised, the system under test must be checked for compliance. Minimally, a property must define which behaviours are to be considered as being valid or invalid. Properties can be expressed using multiple notations, and not all notations are created equal. Some formalisations are concerned with analysing specific system aspects that dictate which verification method is to be used. For example, while *runtime monitoring* might be adequate for verifying certain *safety* properties, it may fail when checking particular temporal properties that concern future events, as will be discussed in Chapter 3.

As established earlier, testing is generally an expensive endeavour. Consequently, while unit testing can help in identifying errors which would be much harder to detect at the system level, it is often neglected due to its associated cost [Ber07]. Part of the expense stems from the difficulty of simulating the unit's execution environment and the time spent implementing mechanisms to check the unit's input and output values.

The following section describes three mainstream verification tools available to Erlang that aid in automating the verification of a system, namely the *EUnit* and *Erlang Common Test* frameworks and the *QuickCheck* random test case generation and testing tool. The former two serve to orchestrate the execution of a test suite, and handle several boilerplate operations such as the initialisation and resetting of the testing environment. They also facilitate testing by adding specific testing functionality such as assertion checking and error reporting. QuickCheck differs from the former tools in that it allows properties to be defined over quantified input variables, which are then tested using randomly-generated input values of the correct type. QuickCheck can also be used to generate and execute call chains and verify their behaviour with respect to a property.

2.5.1 EUnit

EUnit [CR09] is an open-source testing framework designed for the creation and execution of *unit tests* in Erlang, based on the JUnit[13] framework. Several other programming languages have an equivalent implementation of the underlying xUnit framework. While a unit can be of any arbitrary size, the term generally refers to either a function or a module. For JUnit, [MH03] defines units as tasks whose completion does not depend directly on that of other tasks.

Unit tests are often used to verify that an implemented module conforms to its interface contract. EUnit allows unit tests to run independently of each other, with error reports being generated on a test-by-test basis. It also simplifies the act of choosing which tests to execute, allowing a tester to enable or disable tests within a suite at will. Tests are written by creating a series of functions within an EUnit module which correspond to units in the actual program implementation. Alternatively, one can define functions which *generate* test functions.

EUnit provides tools such as *assertions* which can be used to check whether a specific condition holds at a given point, and can be used both within a test case as well as in the system under test. For example, a tester may define an assertion that checks whether a specified method returns **true**. When using EUnit, one must keep in mind that it only serves as a framework for the creation and execution of tests. EUnit on its own will not ensure coverage, as it is still the tester's responsibility to design unit tests.

An Example

Unit tests can range from the trivial to the exceedingly complex. The following section considers the testing of a very simple function and serves to highlight the basic constructs used when implementing EUnit tests. Consider the function defined in Listing 2.1, which implements the Babylonian method for approximating square roots [Car10]. The `sqrt` function takes two parameters, **In** and **Epsilon**, and returns an approximation to the square root of **In** which deviates from the true square root value by at most **Epsilon**. The algorithm is recursive, with the depth of recursion increasing as **Epsilon** approaches

[13]Project website: `http://www.junit.org/` (last accessed July 2011)

zero. Since **Epsilon** is a measure of magnitude, the algorithm only makes use of its absolute value and discards its sign.

Listing 2.1: The Babylonian method for approximating square roots

```
1  -module(sut).          % System under test
2  -export([sqrt/2]).     % Only one function exported
3
4  sqrt2(In, Root, Epsilon) ->
5    if
6      abs(In - (Root*Root)) > Epsilon ->
7        sqrt2(In, (Root + (In/Root))/2, Epsilon);
8      true ->
9        Root             % Within error bound
10   end.
11
12 % Main function, takes Number and Error value
13 sqrt(0,_) ->
14   0;
15 sqrt(N,_) when N < 0 ->
16   bad_arg;
17 sqrt(N, Epsilon) ->
18   sqrt2(N, N, abs(Epsilon)).
```

The simplest test would be a direct invocation of the function, as in Listing 2.2. The unit test calls the square root function and will only fail if the function crashes and throws an exception.

Listing 2.2: A simple EUnit test

```
1  simple_test() ->
2    sut:sqrt(25,0.01). % Returns 5.000023178253949
```

A more useful test would verify that one or more *assertions* hold. Listing 2.3 demonstrates the use of assertions within the `range_test` function, which ensure that the square root value returned lies within the specified error bound. Other assertions exist, including assertions that check for equality, raised exceptions or bounded-time execution. The EUnit framework also provides a series of debugging macros that can be used during development for displaying error and diagnostic information.

Listing 2.3: EUnit test with assertions

```
1  range_test() ->
2    Value = 25,
3    Error = 0.25,
4    Root = sut:sqrt(Value, 0.25),
5    ?assert(abs(Value - (Root*Root)) =< Error).
```

EUnit provides a tester with very fine grained control over how the unit tests will be performed. Notably, it allows one to specify which tests should execute as well as their execution order, even allowing tests to be executed in parallel. It also provides mechanisms for binding setup and clean-up code, known as *fixtures*, to individual or groups of tests. These are used to configure the environment in anticipation of the tests, replicating any relevant test conditions.

2.5.2 Erlang Common Test

As with the EUnit framework described in Section 2.5.1, the Erlang *Common Test* (CT) framework aids in the creation and automated execution of test cases. It allows one to create a set of functions which serve as unit tests, each of which can test one or more aspects of the system. [Eri10b]

The creation of test cases is facilitated by the existence of several supporting functions. Apart from the basic libraries provided by Erlang, unit tests can also make use of CT-specific functionality as well as libraries provided by the user. CT libraries provide standard functions for logging test results, reading configuration data and aborting tests. They also provide facilities for communicating over several protocols, including *Telnet* and *Remote Procedure Calls*, aiding error reporting. CT also supports *large-scale automated testing*, whereby a single dedicated Erlang node can distribute tests onto any number of CT test nodes. *Test specifications* (Section 2.5.2) determine which tests will be executed where. The master node logs all test results, collecting statistics generated by each independent CT server [Eri10b]. The existence of supporting libraries make CT suitable for white-box testing, as unit tests can invoke functions directly, as well as black box testing through several supported interfaces.

Test Cases and Suites

A CT test case is the smallest testing primitive for CT. As with EUnit, test cases can be of arbitrary length and can test more than a single property, as they can connect to multiple interfaces. [Eri10b] states that making unit tests too small may result in a high degree of code duplication, whereas large tests can obfuscate what is being tested. There are no mechanical or set rules for determining an adequate degree of test complexity.

Unit tests can be executed individually or as part of a test suite, which is implemented as an Erlang module. [Eri10b] states that test cases execute as separate processes. A test case is classified as *failing* if it crashes. Whenever a test case returns a value, it implies that the test has *succeeded*. A test may also return values with a specific interpretation, namely:

Returned Value	**Interpretation**
{skip, **Reason**}	The test has been skipped and is logged as such
{comment, **Comment**}	Test succeeded, **Comment** placed in log

Each test case has access to a list of tuples containing several test environment pa-

rameters such as the *current working directory*. Additionally, every test case can have an identically-named companion function which takes no parameters and specifies *test constraints*, including the test's maximum execution time (the default being 30 minutes), a test's data dependency as well as any arbitrary user data.

Apart from test cases, one can define *configuration functions* which prepare the environment for testing on a per-test or per-suite basis. For example, a configuration function could initialise data structures and connections which are subsequently used by the test cases. Other configuration functions can be set to execute once a test or a test suite terminates, and are often used to free up any initialised resources. A test suite can also have an associated *test specification*, which defines the tests' execution order and can be used to set working and logging directories as well as path aliases. [Eri10b]

Test Strategies

Tests in a test suite can be partitioned into a series of *groups*. Groups can contain other groups, allowing test sub-grouping. Groups are assigned a *test strategy*, which contains a list of properties that specify how testing should be carried out for that group. For example, one can specify an execution order for test cases within a group [Eri10b] by setting the relevant property to one of the following values, namely

Value	Reason
parallel	All tests are executed in parallel
sequence	All tests executed in the order in which they are defined
shuffle	Test cases are executed in random order

When specifying **shuffle**, one can also set a fixed seed value for the randomiser, making the test's conditions reproducible. Also, one can set the **repeat** property, which causes the test group to be tested repeatedly for a set number of times or until any or all test cases fail or succeed.

Logging

The CT framework includes a comprehensive logging mechanism. Test results generated by a test suite are placed within the *major log file*, which contains a detailed report of a test's execution, including test statistics, summaries and failure reasons. The major log file is also generated in HTML format and allows test case results to be viewed separately. Results for each test case are placed within the *minor log file*, with a separate file for each result. This allows results to be compared between test runs. [Eri10b]

2.5.3 QuickCheck

The ease with which properties and models can be constructed and the generality of the problem domain to which a testing tool can be consistently applied is essential to

its success, and are part of the reason why the *QuickCheck* testing tool was developed. QuickCheck is an automated testing tool which was originally designed for the *Haskell*[14] functional language. The Erlang implementation provides several extensions over the Haskell version, including *counterexample reduction* [AHJW06], which will be described shortly. QuickCheck can be used both for positive testing, checking that a system behaves as expected when given valid inputs, and negative testing, verifying that invalid inputs are handled correctly.

Instead of specifying individual test cases, QuickCheck can automatically generate random values as stimuli for the system. The values are entered, and the observed behaviour is verified against a specified property. This allows the property to be tested under a multitude of conditions, extending the test's capabilities without additional manual intervention. Apart from values, QuickCheck can also be used to generate call sequences from a defined model of the system.

When a test case fails, QuickCheck reduces the counterexample into a *minimal failing case* [PA09]. [ZH02] describes a test case as being minimal when none of its subsets cause the test to fail. While there may exist some globally minimal test case, finding it can be very computationally expensive. Thus, rather than finding a global minimum, QuickCheck starts with a counterexample and minimises it to the point where each element is considered significant and contributing to the test's failure. Counterexamples in QuickCheck are minimised by simplifying the input values used and removing calls to functions which do not contribute to the property's failure. This facilitates fault finding by abstracting the error and emphasising the underlying cause. For example, simplifying an integer input may indicate that an observed error is triggered by the value's sign rather than its absolute value. Minimisation is also useful when performing *fuzz testing* [ZH02], where large random strings are used as inputs to a system, by eliminating noise from the test data and exposing the characteristic that led to failure.

The following is a description of three primary components of QuickCheck, namely *properties*, *generators* and *finite state machines*.

Properties

Properties are QuickCheck's fundamental verification constructs, and test aspects of a system. As described by [Quv10], a property has the following structure:

`?PROP_TYPE(`**X**, **Generator**, **Property**`)`

- **Generator** is a set of values combined with a probability distribution.
- **X** is a value bound from **Generator**. Values are chosen with a frequency specified by the generator.
- **Property** is a fragment of code which tests the aspect of the system being investigated, returning **true** if the test succeeds.

[14] Project website: `http://www.haskell.org/` (last accessed July 2011)

Listing 2.4: A failing QuickCheck property for testing square root

```
1  -module(main).
2  -include_lib("eqc/include/eqc.hrl").
3  -export([main/0]).
4  -define(R_ERROR, 1.0e-4).
5
6  % Test that sqrt works by checking that
7  % ∀X : int ·X − √X² ∈ [-R_ERROR, R_ERROR]
8  prop_sqrt() ->
9    ?FORALL(X,
10         int(),  % Fails: generator produces -ves
11         begin
12           Eps = X - math:pow(math:sqrt(X), 2),
13           (Eps =< ?R_ERROR) and (Eps >= -?R_ERROR)
14         end).
15
16 main() ->
17   eqc:start(),  % Start the server and test
18   eqc:quickcheck(prop_sqrt()),
19   eqc:stop().   % Stop server
```

Listing 2.4 shows the implementation of a property `prop_sqrt()`, which tests the square root function by checking whether squaring the root will result in the original value being returned. The square root function should fail when given negative values. The following observations can be made regarding the property described, namely:

- The property uses the universal quantifier `FORALL` and checks that the property holds for all the values produced by the generator.
- So as to compensate for rounding errors introduced by floating-point operations [ACH08], the product of the roots is not compared directly to the original value. Instead, the property allows for deviations within a tolerance threshold set by **R_ERROR**.
- The QuickCheck module must be launched before testing, and should be terminated once testing is complete.
- In its current form, the test will fail. This is because the `int()` generator may bind a negative value to **X**. One can restrict the generated values to the natural numbers by replacing the generator with `nat()`.

[PA09] states that properties are simpler to comprehend than test cases, both because the latter may be overly complex as well as the fact that several test cases may have to be implemented in order to describe a single equivalent property. As a general rule, properties should not re-implement the functions which are being tested, as faults within the implementation would be carried over into the property, defeating its purpose. Several property and quantifier types exist, with [Quv10] describing the following:

Property	Description
`FORALL(`**X**`,`**Gen**`,`**Prop**`)`	Checks that **Prop** holds for all values of **X** in **Gen**.
`IMPLIES(`**Pre**`,`**Prop**`)`	Checks that **Prop** is true whenever **Pre** is true. The precondition should be satisfied often, as tests must be generated before they are checked.
`WHENFAIL(`**Action**`,`**Prop**`)`	Invokes **Action** whenever **Prop** is false.
`TRAPEXIT(`**Prop**`)`	Tests **Prop** within a separate process. If the process terminates due to a linked process exiting (Section A.4.1), then the test is considered to have failed.
`TIMEOUT(`**Limit**`,`**Prop**`)`	Tests **Prop**, reporting failure if a result is not returned within **Limit** milliseconds.
`ALWAYS(`**N**`,`**Prop**`)`	Tests **Prop** for **N** times, stopping if **Prop** fails at least once.
`SOMETIMES(`**N**`,`**Prop**`)`	Tests **Prop** for **N** times, failing only if all tests fail, that is, if **Prop** is *never* true.

Generators

Generators are used to generate random data which will drive the system being tested. Generators must specify a set of generated values, a probability distribution over the set and a process for shrinking generated values used during counterexample minimisation [Quv10]. Generators are written as functions using a combination of supplied generators and macros. Generating functions can also be placed in structures such as tuples, records and lists in order to construct more complex data structures.

The following is an overview of the primary macros that are used when writing generators.

Generator	Description
`LET(`**Pat**`,`$\mathbf{G}_1$`,`$\mathbf{G}_2$`)`	Maps generated values from $\mathbf{G}_1$ onto **Pat**, which can then be used as values in generator $\mathbf{G}_2$.
`SUCHTHAT(`**X**`,`**G**`,`**P**`)`	Maps values of **G** onto **X** whenever **P** is true.
`SUCHTHATMAYBE(`**X**`,`**G**`,`**P**`)`	Same as `SUCHTHAT`, except that the result is returned as a $\{\mathbf{value}, \mathbf{X}\}$ pair, or **false** if a suitable **X** was not found in a "reasonable" amount of time.
`SHRINK(`**G**`,`**Gs**`)`	Allows one to define shrinking rules. If a test case fails, values generated by **G** are translated into values generated by the alternative shrinking generators defined in the list **Gs**.
`LETSHRINK(`**Pat**`,`$\mathbf{G}_1$`,`$\mathbf{G}_2$`)`	`SHRINK` is often embedded within a `LET` command, with shrinking rules being applied to values mapped from a generator specified by the `LET` command. `LETSHRINK` combines `LET` and `SHRINK` into a single command. Values from $\mathbf{G}_1$ are mapped onto **Pat**, which are then used within $\mathbf{G}_2$. In case of failure, **Pat** is used as a shrinking alternative.

`SIZED(`**Size**`,`**G**`)`	Maps the current generated test case size to **Size**, which can then be used by generator **G**. Generators can then create test values which grow in proportion to the number of executed test cases.
`LAZY(`**G**`)`	Returns generator **G**, but evaluates its body *lazily*.

Lazy evaluation can avoid infinite recursion or unnecessary computation. For example, in Listing 2.5, `gen_infinite()` would always attempt to evaluate both items in the parameter list of `oneof` (a function that randomly chooses an element from the given list), causing the generator to recur indefinitely. Using `?LAZY`, `gen_lazy` breaks the cycle by only performing a recursive call whenever the second branch is taken.

Listing 2.5: Eager vs lazy evaluation

```
1 gen_infinite() ->    % Eager
2   oneof([true, gen_infinite()]).
3
4 gen_lazy() ->        % Lazy
5   ?LAZY(oneof([true, gen_lazy()])).
```

Listing 2.5 also demonstrates the creation of a *recursively-defined generator*. Recursive generators tend to require the creation of custom shrinking rules [ACH08]. They can also make use of the `SIZED` macro in order to relate the depth of recursion to the test instance size.

2.5.4 QuickCheck Finite State Automata

To this point, generators were taken to produce single or composite values which would then be used within a property. Yet QuickCheck also allows the generation and verification of *call sequences* using *QuickCheck Finite State Automata*, as will be described in the following section.

Overview

QuickCheck Finite State Automata (QCFSA) [PA09] 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*. QCFSAs allow one to define transitions between conceptual system states in terms of implemented system functions. As an example, a QCFSA describing a light switch may represent the 'on' position as a simple abstract **on** state, without specifying the actual system variables that would lead to the light being on, as shown in Figure 2.5. The number of system states represented by a QCFSA's abstract states is directly influenced by the program's implementation, and will affect the *granularity* of the generated test cases. This will be discussed in greater detail in Section 6.4.

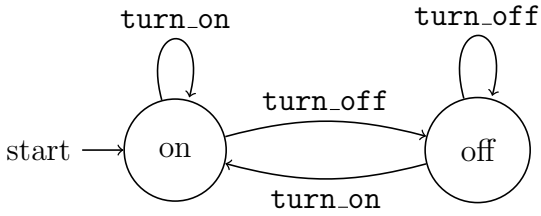

Figure 2.5: A simple two-state light switch model with toggle functions. Nodes represent abstract system states, and transitions correspond to implemented system functions.

The purpose of a QCFSA is to describe (possibly infinite) sequences of function calls and properties that should hold over each function's execution. For example, the model described in Figure 2.5 recognises an infinite sequence of `turn_on` and `turn_off` calls, that is, (`turn_on` | `turn_off`)*. QuickCheck uses automata as generators of program traces over which properties can be checked. Generation is performed by traversing the automaton over a finite number of arcs, choosing each next state randomly. One can stop certain traces from being generated by associating *preconditions* with arcs, which must hold for that transition to be taken. A generated trace may then be executed within a QuickCheck property.

QCFSAs can be used solely as generators, yet this approach may be rather limited in scope. Instead, QCFSAs allow one to specify properties using *postconditions*, which are functions that must hold when executing functions within a generated sequence. This increases the utility of QCFSA by allowing verification to be performed on intermediate value changes and conditions. For example, in the aforementioned light switch model, one could check that every toggle operation does indeed change the light switch status.

Structure of a QCFSA

A QCFSA contains constructs which can broadly be classified as concerning either *trace generation* or *verification*, and is implemented as an Erlang module that adheres to a defined interface which, as described in [PA09] and [Quv10], must specify:

- An *initial state* with name **S** and starting data **D**
  - **S** is specified as an atom returned from `initial_state()`
  - **D** is specified as a data structure returned from `initial_state_data()`

  QCFSAs can make use of a private data store, or *state data*, which persists over transitions. This allows an automaton to preserve state. The data can be of any arbitrary type, but is typically a record structure so as to allow multiple values to be stored. The state data can be accessed during a transition, and can only be modified once a transition is taken, as will be explained shortly.

  States can have an associated *state attribute*, changing the state name to a tuple of the form {**name**, **attribute**}. States with different attribute values are treated as separate states. Transitions can contain conditions on attribute values.

- A list of *transitions*. Each transition is described by implementing two callback functions, which describe
  - The transition from state **S** with state data **D** to **S′** while executing function **Mod:Fun** with parameters **Args**, given as

    $$\mathbf{S}(\mathbf{D}) \rightarrow [\{\mathbf{S}', \{\texttt{call}, \mathbf{Mod}, \mathbf{Fun}, \mathbf{Args}\}\}].$$

    S can have several outgoing transitions, which are defined by adding tuples for each transition in the list returned by the function, as above. State data **D** can be referenced within **Args**. Transition restrictions on state data cannot be imposed within this construct, and the choice of next state may only be influenced through the *precondition* construct.
  - The transformation which data undergoes when moving from state **S** with data **D** to **S′**. The rules are defined using the function

    $$\texttt{next\_state\_data}(\mathbf{S}, \mathbf{S}', \mathbf{D}, \mathbf{Result}, \mathbf{Call})$$

    where **Result** is the value returned by the implemented function associated with the transition and **Call** is the symbolic representation of the function's invocation.
- A list of *preconditions* defined as functions. The precondition on a transition from **S** to **S′** is defined as

  $$\texttt{precondition}(\mathbf{S}, \mathbf{S}', \mathbf{D}, \mathbf{Call})$$

  If the precondition evaluates to **false**, then the trace will not be generated for testing.
- A list of *postconditions* defined as functions. The postcondition on a transition from **S** to **S′** is checked once the transition is performed and its associated function is executed, and is defined as

  $$\texttt{postcondition}(\mathbf{S}, \mathbf{S}', \mathbf{D}, \mathbf{Call}, \mathbf{Result})$$

  If the postcondition evaluates to **false**, then the trace is considered to have failed and the relevant report will be generated.

Trace Generation and Execution

Before a trace is executed, it must first be generated. Generation and execution require two separate passes through the automaton.

Given a module $\mathcal{M}$ containing an implemented QCFSA, one may generate traces using the `commands(`$\mathcal{M}$`)` generator. A trace φ produced by this generator may be executed with live values using `run_commands(`$\mathcal{M}, \varphi$`)`, which returns the trace leading to the final state, the last successfully-executed transition and the reason for termination (**ok** on success). Listing 2.6 shows a simple test harness that generates and verifies traces using a given QCFSA.

Listing 2.6: A simple QCFSA generation and execution harness. `?FSM` is the name of the module containing the automaton.

```
1  prop_fsm() ->
2    ?FORALL(Cmds, commands(?FSM),
3      begin
4        {_History, _State, Reason} = run_commands(?FSM, Cmds),
5        case Reason of
6          ok -> true;
7          _  -> false
8        end
9      end).
10
11 main() ->
12   eqc:start(),
13   eqc:quickcheck(prop_fsm()),
14   eqc:stop().
```

The process of trace generation is carried out as follows. Starting from an initial state, QuickCheck identifies the set of eligible next states and chooses a transition that leads to one of these states. By default, the engine attempts to choose outgoing transitions with approximately equal probability. Alternatively, one can associate probabilities with transitions manually using the `weight()` function [Quv10].

If the precondition function associated with a transition returns **false**, then the transition is excluded from the trace and another transition is taken. In this way, preconditions serve to limit the set of traces which will be tested, typically so as to direct searches towards domains with an increased likelihood of failure (the consequences of restricting the set of traces that can be generated will be discussed in greater detail in Section 4.6.1). Additionally, a transition will not be chosen if its generation will lead to an exception and alternative paths exist [Quv10]. Once a valid transition is identified, the state data is updated by computing the given transition's associated next state data function, and the system moves to the next state. One should note that during generation, the `postcondition` function is never invoked. Traces are finite, and their length increases with the number of generated traces.

During the generation phase, QuickCheck uses *symbolic variables* instead of concrete variable values [Quv10]. Symbolic variables are placeholders for variables whose values will be determined during the execution phase, and are of the form {`var`, **N**}, where **N** is an integer that identifies that placeholder uniquely. Within a generated trace, the result of the execution of a transition's associated function will be represented by a symbolic variable. Consequently, the `next_state_data` function will only have access to a placeholder for the result, rather than a concrete result. For example, if a function's execution returns a record, then one cannot access the individual record's members from within the `next_state_data` function. This can limit the degree with which trace generation can be fine tuned in reaction to the system's behaviour, as one would not be able to alter a precondition's verdict based on the result of previously-executed functions.

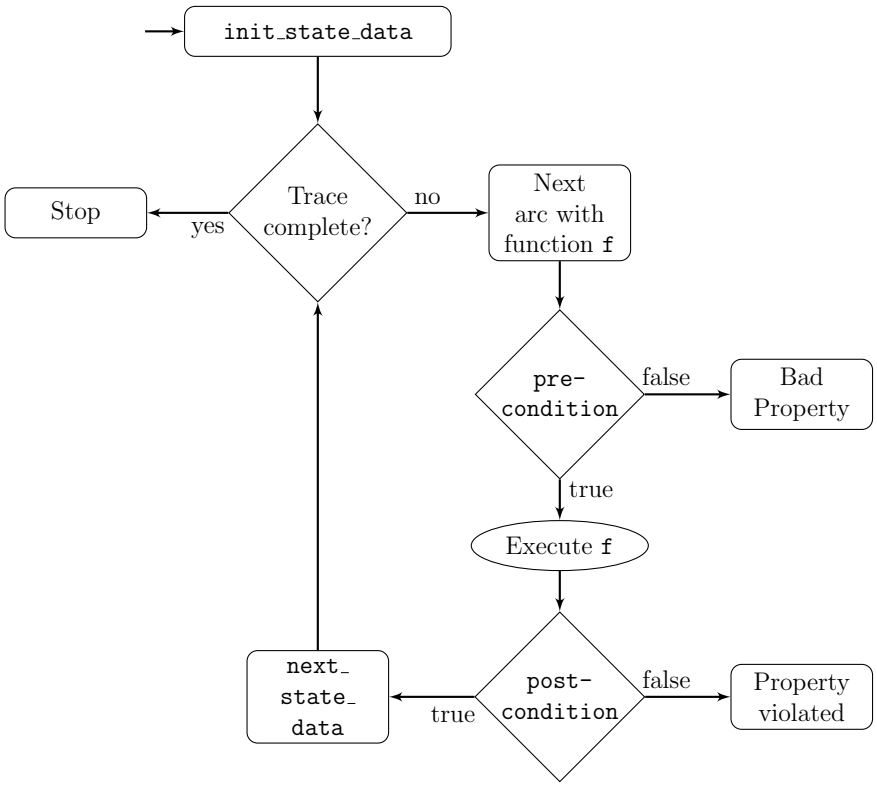

Figure 2.6: Flowchart showing the execution of a single, generated trace

Executing a trace is simply a matter of replacing the symbolic variables with the computed (live) values, and reporting an error on encountering a failed postcondition. During execution, a precondition must not fail, as this would imply that the automaton is being traversed inconsistently during generation and execution. Thus, pattern matching on symbolic variables should not be performed if it will result in different verdicts being reached during the separate passes.

Adding Conditions to the Light Switch Controller

Using the aforementioned light-switch model (Figure 2.5) and the definition of a QCFSA's structure, one can augment the automaton to also act as a classifier. Given that the property checks that the returned new state of the light's status with each operation must match the expected value, the QCFSA may be written as in Listing 2.7.

Listing 2.7: A two-state QCFSA that generates and tests light switch controller traces

```
1 initial_state() -> on.
2 initial_state_data() -> none.
3
4 on(_D) ->
5   [{on,   {call, light, turn_on,  []}},
6    {off,  {call, light, turn_off, []}}].
```

```
7  off(_D) ->
8    [{on,   {call, light, turn_on,  []}},
9     {off,  {call, light, turn_off, []}}].
10
11 precondition(_S, _S2, _D, _Call) ->
12   true.
13
14 postcondition(_, off, _, _, R) ->
15   R == off;
16 postcondition(_, on, _, _, R) ->
17   R == on.
18
19 next_state_data(_, _, _, _, _) ->
20   none.
```

Since the automaton admits all the possible interleavings of `turn_on` and `turn_off`, the automaton's states can be merged, with the light's current status being stored within the automaton's state data structure. Listing 2.8 defines such an automaton. As the QCFSA's states no longer encode the light's target status, one cannot verify the postcondition solely through the analysis of the next state. Instead, the postcondition inspects the **call** parameter directly and bases its verdict on the function being called.

Listing 2.8: An equivalent single-state QCFSA

```
1  initial_state() -> s.
2  initial_state_data() -> {light, on}.
3
4  s(_D) ->
5    [{s,  {call, light, turn_on,  []}},
6     {s,  {call, light, turn_off, []}}].
7
8  precondition(_S, _S2, {light, Status}, _Call) ->
9    true.
10
11 postcondition(_, s, _, {call, light, Op, _}, R) ->
12   case Op of
13     turn_on  -> R == on;
14     turn_off -> R == off
15   end.
16
17 next_state_data(_, _, _, R, _) ->
18   {light, R}.
```

Maintaining the light's state within the automaton's state data structure allows one to place restrictions on the automaton's behaviour. For example, one may desire to restrict verification to traces which do not call the same operation consecutively, that is, calls to the light switch controller that do not change the light's state should be ignored. Fulfilling

this requirement using the two-state automaton is a simple matter of removing transitions that loop back onto the same state. In the case of the single-state automaton, one would have to modify the precondition to filter out identical and consecutive traces. This is done by only accepting transitions that will toggle the currently-stored light state value.

Listing 2.9: Modified single-state QCFSA functions

```
1 precondition(_S, _S2, {light, Status}, {call, light, Op, _}) ->
2   ((Op == turn_on) and (Status == off)) or
3   ((Op == turn_off) and (Status == on)).
4
5 next_state_data(_, s, _, _, {call, light, Op, _}) ->
6   case Op of
7     turn_on  -> {light, on};
8     turn_off -> {light, off}
9   end.
```

Listing 2.9 details the modifications which have to be made to the single-state QCFSA so as to only generate and verify traces containing alternating sequences of operations. The `next_state_data` function has to be reimplemented so as to avoid using the function call's return value. This is because, as discussed earlier, a function call's result is replaced by a symbolic variable during the generation phase, thus disallowing a precondition's verdict from depending on its concrete value.

A Note on Determinism

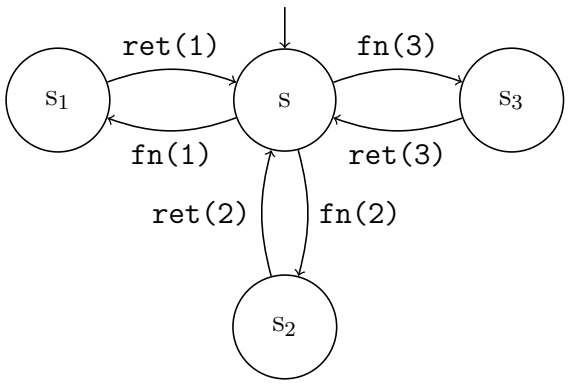

Figure 2.7: A valid, deterministic QCFSA. Every transition's precondition returns **true**. The functions `fn`:: $\mathbb{N} \rightarrow$ *true* and `ret`:: $\mathbb{N} \rightarrow$ *true* are simple functions.

QuickCheck automata must be deterministic, that is, a state can only have multiple outgoing arcs for the same function if at most one arc's precondition evaluates to true during traversal [Quv10]. Functions invoked with different arguments are regarded as separate transitions, and their preconditions can intersect. This is illustrated by the automaton in Figure 2.7, which, in spite of having multiple outgoing arcs for `fn`, is deterministic.

2.6 Applications

Historically, testing has been used as an evaluation technique. Testing can be used to demonstrate a system's non-compliance with the requirements as well as conformance to a specification. It can also be used to determine a system's capacity to withstand stressful loads, as well as to measure other system attributes such as performance.

An alternate use for testing is as an implementation aid, as is done in *test-driven development*, which streamlines the testing process with that of development. This approach is exemplified by the *Mock Objects* approach [MFC01] to unit testing, which centres on the use of dummy implementations for modelling domain code. Mock Object-based testing is described by [FMPW04] as a technique for identifying system types based on the roles that the objects play. Before writing the actual system implementation, programmers first create *programmer tests* for a unit. This is primarily a design activity that causes the programmer to build functionality based on its use rather than its implementation. Objects are treated as services characterised by their outgoing interfaces, which describe what the service consumes and provides. [FMPW04] reports that this encourages locality between objects and message passing, which in turn facilitates scalability and aids in assessing test coverage. Another effect of basing implementations on object interfaces is that it encourages *need-driven development*, where only functionality related to the object's role is implemented.

2.7 Conclusion

Testing can be performed on a system at different levels of abstraction, yet fundamentally, testing always involves the checking of a system against some criteria, or property. While the act of checking a property can be automated to a very large degree, the process of creating properties often requires manual intervention, despite the existence of several automatic inference techniques. The issue stems from the fact that the notion of correctness is often based on the semantic behaviour of a program and is subject to what is required from it, whereas automated property generation techniques rely on resident predetermined assumptions on what constitutes correct behaviour. For example, a tool which automatically performs fuzz testing (passing large random values as inputs to a system in an attempt to induce failure) may assume that a crash corresponds to a program malfunction, yet it cannot differentiate between valid and invalid behaviours which do not precipitate such a failure. Consequently, inferred properties may be unsound or incomplete. Simply inferring a specification or property does not in itself testify to a system's correctness or otherwise, as errors in the program would be included in the properties. Thus, the automated property and test suite generators serve primarily as aids in the verification process and must be used in conjunction with other techniques.

The true merit of automation lies in its consistency and the ease with which it can be adapted to accommodate larger volumes of tests, thereby increasing the probability of errors being caught. Also, while properties cannot always be generated automatically, other parts of the testing process, such as instrumentation and the logging of results, can

often be fully automated. Three frameworks for test automation in Erlang were examined, with particular emphasis on QuickCheck. QuickCheck differentiates itself from the other frameworks in that it focuses on the automatic generation of test cases, allowing arbitrarily large suites to be produced and tested.

Processing large test suites comes at the cost of performance. Furthermore, while the odds of finding an existing error should increase as the number of verified tests grows, a program's state space may be enormous, requiring far too many tests to be executed and limiting a suite's effectiveness. The next chapter introduces *runtime verification*, which attempts to address these issues by delaying verification until execution and only checking the correctness of the active system trace.

Chapter 3

Runtime Verification

3.1 Introduction

Testing is rarely comprehensive, and full program coverage is seldom achieved. Certain program behaviours may thus remain untested, allowing faults to propagate into the deployed system. Rather than considering every possible program execution path, *runtime verification* restricts itself to the analysis of the active thread of execution, that is, it only considers program traversals that are pursued at *runtime*. Runtime verification thus checks that the behaviour exhibited by a deployed system complies with a property, detecting deviations from its specification [CM05]. Complete specifications of programs are not always available, and properties are typically restricted to describing a subset of the system requirements that is considered critical.

Runtime verification can be used to react to system failures as soon as they occur. It can also be used to monitor contract enforcement, ensuring that all parties within a negotiation fulfil their contractual obligations and that penalties are paid on contract violation [LS09]. Runtime monitoring simplifies the verification of complex interaction and communication scenarios, as they would not have to be recreated within a test bed. This can also help in making verification more robust by anticipating and catering for failures that have not been observed at the testing state.

Properties can be expressed in some form of *temporal logic* such as LTL [LS09] or using *automata* such as DATEs [Col08]. The representation's expressivity will affect what can be monitored, with complex properties requiring more computational resources to verify.

The following chapter discusses the theoretical underpinnings of runtime verification, describing the differences between *online* and *offline* verification and the use of monitors. This is followed by an introduction to the DATE property constructs, which are elaborated upon further in subsequent chapters. The chapter closes with a comparison of runtime verification with other techniques, as well as its pitfalls and possible ways to mitigate their effects.

3.1.1 Offline vs Online Verification

Runtime verification can be performed either on a trace as it is being built (*online* verification), or once execution has terminated and the trace has been recorded (*offline* verification) [Col08]. Online verification involves running the verification engine in parallel with the target program, with the monitor receiving events as they occur. This allows the system to correct itself at runtime whenever it detects that it has entered a bad state. One drawback of online verification is that the monitor only has access to a partial execution, rather than the full run. This may hinder the verification of certain liveness properties, with some properties becoming unmonitorable, for reasons described in Section 3.2. Another problem brought about by online verification is that the monitor consumes the system's resources, limiting the complexity of the properties that can be checked. This makes runtime verification intrusive, and the presence of a verifier could alter the program's temporal characteristics and itself induce violations which would otherwise not occur. The impact of online verification can be reduced either by adding special hardware [LS09], or by decreasing the resource demands of the verifier, as described in Section 3.4.

Offline verification is performed on recorded runs of previous program executions. It can help in finding errors during testing, and minimizes the verification overheads to those introduced by the instrumentation code that records the executions. Since the trace is known in its entirety, future temporal properties can be verified, provided that the instrumentation code records all of the relevant data describing the event (such as a timestamp or the trace's event ordering). The drawback of offline verification is that it does not allow compensatory actions to be performed dynamically during execution, limiting its application to live systems [LS09].

3.1.2 Monitors

Property violations are detected using a *monitor*, which analyses traces and reports any mismatches between the observed and expected system behaviour. Traces are obtained though instrumentation, operating either at the code or the binary level. Instrumentation should be performed automatically so as to maintain consistency and reduce errors. If a program cannot be instrumented directly, then one may have to modify its execution environment to output the necessary event information. Instrumenting at the system level may also widen the extent of what can be verified by offering a global view of the system, allowing a single monitor access to events emanating from all of the executing subsystems. [CM05]

If $[\![\varphi]\!]$ is the set of negative traces covered by the property φ, then the problem addressed by runtime verification can be seen as determining whether a given execution w is an element of $[\![\varphi]\!]$. Thus, a monitor $\mathcal{M}_\varphi$ derived from φ takes a finite trace as input and returns a *verdict* based on the observed trace's membership in $[\![\varphi]\!]$, where a verdict is a value in some truth domain [LS09]. For example, with a two-valued truth domain, a monitor could return a verdict $v \in \{\text{true}, \text{false}\}$. In online monitoring, executions are examined incrementally, and a monitor updates its verdict with every new event. Monitors should

avoid having to access an execution's history, as this would slow down verification, with delays becoming progressively larger as the length of the execution grows. [LS09] states that in order to guarantee that a monitor does not evaluate a verdict either prematurely or too far along an execution, it must fulfil the principles of:

1. **impartiality**, where a verdict is not set to a final value as long as there still exists a path that can be taken which will cause the monitor to reach a different conclusion. This requires at least three verdict values, one of which would signify an *inconclusive* verdict.

2. **anticipation**, where all paths from the point under consideration onwards will result in the same verdict being evaluated. This implies that a verdict must be produced as soon as it is possible to do so.

In theory, runtime verification is only concerned with the detection of property violations [CM05]. Notwithstanding this, runtime verifiers can also incorporate mechanisms for taking actions on certain event triggers, such as on detecting a bad state. Monitors which incorporate these abilities can be structured into different layers or components, based on their concerns. For example, one component could harvest events as they occur within the system and use them to construct a finite trace, which would then be passed to a *verifier* that checks that it adheres to the given specification. Finally, defined actions can be triggered when specified conditions are met, and reparatory routines could be invoked on detecting entry into a bad state [Col08].

[LS09] describes three approaches to reacting to system conditions, as follows:

- **Fault Detection, Identification and Recovery** (FDIR) regards system failures as manifestations of faults. On detecting a fault, the system must be able to identify the point at which the system failed and reconfigure the system accordingly.

- **Runtime Reflection** (RR) utilises a four-tiered architecture, defined as the *logging*, *monitoring*, *diagnosis* and *mitigation* layers. The logging layer harvests system events and passes them on to one or more monitors, which inspect the event stream as it is being built and search for failures. The diagnosis layer waits for failures to be reported by the monitoring layer, identifying their point of origin and reporting them to the mitigation layer, which reconfigures the system so as to correct itself. If a failure cannot be mitigated, then an error report is generated, which can serve to direct manual intervention.

- **Monitor-Oriented Programming** (MOP) is a methodology that allows a programmer to specify properties from which monitors are automatically generated and integrated into the system. One must also specify what actions must be taken when a property is violated. MOP allows the development of introspective systems which can alter their own behaviour. MOP differs from RR in that the former is a programming methodology which is tightly coupled to a programming language, whereas the latter is an architectural pattern that defines a system structure. Another difference is that RR classifies failure types using a diagnosis layer built over the monitoring layer.

3.2 Defining Properties

Properties serve as the basis for determining whether a system is in a compliant or non-compliant state by specifying its required behaviour. As described in [CPS08], runtime monitors are often derived directly from properties. Properties may also be used to guide the automatic instrumentation of programs. Certain properties can be inferred automatically by observing and synthesising certain aspects of a program's behaviour (*learned properties*) [CM05], and can facilitate regression testing by identifying unanticipated changes in functionality. Other properties can be derived algorithmically through the use of *invariant detection* or *error pattern analysis*, identifying standard failure sequences such as deadlocks and data races, as described in Section 2.3. Often, properties are simply written manually.

When defining properties, it is sometimes easier to characterise a program's correct behaviour by defining its invalid behaviour, that is, one can specify a system's bad states and assume that the property holds as long as the system is not in one of those states. Properties can be written in several ways and notations, each having different degrees of expressivity, with some allowing automatic translations from one to the other. For the purpose of runtime verification, property logics should typically be able to express *temporal* and *contextual* constraints on traces, and may also have the ability to handle *exceptions* [CPS08]. Temporal constraints refer specifically to the ability to express event *consequentiality*, giving events an ordering, and the ability to specify *real-time constraints* such as upperbounds on execution time. A property's contextual aspects refer to its ability to set the scope within which the property will be monitored, such as global invariance or conditions on a function's return value.

Properties can be broadly categorised as being either *liveness* or *safety* properties. [AS85] defines safety properties informally as properties which ascertain that bad things will not happen. Specifically, φ is a safety property if it guards against a discrete violation which can occur at an identifiable point within an execution. Liveness properties are consequently defined as those which ensure that a good thing will happen in time during an execution. An execution is *live* if there exists a continuation of the trace such that the property will be satisfied. Thus, a liveness property is one for which every execution is live.

Several temporal property logics exist. For example, *Linear Temporal Logic* (LTL) [Pnu77] allows the creation of properties relating events and their ordering. LTL can be used to reason about infinite traces, and supports the definition of unmonitorable properties which rely on knowledge of future events and their occurrence. Alternatives designed to address monitorability issues exist, with extensions such as LTL_3 [BLS07] making use of three-valued logic and introducing the notion of *inconclusive* traces, and other logics opting to restrict verification to past events. An alternative to temporal logics is to define properties using *automata*, which [Col08] reports as being simpler and more intuitive to use. Security properties tend to be easier to express using automata rather than logic-based formalisms [AN08]. In addition, certain temporal logics can be translated into automata, allowing them to be used with tools accepting automata as inputs. The following section describes *Dynamic Automata with Timers and Events*, an automaton-

based logic designed for runtime verification within the LARVA [Col08] framework.

3.2.1 Dynamic Automata With Timers and Events

Dynamic Automata With Timers and Events (DATEs) are automata used for describing temporal properties for runtime verification, as detailed in [CPS08]. They serve as the input property language from which monitors used within the Java-based LARVA[1] runtime verification tool are derived. DATEs facilitate *event-based* runtime monitoring, where an event can be a visible *system action* such as the invocation of a method or exception handler. Other events, such as timer and channel synchronisation events, are also supported, as well as a combination of the aforementioned through the use of *composite events*, as will be seen shortly.

The following section formally defines the DATE logic model based on the description provided in [Col08]. This exposition is built upon by subsequent chapters, which describe the automaton's behaviour and operation in greater detail.

Events

Given a set of events **systemevent** generated by the system, a set of timer variables **timer** and a set of channels **channel**, a *basic event* is defined as

event ::= systemevent
| channel? (channel synchronisation)
| timer @ δ (timeout after δ)

Basic events can be combined into more complex event expressions known as *composite events*, which define the *choice* and *complement* operators on basic events. A basic event **x** firing a composite event expression **e** is denoted as $\mathbf{x} \models \mathbf{e}$, and is defined as follows:

$x \models e$ if
$x = e$
$\vee\ e = e_1 + e_2 \wedge ((x \models e_1 \wedge x \not\models e_2) \vee (x \not\models e_1 \wedge x \models e_2))$
$\vee\ x \in systemevent \wedge e = \overline{e_1} \wedge x \not\models e_1$

The firing of events can be lifted for sets of basic events **X** as $\mathbf{X} \models \mathbf{e}$, where an element of **X** fires the event **e**.

DATE timers are described through their configuration. A *timer configuration* $\mathcal{CT}$ is a function which takes a timer and returns its value and state, that is,

$$\mathcal{CT} :: Timer \rightarrow Value \times State$$

[1] LARVA, and an equivalent Erlang-based runtime monitoring framework, are described in further detail in Chapter 5.

where
Value $\in \mathbb{R}_0^+$
State $\in$ { running, paused }

Timer operations that reset, pause and resume timers are defined as functions which take a configuration and return an updated version, modifying either the state or value. *Reset* sets the timer value to zero, while *pause* and *resume* toggle the timer state. A *timer action* $\mathcal{TA}$ is subsequently defined as the functional composition of a finite number of timer operations.

Symbolic Timed Automata

A *symbolic timed automaton* $\mathcal{M}$ over a system with states of type Θ is described by a tuple $\langle Q, q_0, \rightarrow, B \rangle$, where

Q is a set of system states
$\mathbf{q_0} \in \mathbf{Q}$ is the initial state
$\rightarrow$ is the set of transitions
B is the set of bad states

The automaton moves from state to state, choosing the highest-priority transition whose guard condition is satisfied. Transitions can in turn fire other transitions as they are traversed. The transition relation between a current automaton state to the next state is defined as:

$$Q \times \underbrace{event}_{\text{Triggering expression}} \times \overbrace{(\Theta \times \mathcal{CT} \rightarrow Bool)}^{\text{Condition on timers \& state}} \times \underbrace{\mathcal{TA}}_{\text{Perform timer action}} \times \overbrace{2^{channel}}^{\text{Signal event over channels}} \times \underbrace{(\Theta \rightarrow \Theta)}_{\text{State changing code}} \times Q$$

Moving from one configuration of a symbolic automaton $\mathcal{M}$ to another configuration, where X is a set of system scheduled events and $\theta \in \Theta$, whilst performing timer update t' with timer configuration T, is denoted as

$$(X, \theta, q) \Rightarrow_{t'}^{T} (X, \theta', q')$$

Such a step can be performed provided that q is not a bad state and that a transition to q' with a satisfied guard condition exists. Otherwise, the event is consumed without altering the timer configurations or moving to a different state. Transitions between configurations can be extended for vectors of communicating automata with shared timers, provided that each automaton can move.

Defining DATEs

DATEs are based on the use of symbolic timed automata, and allow new automata to be spawned during traversal. Formally, a DATE $\mathcal{M}$ is a pair $\langle \overline{M}_0, v \rangle$, where

$\overline{\mathbf{M}}_\mathbf{0}$ is the initial set of automata

$\mathbf{v}$ is a set of *automaton constructors*

An automaton constructor has the form

$$\overbrace{event}^{\text{Trigger}} \times \overbrace{(\Theta \times \mathcal{CT} \to Bool)}^{\text{Timer Condition}} \times \overbrace{(\Theta \times \mathcal{CT} \to Automaton)}^{\text{Automata Creation Function}}$$

For each $(e, c, a) \in v$ where e has been detected, an automaton is created using function a provided that the state and timer conditions satisfy c. Thus, the set of triggered automata with timer configuration T, events X and system state θ is denoted by $tr(T, X, \theta)$ and is defined as

$$tr(T, X, \theta) \stackrel{\text{def}}{=} \{a(\theta, T) \mid (e, c, a) \in v, X \models e, c(\theta, T)\}$$

A DATE's configuration is given through its timer and system state as well as the state of all executing automata. A full step from configuration $(T, \theta, \overline{q})$ to $(T', \theta', \overline{q'})$ upon receiving a set of system actions X is given as

$(T, \theta, \overline{q}) \mapsto_X (T, \theta', \overline{q'})$ if $\exists n$ such that
$(X_0, \theta_0, \overline{q}_0) \mapsto^T_{t_1} (X_1, \theta_1, \overline{q}_1) \mapsto^T_{t_2} \ldots (X_n, \theta_n, \overline{q}_n) \mapsto^T_{t_{n+1}} (\emptyset, \theta_{n+1}, \overline{q}_{n+1})$
where
$\overline{q} = \overline{q}_0$, $\theta = \theta_0$, $\theta' = \theta_{n+1}$
$T' = (t_{n+1} \circ t_n \circ \ldots t_1)(T)$ (accumulated timer actions)
q' is updated accordingly with each triggered action

A transition is an *accepting full-step* if the system does not go through any intermediate bad states. To ensure the absence of livelock, the system uses *loop-free automata*, which avoid mutual dependencies by not allowing channels to loop onto themselves.

Generating Monitors

Once written, DATEs can be implemented directly and automatically into LARVA as runtime monitors. As described in [Col08], *Aspect-Oriented Programming* techniques are used in order to interleave verification code into the target system's bytecode. This allows instrumentation and flow control code to be inserted, updating the DATE's state and supplying it with event information. It also allows the triggering of actions on receiving an event, which can be used to react to a system behaviour by executing additional or compensatory procedures.

As will be seen in subsequent chapters, each element of the DATE model has an implemented counterpart [CPS08]. *System events* correspond to method calls, exception handlers and object initialisation routines within the system under test. *Actions* are implemented as blocks of code that are invoked when their corresponding transition is taken. Within the implementation, actions can directly alter the state of the system under test.

Java DATE properties may also be bound to specific *contexts*, whereby a property is bound to a single instance or an entire class of objects, with separate monitors being spawned for each context. This allows properties to be quantified over sets of objects. For example, one can bind a property to a class implementing a buffer. The monitoring framework would then spawn a new monitor for every created instance of that class, keeping track of the objects as they evolve within the system. Context levels can also be nested, and monitors can be tied to program elements through pattern matching with the program's source code. DATEs can also be used to verify *invariance* by placing a condition on every event's transition. *Clocks* and real-time constraints are implemented using Java's `wait()` operation on threads. [Col08]

3.3 Comparison With Other Verification Techniques

3.3.1 Runtime Verification and Model Checking

Model checking is typically performed by deriving a *model* of the system being verified and analysing it. Model checking analyses a system statically, meaning that verification will not interfere with a system's execution, which in turn allows complex properties to be checked. Properties can also be checked for all paths, rather than single executions, as is the case with runtime verification. Given model $\mathcal{M}$ and correctness property φ, [LS09] defines model checking as the act of finding whether all computations of M satisfy φ. One can also approach the problem by constructing an automaton $\mathcal{M}_{\neg\varphi}$ that accepts all runs violating φ and comparing it with $\mathcal{M}$ to see whether it contains any bad executions.

Model checking requires the availability of or the ability to create an accurate model of the system, which may not always be possible. If such a model exists, and its state space is bounded, then one can apply bounded model checking techniques. For example, one can analyse paths starting from an end node and work backwards, which cannot be done using runtime verification due to the absence of a complete execution. It also allows the analysis of infinite traces, which are known to contain loops if the number of states is finite. When verifying an infinite trace, showing that a property holds up to state $\mathbf{N}+1$ automatically proves that it holds up to state $\mathbf{N}$. This cannot be done using runtime verification, as the observed events do not fully describe the system's state, making it impossible to infer loops from repeated states. [LS09]

Model checking has several potential drawbacks, foremost of which is the fact that it does not always scale well with a system's complexity. Programs may contain many states as a result of a combinatorial explosion, rendering the checking of an entire model intractable. Additionally, when performing model checking, one must assume that the

model mirrors the target system properly, and that no errors were introduced during synthesis. Similarly, a model may fail to adequately capture the effects of the environment within which the system will execute, in contrast to runtime verification. [CM05]

3.3.2 Runtime Verification and Testing

Neither runtime verification nor testing provide full coverage. When testing, a test suite consisting of a finite set of input and output sequences is fed into the system, ensuring that the emitted results comply with those defined within the test suite. In *oracle-based* testing [LS09], a test suite only consists of input sequences. An oracle is implemented and included with the system under test, checking generated outputs and categorising test results. This is similar to runtime verification using monitors, except that in runtime verification, one does not typically define a monitor directly, rather it is derived automatically from a high-level specification. In addition, runtime verification does not require that a set of test inputs be provided. Unlike testing via test cases, runtime verification can be carried out indefinitely on a live system after it is deployed.

Runtime verification tends to be favoured when environmental effects must be factored in. It can also complement the checking of safety properties throughout system deployment, especially when the system under test is critical and must not fail. Testing tends to be used when performance is an issue and the overheads incurred by runtime verification cannot be afforded. [CM05] states that testing can be composed of a combination of runtime verification and testing through test cases.

3.4 Issues With Runtime Verification

While the benefits of extending verification throughout a system's lifetime are considerable, runtime verification is not free from drawbacks. Runtime monitors typically identify a bad state once it has been reached, without anticipating failures. While this may be acceptable in a test setting, it may be insufficient in a live scenario, as not all errors are reversible. Apart from the possible difficulty in creating a set of comprehensive properties, as well as the fact that certain temporal properties cannot be monitored, there is the more pragmatic problem of *performance*. Runtime verification incurs overheads which are not always acceptable on live systems, and can itself cause temporal properties to fail, even though they would hold in the absence of verification. On the other hand, excluding runtime verification from the target system would remove the benefits of checking properties at runtime as well as decrease the amount of debugging information available should a failure occour.

Several approaches to minimising the effects of verification can be taken. [BHL+07] states that *partitioning* the overheads in space, time, or both, will reduce the overall impact of verification. Verification nodes are tasked with monitoring a subset of the system's events through partial instrumentation routines known as *probes*. Spatial partitioning involves spreading probes onto several nodes, with each node performing a reduced amount

of work and reporting results to a centralised server. This is an imperfect solution, as certain nodes may be bound to verification hot spots, such as when monitoring statements within tight loops, leading to an imbalanced load distribution.

Temporal partitioning performs time slicing on the execution of probes, turning instrumentation on and off periodically. This could result in important events being missed, yet [BHL+07] argues that a missed violation will still be caught given sufficient executions, and that it is only truly crucial that the system never reports *false positives*. The system proposed uses *tracematches*, which are traces of events and associated triggered actions that must be verified and performed on detection. Progress within a tracematch is tracked using finite state machines, with property constraints (*shadows*) expressed in Disjunctive Normal Form and checked on each state. Code to instrument and update shadows is interleaved into the system. The system must then identify the *skip-shadows* which must remain enabled in order to avoid false positives, discarding skip-shadows which originate from states with empty constraints.

Other techniques for reducing the computational costs of runtime verification include *cooperative bug isolation*, *residual test coverage monitoring* and *shadow processing* [BHL+07]. Cooperative bug isolation uses random sampling of a large number of program executions in order to gather information regarding its behaviour. Residual test coverage monitoring instruments the entire program, adding probes in locations which are not covered by the testing criteria. Locations that are very active are checked with decreasing frequency as they are invoked, reducing the amount of computational resources consumed. Shadow processing is a technique whereby runtime verification is offloaded onto idle coprocessors in a multi-processor system.

3.5 Conclusion

This chapter has detailed the core concepts of runtime verification, beginning with a definition of the relevant terminology and describing its characteristics, as well as defining the DATE automaton logic. Runtime verification aims to address the shortcomings of testing, effectively bypassing concerns related to coverage by ensuring that paths taken during a program's execution are verified. It also verifies the system within its true executing environment, automatically factoring in the effects of external interactions and changing system topologies, which could otherwise be hard to model.

Runtime monitoring simplifies the checking of contract compliance, as the task of verifying whether a contract complies with a user policy through equality is undecidable if both are characterised using a context-free grammar [LQS08]. The alternative would be to limit the expressive power of the contract languages. Hence, contracts are often translated into automata that represent whether the system is in a compliant or a contractually-incorrect state, which are then verified as properties. Merely detecting a failure is not enough, and one must specify what course of action should be taken when a property is violated. For example, a roll-back strategy could be specified using a *rescue clause* [Mey92] which defines a service's alternate behaviour should a failure arise.

Although runtime verification can be deployed successfully in several scenarios, there are some caveats. In the case of online verification, the computational effort of verification is shifted from the testing phase to the deployment phase, consuming resources during the system's execution. Runtime verification can impose unacceptable overheads, and steps to mitigate its impact on performance may have to be taken. These measures generally focus either on improving monitoring efficiency or offloading computations onto other processors. Verifying services at runtime may not always scale well, as the amount of context that must be saved for each service will increase as the number of communicating agents and dependencies grows. [LPSS09] addresses this issue by compressing the description of all contractually correct behaviours into symbolic representations such as timed automata, which are continuously verified and updated from the system's event input stream. Another issue inherent to online verification is that it examines partial executions, which may not be sufficient for verifying a given property. Consequently, without some mechanism for anticipating future events, runtime verification will only identify a property violation once it occurs. While in some cases one can perform compensatory actions to exit a bad state, not all failures are reversible.

In summary, testing is useful in that it can catch errors *in vitro*, while runtime verification extends the process to deployment. To verify a system with respect to a given property using both methods, one may have to express the same property (or an approximation of it) using two different notations and perform each verification process independently based on the techniques used. The next chapter attempts to unify both approaches by describing an automated method of transforming testing properties into runtime verification properties. This would grant the analyser the choice of employing either approach without necessitating a rewrite of the property of interest.

Chapter 4

From QCFSAs to DATEs

4.1 Introduction

Testing and runtime verification often use different types of properties as inputs, with constructs tailored for the technique being employed. Given the significant investment required in writing properties, there is much to gain from being able to automatically retarget properties written for one technique to another.

The following chapter investigates the automatic translation of *QuickCheck Finite State Automata* (QCFSA) into *Dynamic Automata with Timers and Events* (DATEs), which are then used to create runtime verification monitors. A QCFSA embodies a property described in terms of valid and invalid function call sequences and behaviours, and intertwines the system's model with a verification oracle. It is thus conceivable that a QCFSA's concerns could be separated into those of a generator and a classifier. To extend QCFSAs to runtime verification, one would forego the generation of traces and directly classify traces captured from the instrumented system under test. The primary difficulty lies in separating the aspects of the QCFSA correctly and in preserving the property's original semantics within the synthesised monitor, ensuring that the translated property remains sound, and that only verdicts over traces which could be generated will be produced.

The chapter begins by presenting a formal model of QCFSAs and DATEs as used in Erlang. The notion of *negative* and *testable* traces for QCFSAs is introduced, the former being system traces which should be categorised as being invalid, whereas the latter are traces on which a property can produce a verdict. A construction for transforming QCFSAs into DATEs is then presented, along with a series of proofs that assert that a generated runtime monitor's set of negative traces is equal to that of the original property.

4.2 Formalising QCFSAs

In Section 2.5.4, QuickCheck Finite State Automata were presented in practical terms through a discussion based primarily on their implementation. The following section presents a formal description of a simplified model of QCFSAs. The model is simplified in that it abstracts certain implementation details, as follows:

- Pre- and postconditions are modelled as sets of valid system states. Compliance with a condition is transformed into a membership problem, that is, the system will satisfy a condition if its current state is in the condition's valid state set.

  When writing QCFSAs, one would normally characterise the conditions using total functions (`precondition` and `postcondition`) rather than through the explicit enumeration of valid system states. It is assumed that the act of checking for membership, and consequently the execution of the condition's associated function, is an operation which is free from *side-effects*. While not a direct requirement of the model itself, the translations detailed in Section 4.5 assume that a condition can be checked repeatedly without changing the system's state. This restriction could in theory be lifted, provided that the implemented monitor were to cache results so as to only check a transition's conditions once per transition.

  The use of system states serves to abstract away the application of `pre-` and `postcondition` functions without compromising the expressivity of automata. For example, given an implementation of a QCFSA, one could define the set of valid precondition states for a transition e going from state S to S' as $\{Q \mid Q \in 2^{\Theta} \wedge D \in Q \wedge \mathit{precondition}(S, S', e, D)\}$, where Θ is the set of system states.

- Arguments within a transition's function call are omitted. Instead, a *global system state* containing the entire application and monitor's state is used, with operations and transitions accessing and performing transformations on it.

- The existence of a `run()` function which returns the new global system state produced on executing a function is assumed. This is necessary as a postcondition will depend on the updated state. The value of `run()` is computed by the QuickCheck engine as an internal step during a QCFSA transition, and thus can be seen as a tap into the intermediate computation.

- Certain auxiliary functions, namely the ability to weight transitions, will not be included in this exposition.

In the context of QCFSAs, the term *event* refers to a symbolic call corresponding to an implemented function in the system under test. The automata being transformed are assumed to be *deterministic*, as described in Section 2.5.4.

4.2.1 The QCFSA Model

A QCFSA $\mathcal{Q}$ is described by a tuple $\langle Q, q_0, \Sigma, \theta_0, run, \rightarrow \rangle$, where

$\mathbf{Q}$ is a set of states

$\mathbf{q_0} \in \mathbf{Q}$ is the initial state

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

$\theta_\mathbf{0}$ is the initial global system state containing all of the program's state variables, including $\mathcal{Q}$'s local state data

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

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

Θ represents a *system state*. The transition relation $\to$ serves to relate 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. 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 the state following the event is an element of the postcondition set.

Using this definition of a QCFSA's structure, the two-state light switch model described in Section 2.5.4 can be formalised as $\langle Q, \texttt{on}, E, \{\texttt{none}\}, run, \Gamma \rangle$, where

$$
\begin{aligned}
\mathbf{Q} &= \{off, on\} \\
\mathbf{E} &= \{e_{on}, e_{off}\}, \\
\mathbf{run} &= \lambda e, t \cdot \\
&\qquad | \ (e = e_{on}) \to \{on\} \\
&\qquad | \ (e = e_{off}) \to \{off\}
\end{aligned}
$$

$$
\begin{aligned}
\mathbf{\Gamma} &= \{(S, (pre, e, post, \lambda t \cdot t), S') \mid \\
&\qquad pre \in 2^\Theta, \theta \in 2^\Theta, S \in Q, S' \in Q, e \in E, \\
&\qquad post = run(e, \theta) \wedge \{S'\} = post\}
\end{aligned}
$$

Definition 4.2.1.1. *Moving from state* $\mathbf{q}$ *to* $\mathbf{q'}$ *on event* $\mathbf{a}$ *with system state sets* **pre** *and* **post** *and executing state-changing action* α *is denoted by*

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

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

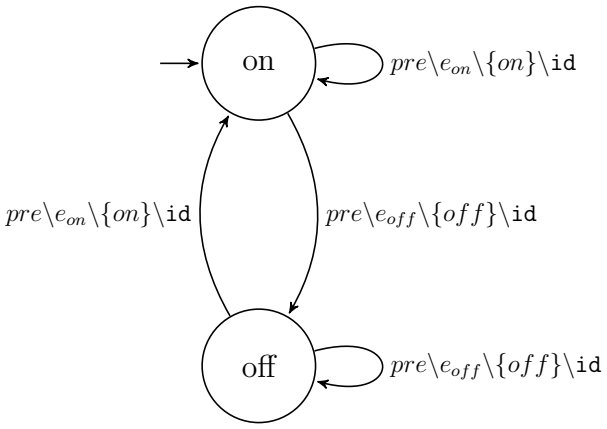

Figure 4.1: Formalised QCFSA for the two-state light switch model, where $pre = 2^Q$ and $\texttt{id} = \lambda x \cdot x$. Transitions are labelled as $\langle preconditions \backslash event \backslash postconditions \backslash action \rangle$ tuples.

4.2.2 Modelling Determinism

For the model to conform with the definition of determinism stated in Section 2.5.4, the following condition must hold; 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,

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

While the current version of QCFSA must, by definition, be deterministic, the model could potentially be modified to support *non-deterministic* QuickCheck Finite State Automata. Translating such an automaton into a runtime monitor would require that the latter be determinised in some manner, or the verification engine would have to be modified to cater for non-determinism.

4.2.3 Configurations

The *configuration* of a QCFSA consists of a pair (q, θ), where q is the current state and θ is the global system state at that point in the execution.

Single Events

Definition 4.2.3.1. *Performing a step from a configuration (q, θ) to a valid configuration (q', θ') on executing an event* **a** *is denoted by $(q, \theta) \overset{a}{\Rightarrow} (q', \theta')$, and is possible if*

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

A transition leads to a bad configuration if its postcondition is violated once an event's associated function has executed. As described earlier, a postcondition is violated if the new state is not part of the valid system state set.

Definition 4.2.3.2. *Moving to a bad configuration under event* **a** *is defined as*

$$(q, \theta) \overset{a}{\Mapsto} \otimes \overset{def}{=} \exists pre, post, \alpha, q' \cdot q \xrightarrow[\alpha]{\{\text{pre}\}\ a\ \{\text{post}\}} q' \wedge \theta \in pre \wedge run(\alpha, \theta) \notin post$$

Event Sequences

Trivially, a configuration does not change if no events are received. Thus, if ε is the null event,

$$(q, \theta) \overset{\varepsilon}{\Rightarrow} (q', \theta') \overset{\text{def}}{=} q = q' \wedge \theta = \theta'$$

A system cannot move to a bad configuration on an empty event, that is, $(q, \theta) \overset{\varepsilon}{\not\Mapsto} \otimes$.

Definition 4.2.3.3. *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*

$$(q, \theta) \overset{a:s}{\Rightarrow} (q', \theta') \overset{def}{=} \exists q'', \theta'' \cdot (q, \theta) \overset{a}{\Rightarrow} (q'', \theta'') \wedge (q'', \theta'') \overset{s}{\Rightarrow} (q', \theta')$$

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. Formally, $\overset{a_1}{\Rightarrow}; \overset{a_2}{\Rightarrow} = \overset{a_1 a_2}{\Rightarrow}$.

Definition 4.2.3.4. *Moving to a bad configuration given a sequence of events is defined as*

$$(q, \theta) \overset{s +\!\!+ \langle a \rangle}{\Longmapsto} \otimes \overset{def}{=} \exists q', \theta' \cdot (q, \theta) \overset{s}{\Rightarrow} (q', \theta') \overset{a}{\Mapsto} \otimes$$

with the last step leading to a bad configuration.

4.2.4 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* $T(\mathcal{Q})$.

Definition 4.2.4.1. *The set of negative traces $N(\mathcal{Q})$ is the set of traces which, starting from the automaton's initial configuration, lead to a bad configuration. Thus,*

$$N(\mathcal{Q}) \stackrel{def}{=} \{w \colon \Sigma^* \mid (q_0, \theta_0) \overset{w}{\Rightarrow} \otimes\}$$

Definition 4.2.4.2. *The set of testable traces $T(\mathcal{Q})$ refers to the set of traces with which the automaton is concerned, that is, the traces which it tests. Thus,*

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

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 $w \notin N(Q)$. Similarly, a program is correct with respect to $\mathcal{Q}$ if it can never produce a trace in $N(Q)$ when starting with the initial conditions set by $\mathcal{Q}$. The size of the testable trace set is limited by the automaton's structure and its preconditions. As implied, a QCFSA's negative trace set is always a subset of its testable trace set, even though the automaton's classification logic could theoretically be applied to a larger set of valid system traces. Thus, very selective preconditions may limit the testable trace set, and subsequently, its negative trace set. Section 4.6.1 expounds upon the impact of preconditions on a generated runtime monitor's classification capabilities.

4.3 Formalising DATEs for Erlang

The DATEs produced by the translation do not make use of the automata's channel communication and timer constructs. So as to maintain a clear and simple representation, Erlang DATEs defined throughout the remainder of this text omit the timer and channel constructs, and are described by a tuple $\langle Q, q_0, \rightarrow, B, A \rangle$, where:

- $\mathbf{Q}$ is a set of states
- $\mathbf{q_0} \in \mathbf{Q}$ is the initial state
- $\rightarrow \subseteq \mathbf{Q} \times \mathbf{event} \times (\mathbf{\Theta} \rightarrow \mathbb{B}) \times (\mathbf{\Theta} \rightarrow \mathbf{\Theta}) \times \mathbf{Q}$ is the transition relation
- $\mathbf{B}$ is the set of bad states
- $\mathbf{A}$ is the set of accepting states

By adding an empty set of channels and null timer actions, one can rewrite the transition relation described above into its equivalent full DATE notation as

$$\rightarrow \subseteq Q \times event \times ((\Theta \times \varepsilon) \rightarrow \Theta \rightarrow \mathbb{B}) \times \varepsilon \times \emptyset \times (\Theta \rightarrow \Theta) \times Q$$

Accepting states exist to signify that a monitor has terminated, and are used primarily for implementation purposes to notify the runtime verification framework that it may dispose of a monitor.

4.4 The Principles of Translation

When comparing the QCFSA (Figure 4.1) and DATE (Figure 5.2) properties defined for the light switch controller example, it is apparent that whilst DATE monitors serve solely as classifiers, QCFSAs must contain instructions to both generate and classify traces. At an abstract level, translating a QCFSA into an DATE involves the extraction of the classification components from the QCFSA and their reformulation into a DATE structure. It is imperative that the semantics of the original QuickCheck automaton's verification logic are preserved through the translation, that is, the derived automaton must perform identically to the original automaton with regards to its capabilities as a classifier.

The following discussion concerns the translation of the theoretical QCFSA models into equivalent DATE structures, making links to the implementation of the procedures used where necessary. Subsequent chapters build on this analysis, providing a more complete treatment of the implementation and application of the described techniques.

4.4.1 Primary Differences Between QCFSAs and DATEs

The aim of the translation is to find suitable analogues to the QCFSA constructs within DATEs. While certain notions within QCFSAs, such as *state-changing actions*, can be readily translated into DATE counterparts, other elements require some degree of reinterpretation. The following is an overview of two fundamental differences between the notations, namely how they represent error states and how they handle the basic units of monitorable computation, that is, *events*.

Explicit Bad and Idle States

DATEs and QCFSAs differ in that the former make use of explicit *bad* and *accepting* states. In a QCFSA, each state is, by definition, accepting, as traces of any length may be generated, with the generation engine being able to stop at any state. Whether or not a system is in a valid state is determined based on the evaluation of each transition's postcondition. In contrast, violations within DATEs are only detected once a bad state is reached.

When converting QCFSAs into DATEs, a bad state is added as a catch-all end state for transitions that lead to failure. Additionally, an accepting *idle* state is added, for reasons which will be elucidated shortly.

QCFSA and DATE events

A transition in a QCFSA is taken based on its precondition, and is classified as correct or failing by checking its postcondition after a function has returned. The valuation of these conditions will vary based on the system state at the point of inspection. Although the generation and execution phases of a QuickCheck automaton are performed separately, taking a transition in a QCFSA can be seen as a single action which internalises several steps. In a single step, the QuickCheck engine:

1. Chooses the next eligible function based on its precondition's valuation
2. Executes the function and stores its return value
3. Classifies success or failure based on the postcondition's valuation, given the modified state and return value
    - If the transition is valid, it executes an associated state-changing action.

In this respect, QuickCheck transitions must be treated as single, atomic actions, which either succeed and progress to a next state or fail. Thus, a QuickCheck event represents an entire *interval* that combines the function's entry and exit processes into a single event, and the internal steps cannot be accessed separately. One cannot, for example, classify an event immediately on entering a function.

Unlike QuickCheck, where traces are created and interpreted by its engine, runtime verification deals with the classification of externally-generated event sequences. In this scenario, events can be any trigger points within a program, such as state-changing actions. For example, DATEs allow events to be triggered on entering or leaving a function. Given a QuickCheck event, one can inspect a system state *prior to* or *immediately following* the execution of its associated function. Most events will modify the system state in some manner. For instance, a function call may increment some global counter[1] or return a result. As a function's return value is taken to be part of the new global system state, it is rarely the case that a function will not modify the system state, as such a function would be of very limited utility. In general, given event e_f associated with function `f()`, one can inspect the system state:

- before executing `f()`, or
- immediately after `f()` has returned. The new state will include the function's return value.

The point at which an event is monitored is based on the property being checked. For example, to check that a function is never invoked with a specific argument value, one would monitor the system prior to the function's execution, or at event *entry*. Conversely, checking that a function's execution does not violate a system invariant would be done once it terminates, before an event *exits*.

[1] In the case of Erlang, a global counter would have to be managed by and accessed through a server process. In a system with multiple processes, this would still have the effect of introducing *side-effects*, with such a server behaving similarly to a shared variable.

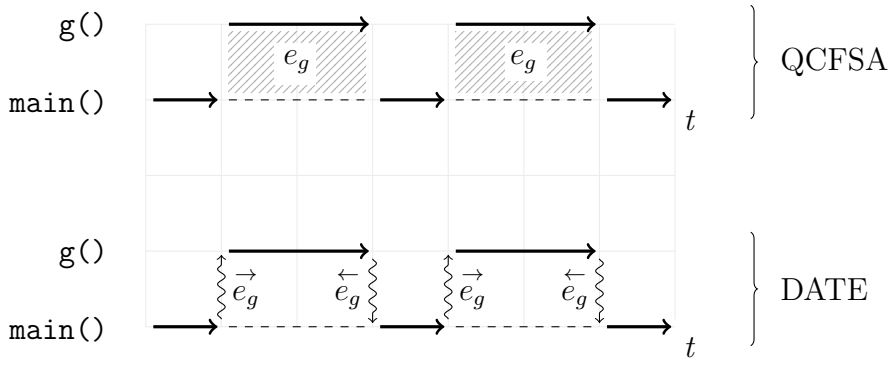

Figure 4.2: Timeline of events triggered when invoking a function `g()` twice in succession. QCFSA event e_g can be seen to embody $\overrightarrow{e_g} \cdot \overleftarrow{e_g}$, with some caveats.

Unlike QCFSAs, DATEs are concerned with events happening at *points* (*edge triggering*). Functions are no longer treated as single, monolithic events, rather they consist of a function entry event followed by zero or more internally-generated events, after which a function exit event is generated, provided that the function terminates. When translating QCFSAs into DATEs, it is thus imperative that the system be instrumented effectively, and that interval events are correctly translated in terms of point-events without introducing unintended behaviours into the system.

4.4.2 A Simplified Translation

The following section describes a first approximation to a complete description of a translation procedure which converts a QCFSA into a DATE, illustrated through an example.

The comprehension of any approach to the translation relies on the understanding of the purpose of pre- and postconditions. Preconditions serve to limit the generation of certain traces. They do not, by themselves, classify traces, rather they restrict the monitoring process to some identified interesting subset of possible traversals. Making preconditions more restrictive will decrease the size of the set of testable traces, which often helps during the trace generation phase by preventing uninteresting or previously-tested traces from being tested again. Nevertheless, very selective preconditions may lessen the efficaciousness of the generated runtime monitor, as will be discussed in Section 4.6.1.

A failing precondition does not imply that a violation has been detected. Instead, it simply states that further verification on that trace should be suspended, as the property no longer concerns that particular sequence of events. Within QuickCheck, a failing precondition would lead to different traces being generated and tested, yet in the case of runtime monitoring, one would have to cease monitoring that particular instance of a program. The precise mechanics that dictate how this is done will be covered in Section 4.4.2. Postconditions serve as classifiers, and are checked once a precondition has suc-

ceeded and the related function has executed. Thus, a trace can only be invalid if one of its transitions has a valid precondition and an invalid postcondition.

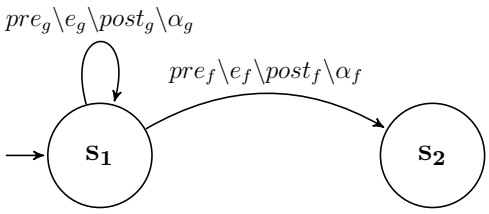

Figure 4.3: A simple QCFSA concerned with two events, e_g and e_f, which represent functions `g()` and `f()`, respectively. Precondition and postcondition sets, as well as the state changing actions α_g and α_f, are defined using abstract values.

Figure 4.3 is an example of a simple, deterministic QuickCheck automaton. While the *pre* and *post* variables represent sets, one must remember that they are ultimately characterised using the total `precondition` and `postcondition` functions defined for the automaton. Given that events e_g and e_f are triggered on taking transitions involving functions `g` and `f`, respectively, the automaton is generating and subsequently testing traces which contain zero or more invocations to function `g`, followed by a single invocation of `f`. Thus, the automaton is concerned with traces of the form $e_g{}^* e_f$.

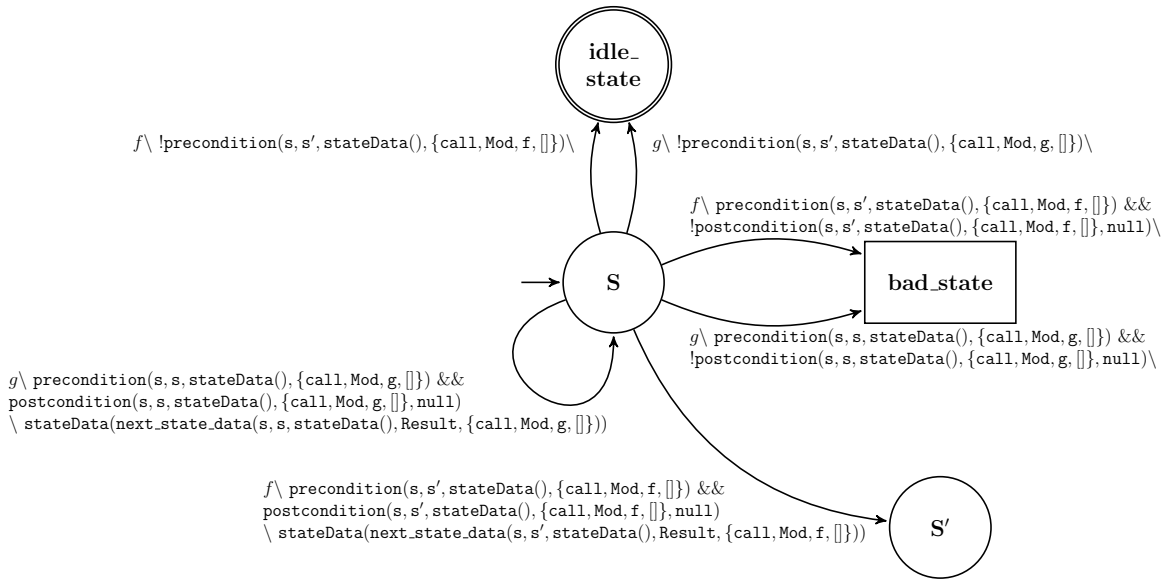

Figure 4.4: The QCFSA translated into an implemented DATE. For **Result** to be defined, the events must be triggered on event exit.

For now, it is assumed that neither event will trigger subsequent events during the execution of their corresponding function. The central idea in translating from a QCFSA $\mathcal{Q}$ to a DATE $\mathcal{E}$ is that each function on a transition in $\mathcal{Q}$ is mirrored by a corresponding event in $\mathcal{E}$. This offloads the act of generating a sequence of events to the system under test. As Erlang does not allow the use of global variables, the automaton's state data is stored within a server process. The function `stateData()` is thus defined to serve as a

data store. When passed a parameter, `stateData` updates the stored version of the data with the new value specified. When invoked without arguments, it returns the current state data. In addition, as described in Section 4.4.1, explicit bad and idle states are added to the translated automaton so as to replicate the QCFSA's behaviour within a DATE.

For each event e in the original automaton, transitions are added so that:

I) If e occurs and its precondition and postcondition succeed, then the system moves to the original target state whilst updating the automaton's state.

II) If e occurs and its postcondition fails after its precondition succeeds, then the system moves into a bad state.

III) If e occurs and its precondition fails, then the automaton moves to an idle state.

Type I transitions represent a successful transition, where both pre- and postconditions return true. Type II transitions cater for those cases were the precondition was met (signifying that monitoring should be applied) yet the postcondition failed, implying that the property has been violated. Type III transitions regard instances whereby the system has generated a series of events which the original property was not designed to classify, in which case monitoring should cease.

Figure 4.4 illustrates a DATE derived from the QCFSA in Figure 4.3. **Mod** refers to the module within which the functions being tested reside, while **Args** is a reserved keyword managed by the runtime monitoring engine, which replaces the variable with the argument list with which the associated function has been invoked, as will be described in Section 5.3.2.

Conditions		**Next State**
precondition	postcondition	
T	T	Valid next state
T	F	Bad state
F	-	Idle state

Table 4.1: The types of transitions generated on translation, their conditions (simplified) and end states.

Although it is free from non-determinism, the translation depicted suffers from two major shortcomings. The first is related to the fact that events are triggered either on function entry or exit, at which point the conditions must be checked. If an event is triggered on entry, then the conditional guards will not have access to the function's result. On the other hand, triggering on the exit event will cause the precondition to be evaluated after the function has executed. This may result in the precondition being evaluated incorrectly, as the condition may no longer hold. In general, one cannot assume that a function will not modify the system state, thus disallowing both conditions from being checked simultaneously and requiring the system to respect the original precedence of calls.

The second issue is that events will be generated whenever a monitored (and hence instrumented) function will be entered or exited. If one were to remove the assumption that the functions being monitored do not themselves invoke other instrumented functions, it could be the case that traces which were not originally meant for verification will be monitored. These issues will be addressed within the following sections, so as to provide a more complete translation.

Totality

When translating into a DATE, it is imperative that the original QCFSA's semantics are preserved. Thus, the DATE should make no attempt to classify traces which the original QCFSA was not designed to test. Instead, it should be constructed so that if it is presented with a trace which does not form part of a set of *testable traces* (Section 4.2), then monitoring should halt, and no violation should be reported. For example, by examining the automaton defined in Figure 4.3, it is evident that the execution $\texttt{f}() \cdot \texttt{g}()$ is not part of the set of testable traces. This sequence should cause the automaton to cease operation without returning a verdict.

As the monitor may be presented with any arbitrary trace, the set of testable traces must be expanded to accept all combinations of interesting events, without changing the size of the set of negative traces. Thus, the QCFSA is rendered *total*, with any new transitions leading to an idle state without reporting failure. This will allow the automaton to accept traces which were previously rejected either due to strict preconditions or non-existent arcs, without altering the negative verdict. Any added arcs will not contribute to the automaton's capacity to identify bad traces, that is, although the testable trace set has grown, its negative trace set remains unaltered.

For an automaton to be total, there must be an outgoing transition for every interesting event at every state such that at least one transition is always eligible for traversal. In addition, as the automaton is also deterministic, there cannot be more than one eligible outgoing transition.

Example 4.4.2.1. *The set of interesting events Σ for the QCFSA described in Figure 4.3 is $\{e_f, e_g\}$. To make the automaton total, an idle state is added, and each state in the automaton is examined separately.*

- *S contains outgoing transitions for all the events in Σ, yet transitions catering for the complement of the preconditions must be added, as follows*

  $(S_1, (\overline{pre_g}, e_g, 2^{\Theta}, \lambda\ s \cdot\ s), idle\_state)$
  $(S_1, (\overline{pre_f}, e_f, 2^{\Theta}, \lambda\ s \cdot\ s), idle\_state)$

- *S' has no outgoing transitions. Thus, transitions must be added for every event in Σ, namely*

  $(S_2, (2^{\Theta}, e_g, 2^{\Theta}, \lambda\ s \cdot\ s), idle\_state)$
  $(S_2, (2^{\Theta}, e_f, 2^{\Theta}, \lambda\ s \cdot\ s), idle\_state)$

Receiving either event at S' would thus lead to the idle state.

- *The idle_state itself must consume all subsequent events. Thus,*

$$(idle\_state,\ (2^{\Theta},\ e_g,\ 2^{\Theta},\ \lambda\ s\cdot\ s),\ idle\_state)$$
$$(idle\_state,\ (2^{\Theta},\ e_f,\ 2^{\Theta},\ \lambda\ s\cdot\ s),\ idle\_state)$$

In theory, the idle state need not be rendered total, as it is a final state and can be used to immediately halt monitoring. Totality ensures that the automaton will behave correctly as a classifier, irrespective of the underlying monitor's implementation.

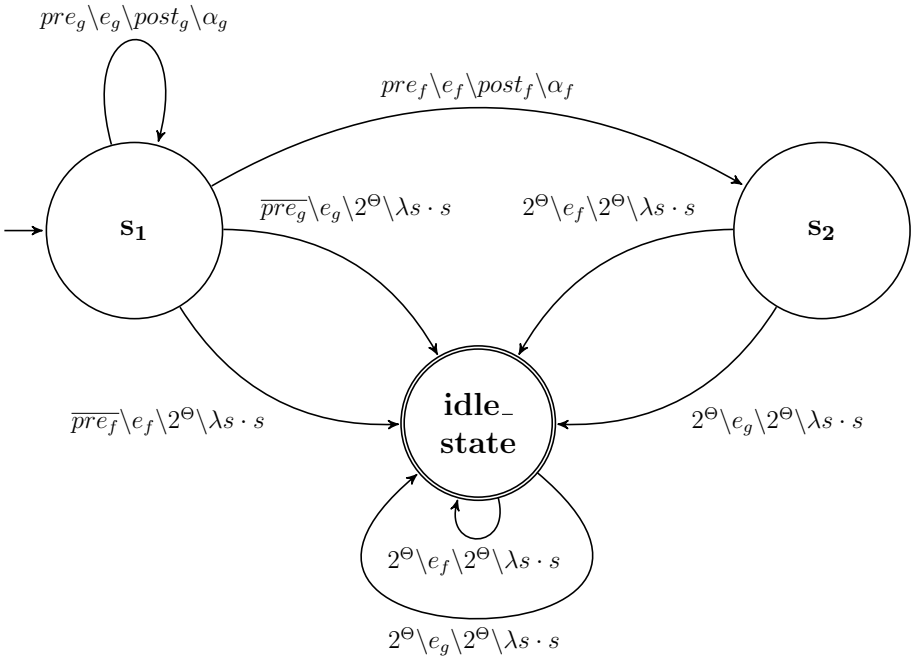

Figure 4.5: The QCFSA in Figure 4.3 rendered total

Splitting Events

As currently formulated, the pre- and postcondition checks are being performed simultaneously at the given point of instrumentation. More specifically, if the event is instrumented at its entry point, then the monitor may only access the state data prior to the body's execution. Similarly, instrumenting at the function's exit point will provide access to the computation's result along with the final system state. So as to replicate the ordering exercised by QuickCheck, preconditions should be checked on function entry, whilst postconditions should be checked at the exit point. This allows assertions to be performed both prior and following an execution, and also enables the verification of a function's result.

To enable the checking of both types of conditions, a function should be instrumented on entry and on exit. Thus, each QCFSA event can be considered as a two-part event,

consisting of a DATE *entry* and *uponReturning* event. If `f()` is a function being monitored by a DATE, $\overrightarrow{e_f}$ is an event fired as soon as `f()` is entered, while $\overleftarrow{e_f}$ is fired once `f()` has completed its execution and is about to return control to its parent function.

As an example, one may consider the branch concerning the function `f()` in the automaton illustrated in Figure 4.3. If the function is instrumented at its entry and exit points, calling `f()` will generate the event sequence $\overrightarrow{e_f} \cdot \overleftarrow{e_f}$, assuming that the function body of `f()` will not, through the invocation of other monitored functions, cause more events to be triggered.

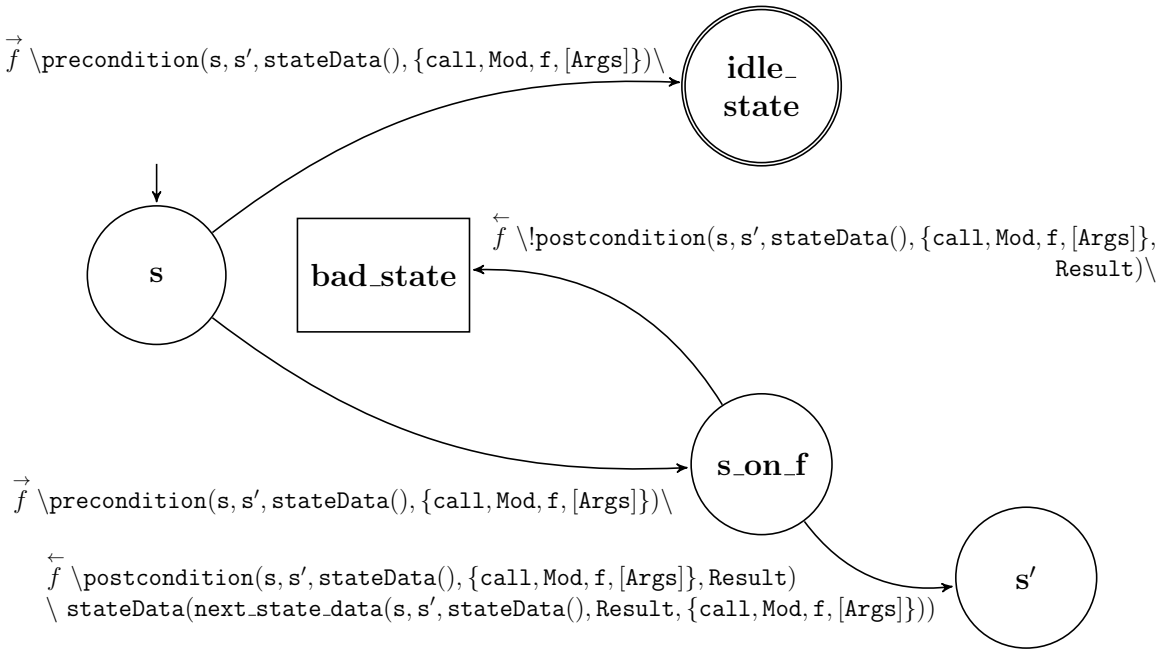

Figure 4.6: Monitoring function `f()` using entry and exit events

Figure 4.6 describes the DATE which monitors the function using split events. On receiving $\overrightarrow{e_f}$, the precondition is checked. On success, the automaton enters an intermediate state, which can be seen as representing the function's computation. Once the system receives the function exit event, the postcondition is evaluated. If it is invalid, then the system enters a bad state, otherwise it moves on to the valid target state.

In general, one cannot assume that a monitored function will never invoke another monitored function as an internal action. To illustrate, one may consider the original QCFSA described in Figure 4.3. It is conceivable that `f()` invokes `g()` within its body. Such an action would be of no consequence for a QCFSA, as it does not monitor the function's intermediate events individually, rather it only checks the function's initial conditions and final result. Yet this is not the case for DATEs. As the functions being tested have been instrumented to fire an event on each call and return, invoking any monitored function will generate an entry and exit event pair.

The issue becomes readily apparent when analysing the monitoring of recursive functions, such as the `dec(N)` function described in Listing 4.1, which recursively decrements a given value whilst incrementing the running result, returning the original value once the argument's value has reached zero. Each recursive call returns the value of the counter at

that point in the call stack.

Listing 4.1: Recursive decrement function

```
1  dec(0) ->
2    0;
3  dec(N) when N > 0 ->
4    dec(N - 1) + 1.
```

Listing 4.2 is a high-level implementation of an instrumented version of `dec()`. The original function is renamed to `e_dec()`, whilst `dec()` is replaced by a function containing the event generation code at the entry and exit points. A call to `e_dec()` is performed between the entry and exit points, saving the returned value in a temporary **Result** variable. One should note that the recursive calls within the original `dec()` function (now `e_dec()`) are not renamed, and refer to the instrumented version of `dec()`.

Listing 4.2: Instrumented `dec()` function

```
1  e_dec(0) ->
2    0;
3  e_dec(N) when N > 0 ->
4    dec(N - 1) + 1.
5
6  dec(N) ->
7    generate e_dec^→ ,
8    Result = e_dec(N),
9    generate e_dec^←(Result),
10   Result.
```

The simplest test would be to check whether, given a positive integer as an input, the result returned would be equal to the input argument. For a QCFSA invoking `dec(N)` over a transition from S to S', this can be readily expressed using pre- and postconditions as shown below.

```
1  precondition(s, s2, _, {call, sut, dec, N}) ->
2    N >= 0.
3
4  postcondition(s, s2, _, {call, sut, dec, N}, Result) ->
5    Result == N.
```

Translating the QCFSA into a DATE produces an automaton similar to that defined for the generic function `f()` in Figure 4.6, replacing `f()` with `dec()`. The problem that arises with internally generated events is that the original QCFSA property does not consider their correct behaviour, as it only places conditions on the parent event. If $e_{dec}^{\rightarrow N}$ represents an entry event with N passed as an argument to `dec(N)`, and $e_{dec}^{\leftarrow =R}$ stands for an exit event which returns value R, calling `dec(3)` would generate the event sequence

$$\overset{\rightarrow\ 3}{e_{dec}}\ \overset{\rightarrow\ 2}{e_{dec}}\ \overset{\rightarrow\ 1}{e_{dec}}\ \overset{\rightarrow\ 0}{e_{dec}}\ \overset{\leftarrow\ =0}{e_{dec}}\ \overset{\leftarrow\ =1}{e_{dec}}\ \overset{\leftarrow\ =2}{e_{dec}}\ \overset{\leftarrow\ =3}{e_{dec}}$$

On receiving $\overset{\rightarrow\ 3}{e_{dec}}$, the monitor will check the precondition's result. As 3 is greater or equal to zero, the condition will be satisfied and the automaton will step into the intermediate state, signalling to the system under test that execution may proceed.

If one cannot identify individual events of the same type, then the intermediate events will mislead the verifier, as:

Discarding all unexpected entry events, or rejecting all entry events of the same type until an exit event is received, will leave the automaton in the same state, yet the precondition will be paired with the wrong postcondition, as the wrong exit event will be considered. More specifically, consuming all events of the same type as the most recent event would give the event chain

$$\overset{\rightarrow\ 3}{e_{dec}}\ \overset{\leftarrow\ =0}{e_{dec}}$$

As $3 \neq 0$, the property will fail.

Updating the current event with the most recent event of the same type would change the property's semantics. In this case, the property would initially succeed, as the event sequence would be $\overset{\rightarrow\ 0}{e_{dec}}\ \overset{\leftarrow\ =0}{e_{dec}}\ \overset{\leftarrow\ =1}{e_{dec}}\ \overset{\leftarrow\ =2}{e_{dec}}\ \overset{\leftarrow\ =3}{e_{dec}}$, with the innermost pair being matched. The problem is that the property is being checked on the wrong event pair, which will cause verdicts to become unpredictable for the general case. In addition, the monitor will have several lingering and superfluous events residing on its event queue.

The simplest and most effective way of handling intermediate events would be to prevent these events from being generated in the first place. One way of achieving this relies on the ability to label events with unique identifiers. That way, the system would be able to discard any unexpected intermediate events, halting progress within the automaton until the corresponding exit event is received. For instance, a unique integer could be assigned by the instrumentation code to every event prior to its forwarding to the runtime monitoring engine.

The `next_id()` function could access a counter stored within a server process which grows in increments of two. Using a counter is helpful in that the identifier of each entry event's matching exit event can be easily inferred from the former's identifier, simply by adding one to this value. Thus, when the monitoring engine receives an entry event, it can suspend progress within the automaton until an event whose identifier is one more than the original entry event's ID is received[2]. Such a scheme avoids having to modify the generated DATEs to cater for intermediate events, rendering the translation cleaner and more intuitive.

Thus, using incrementing event counters, calling `dec(3)` would produce the event trace:

[2]Incrementing by 2 avoids calling the function twice, facilitating concurrent accesses to the counter.

Events	$\overset{\rightarrow 3}{e_{dec}}$	$\overset{\rightarrow 2}{e_{dec}}$	$\overset{\rightarrow 1}{e_{dec}}$	$\overset{\rightarrow 0}{e_{dec}}$	$\overset{\leftarrow =0}{e_{dec}}$	$\overset{\leftarrow =1}{e_{dec}}$	$\overset{\leftarrow =2}{e_{dec}}$	$\overset{\leftarrow =3}{e_{dec}}$
ID	0	2	4	6	7	5	3	1

Discarding all intermediate events whose ID is not equal to 1 will contract the trace to

$$\overset{\rightarrow 3}{e_{dec}}\ \overset{\leftarrow =3}{e_{dec}}$$

which is the top level event entry and exit pair with which the property is concerned. Once a matching event has been found, the system will step through the automaton and the next entry event to be received will be latched on to by the automaton, restarting the event capture process.

For the event consolidation scheme to work, events must be received in the same order in which they were generated, which can be safely assumed when monitoring events originating from a single processes [Eri10a]. Also, recursive functions must terminate at some point, otherwise a corresponding exit event will never be received by the monitoring engine. The latter requirement, while not necessary for runtime monitoring, is always assumed when writing QCFSA properties, as otherwise postconditions would never be checked, rendering verification impossible. Thus, caution must be exercised when verifying server processes.

Figure 4.7 illustrates the automaton produced on applying the complete transformation to the light-switch QCFSA described earlier, first rendering it total and then splitting each transition. This automaton is not total due to the introduction of the intermediate and bad states. If a bad state is entered, the system would stop receiving events, making further transitions unnecessary. In the case of intermediate states, the automaton should not progress unless the appropriate exit event is received. Given an entry event, an exit event of the matching type will always be generated due to the nature of the instrumentation code, unless the function fails to terminate or crashes. In addition, non-intermediate states do not have to cater for outgoing exit events, as these cannot be generated prior to an entry event.

Listing 4.3: Instrumented `dec()` function with unique event IDs

```
1 dec(N) ->
2   ID→ = next_id(),
3   ID← = ID→ + 1,
4
5   generate ⟨ ID→, e→dec ⟩,
6   Result = e_dec(N),
7   generate ⟨ ID←, e←dec(Result) ⟩,
8   Result.
```

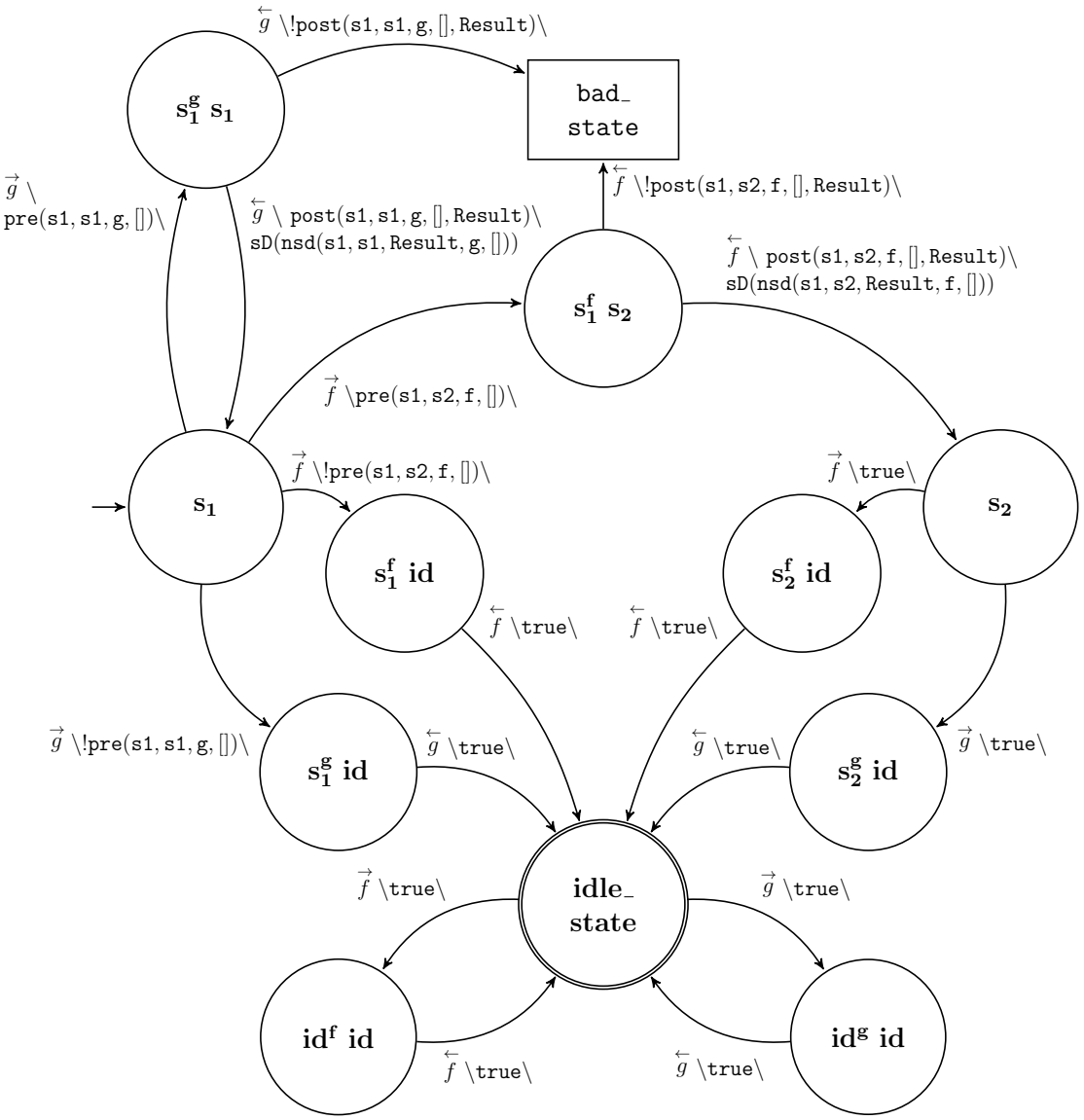

Figure 4.7: The QCFSA in Figure 4.3 transformed into a DATE. The original QCFSA is first rendered total, after which each event is split into entry and exit events using an intermediate state. As internal (silent) actions are handled by the monitoring engine, they do not appear within the given automaton. Functions over transitions are redefined as follows:
$\texttt{sD(S)} \stackrel{\text{def}}{=} \texttt{stateData(S)}$,
$\texttt{pre}(\texttt{s}_1, \texttt{s}_2, \texttt{f}, \texttt{A}) \stackrel{\text{def}}{=} \texttt{precondition}(\texttt{s}_1, \texttt{s}_2, \texttt{stateData}(), \{\texttt{call}, \texttt{Mod}, \texttt{f}, \texttt{A}\})$,
$\texttt{post}(\texttt{s}_1, \texttt{s}_2, \texttt{f}, \texttt{A}, \texttt{R}) \stackrel{\text{def}}{=} \texttt{postcondition}(\texttt{s}_1, \texttt{s}_2, \texttt{stateData}(), \{\texttt{call}, \texttt{Mod}, \texttt{f}, \texttt{A}\}, \texttt{R})$,
$\texttt{nsd}(\texttt{s}_1, \texttt{s}_2, \texttt{R}, \texttt{f}, \texttt{A}) \stackrel{\text{def}}{=} \texttt{next\_state\_data}(\texttt{s}_1, \texttt{s}_2, \texttt{stateData}(), \texttt{R}, \{\texttt{call}, \texttt{Mod}, \texttt{f}, \texttt{A}\})$

4.5 A Formal Translation of QCFSAs into DATEs

In the previous section, a translation from a DATE to a QCFSA was presented using an example. The following section will describe a formal procedure for translating QCFSAs into DATEs that will recognise and classify precisely the same set of negative traces as the former.

The translation consists of two phases. First, an automaton must be rendered *total*, so that it may be able to produce a verdict for traces which the original automaton was not designed to accept. Naturally, verdicts for these newly-introduced transitions will be inconclusive, that is, while the event sequences will be accepted by the automaton, it will never classify them as negative traces. The second phase changes the total automaton from one that recognises QCFSA events to one that consumes entry and exit events generated at runtime.

The translation process assumes that the original QCFSA contains no states named `idle_state` or `bad_state`, which would otherwise have to be renamed prior to executing the translation. In addition, the translation process does not cater for the use of *state attribute values*. As QuickCheck enumerates all reachable states, such values would have to be finite [Quv10], and could be enumerated prior to performing the transition, resulting in no loss of expressivity.

4.5.1 Totality of QCFSA

The following is a description of the process of converting an automaton $\mathcal{Q}$ to its total equivalent $\mathcal{Q}_T$. A total QCFSA will have one or more outgoing transitions for each interesting event at every state, such that there will always be a transition defined for any given precondition and function pair. This is achieved through the addition of an idle state, to which the system will progress whenever it is faced with a function-condition pair which was not catered for within the original automaton.

As a generator, the total version of a QCFSA is of limited utility as it will increase the number of traces which must be tested without a corresponding improvement in its coverage, since the new transitions will not recognise invalid behaviour. The transformed QCFSA is still deterministic, as the preconditions for outgoing arcs of the same event type do not overlap.

Before providing a construction detailing the conversion to a total automaton, it is useful to define the concept of *precondition coverage*. Given an automaton $\mathcal{Q}$, a set of preconditions is said to *cover* an event e at state q if there exists a transition within $\mathcal{Q}$ which is defined for those conditions. The set of uncovered preconditions is thus defined as the set of preconditions for which no outgoing transition from the given state exists, and is defined as follows.

Definition 4.5.1.1 (Uncovered preconditions of a QCFSA state). *Given a QCFSA $\mathcal{Q}$, the uncovered preconditions for an event* **e** *at state* **q** *can be computed as*

$$uncoveredPre_{\mathcal{Q}}(q,e) \stackrel{def}{=} 2^{\Theta} - \bigcup_{(q,(pre,e,post,\alpha),q')\in\rightarrow} \{pre\}$$

Given a QCFSA $\mathcal{Q} = \langle Q, q_0, \Sigma, \Theta_0, run, \rightarrow\rangle$, one may construct an equivalent total QCFSA $\mathcal{Q}_T = \langle Q^T, q_0^T, \Sigma^T, \Theta_0^T, run^T, \rightarrow^T\rangle$ using the aforementioned notion of precondition coverage, resulting in the following construction:

$$\begin{array}{lcl}
\mathbf{Q^T} & \stackrel{\text{def}}{=} & Q \cup \{idle\} \\
\mathbf{q_0^T} & \stackrel{\text{def}}{=} & q_0 \\
\mathbf{\Sigma^T} & \stackrel{\text{def}}{=} & \Sigma \\
\mathbf{\Theta_0^T} & \stackrel{\text{def}}{=} & \Theta_0 \\
\mathbf{run^T} & \stackrel{\text{def}}{=} & run \\
\rightarrow^{\mathbf{T}} & \stackrel{\text{def}}{=} & \{(q, (uncoveredPre_{\mathcal{Q}}(q,e), e, 2^{\Theta}, id), idle) \mid e \in \Sigma, q \in Q\} \\
 & & \cup\ \{(idle, (2^{\Theta}, e, 2^{\Theta}), idle) \mid e \in \Sigma\} \\
 & & \cup\ \rightarrow
\end{array}$$

As the idle state does not form part of the original automaton, transitions onto itself have to be added explicitly.

Proofs

Unless the original automaton $\mathcal{Q}$ is in itself total, the process of rendering $\mathcal{Q}_T$ will introduce new transitions, as dictated by the construction presented (minimally, transitions for the idle state will be added). As a consequence of these new transitions, the testing set of the automaton $\mathcal{Q}_T$ will be larger or at least as large as that of $\mathcal{Q}$, that is, $T(\mathcal{Q}) \subseteq T(\mathcal{Q}_T)$. This is because, in addition to the original traces, the automaton will be able to generate new function call sequences.

For the purpose of runtime monitoring, deriving a runtime monitor from automata with differing testable trace sets would be of no consequence to its ability to detect bad traces, provided that the set of negative traces is the same for both automata. To paraphrase, the identification of bad states depends solely on the automaton's characterisation of negative traces, and is unaffected by its embodiment of the positive traces. It is thus imperative that the translation leaves the set of negative traces intact, neither introducing nor removing any traces from the original automaton's set.

Lemma 4.5.1.1 (**No lost negative traces**). $N(\mathcal{Q}) \subseteq N(\mathcal{Q}_T)$

Proof. The first lemma is derived trivially from the construction. When building $\mathcal{Q}_T$, the transitions of $\mathcal{Q}$ are preserved, and any added transitions do not interfere with those defined within the original automaton. Thus, though the set of testable traces of $\mathcal{Q}_T$ may have grown under the translation, the traces in $N(\mathcal{Q})$ still exist within the total automaton, which implies that the lemma must hold. □

Lemma 4.5.1.2 (**Idle leads to idle**). *For automaton $\mathcal{Q}_T$, all configurations that can be reached from the idle state are in the idle state, that is, if idle $\stackrel{w}{\Rightarrow}$ q then q = idle*

Proof.
Base Case: w = ε

$(idle, \theta) \overset{\varepsilon}{\Rightarrow}_{\mathcal{Q}_T} (q, \theta')$
$\implies$ Definition of $\overset{\varepsilon}{\Rightarrow}$ leaves configuration unchanged
$(idle, \theta) = (q, \theta)$
$\implies$ Matching names
$q = idle$

Inductive Hypothesis: Assume lemma holds for $w = s$

$(idle, \theta) \overset{s}{\Rightarrow}_{\mathcal{Q}_T} (idle, \theta')$

Prove: Lemma holds for $w = s +\!\!+ \langle a \rangle$

$(idle, \theta) \overset{s+\!\!+\langle a \rangle}{\Rightarrow}_{\mathcal{Q}_T} (q, \theta')$
$\implies$ Sequence of events (Definition 4.2.3.3)
$(idle, \theta) \overset{s}{\Rightarrow}_{\mathcal{Q}_T} (q'', \theta'') \overset{a}{\Rightarrow}_{\mathcal{Q}_T} (q, \theta')$
$\implies$ Inductive Hypothesis, $q'' = idle$
$(idle, \theta'') \overset{a}{\Rightarrow}_{\mathcal{Q}_T} (q, \theta')$
$\implies$ All single transitions leaving from *idle* lead to *idle* (by construction of $\mathcal{Q}_T$)
$q = idle$ □

Lemma 4.5.1.3 (**No idle intermediate state if end is not idle**). *If a sequence of transitions in $\mathcal{Q}_T$ does not end with an idle state, then none of the intermediate configurations pass through the idle state. This can be expressed as $(q, \theta) \overset{a}{\Rightarrow}_{\mathcal{Q}_T} (q'', \theta'') \overset{s}{\Rightarrow}_{\mathcal{Q}_T} (q', \theta') \wedge q' \neq idle \implies q'' \neq idle$*

Proof.
Assume that $q'' = idle$

$(q, \theta) \overset{a}{\Rightarrow}_{\mathcal{Q}_T} (idle, \theta'') \overset{s}{\Rightarrow}_{\mathcal{Q}_T} (q', \theta') \wedge q' \neq idle$
$\implies$ From idle, one can only reach idle (Lemma 4.5.1.2)
$q' = idle$
$\implies$ Antecedent $\rightarrow q' \neq idle$
$q' = idle \wedge q' \neq idle$
$\implies$ Contradiction;
$q'' \neq idle$ □

Lemma 4.5.1.4 (**Non-idle transitions in $\mathcal{Q}_T$ are in $\mathcal{Q}$**). *If a transition in $\mathcal{Q}_T$ does not lead to an idle state, then that transition exists within the automaton prior to it being made total. Thus, $(q, \theta) \overset{w}{\Rightarrow}_{\mathcal{Q}_T} (q', \theta') \wedge q' \neq idle \implies (q, \theta) \overset{w}{\Rightarrow}_{\mathcal{Q}} (q', \theta')$*

Proof. The end state q' is, by definition, not idle. Using Lemma 4.5.1.3, this also implies that none of the intermediate configurations from q to q' pass through the idle state. Thus, the trace w never causes the automaton to enter an idle configuration. By examining the construction, one finds that all the transitions in $\mathcal{Q}_T$ are either transitions in $\mathcal{Q}$ or newly-added transitions that lead to an idle state. As the trace w never traverses the latter transitions (as there are no idle configurations), it follows that the transitions form part of the original set of transitions defined in $\mathcal{Q}$. □

Lemma 4.5.1.5 (**No added negative traces**). $N(\mathcal{Q}_T) \subseteq N(\mathcal{Q})$

Proof.

$w \in N(\mathcal{Q}_T)$
$\implies$ Definition of a negative trace (Definition 4.2.4.1)
$(q_0, \theta_0) \overset{w}{\Rightarrow}_{\mathcal{Q}_T} \otimes$
$\implies$ $w = s +\!\!+ \langle a \rangle$, sequence to a bad configuration (Definition 4.2.3.4)
$(q_0, \theta_0) \overset{s}{\Rightarrow}_{\mathcal{Q}_T} (q', \theta') \overset{a}{\Rightarrow}_{\mathcal{Q}_T} \otimes$
$\implies$ q' cannot be *idle* (Lemma 4.5.1.2)
$(q_0, \theta_0) \overset{s}{\Rightarrow}_{\mathcal{Q}_T} (q', \theta') \overset{a}{\Rightarrow}_{\mathcal{Q}_T} \otimes \qquad [q' \neq idle]$
$\implies$ If q' is not idle, then the path exists in $\mathcal{Q}$ (Lemma 4.5.1.4 and construction)
$(q_0, \theta_0) \overset{s}{\Rightarrow}_{\mathcal{Q}} (q', \theta') \overset{a}{\Rightarrow}_{\mathcal{Q}} \otimes$
$\implies$ Definition of a negative trace & catenation of trace (Definition 4.2.4.1)
$w \in N(\mathcal{Q})$ □

Theorem 4.5.1.1 (**Equality of the set of negative traces**). *As $N(\mathcal{Q}) \subseteq N(\mathcal{Q}_T)$ (Lemma 4.5.1.1) and $N(\mathcal{Q}_T) \subseteq N(\mathcal{Q})$ (Lemma 4.5.1.5), it follows that $N(\mathcal{Q}) = N(\mathcal{Q}_T)$.*

4.5.2 A Complete Translation

Based on the example translated earlier, as well as the analysis conducted when considering the splitting of events into pre- and postcondition arcs with intermediate states, a complete translation process will be described herewith.

Given a QCFSA $\mathcal{Q}$ and the total automaton $\mathcal{Q}^T = \langle Q, q_0, \Sigma, \theta_0, run, \rightarrow \rangle$ resulting from the application of the aforementioned construction, one can produce the DATE $\mathcal{D} = \langle Q^D, q_0^D, \rightarrow^D, B^D, A^D \rangle$, where

$$
\begin{array}{lcl}
\mathbf{Q^D} & \overset{\text{def}}{=} & Q \cup \{badstate\} \cup \{int(t) | t \in \rightarrow\} \\
\mathbf{q_0^D} & \overset{\text{def}}{=} & q_0 \\
\mathbf{B} & \overset{\text{def}}{=} & \{badstate\} \\
\mathbf{A} & \overset{\text{def}}{=} & \{idle\} \\
\rightarrow^{\mathbf{D}} & \overset{\text{def}}{=} & \{(q, \vec{e}, \lambda s \cdot (s \in pre), id, q''), \\
 & & \ (q'', \overset{\leftarrow}{e}, \lambda s \cdot (run(e, s) \in post), \alpha, q'), \\
 & & \ (q'', \overset{\leftarrow}{e}, \lambda s \cdot (run(e, s) \notin post), id, badstate) \mid \\
 & & \ \ t \in \rightarrow \wedge\ t = (q, (pre, e, post, \alpha), q') \wedge q'' = int(t)\}
\end{array}
$$

The function $\texttt{id} \overset{\text{def}}{=} \lambda s \cdot s$ is the identity function, while $\texttt{run}:: \Sigma \rightarrow \Theta \rightarrow \Theta$ returns the new system state when executing an event's corresponding function under a given system state, as described in Section 4.2.1. The function $\texttt{int}:: T \rightarrow Q$ returns a unique intermediate state name such that $\forall t \in \rightarrow \cdot \texttt{int}(t) \notin Q$ and $\forall t_1, t_2 \in T \cdot t_1 \neq t_2 \Rightarrow \texttt{int}(t_1) \neq \texttt{int}(t_2)$.

Proofs

The translation from a QCFSA $\mathcal{Q}$ to a DATE $\mathcal{D}$ is a two-step process. First, the automaton is made total, after which the event-splitting translation is applied. Translating a QCFSA $\mathcal{Q}$ to its total equivalent $\mathcal{Q}_T$ was proven to leave the set of negative traces unaltered, that is, $N(\mathcal{Q}_T) = N(\mathcal{Q})$.

Given that $\Pi:: QCFSA \to DATE$ is the function that converts a total QCFSA into a DATE, the next step would be to show that both $\mathcal{Q}_T$ and $\Pi(\mathcal{Q}_T)$ will detect the same set of program failures. This can be done by showing that splitting QCFSA events into DATE entry and exit events will result in sequences that will be recognised by the latter whilst returning the same verdict as the former.

To facilitate this, the *explode* operator $\uparrow$ is defined. Let $\Sigma_{\mathcal{Q}_T}$ be the set of events with which $\mathcal{Q}_T$ is concerned. During execution, every separate invocation of a function related to the events in $\Sigma_{\mathcal{Q}_T}$ can be assigned a unique index, as demonstrated in Section 4.4.2. Thus, let $\Gamma_{\mathcal{Q}_T} \stackrel{\text{def}}{=} \{e_i \mid e \in \Sigma_{\mathcal{Q}_T} \wedge i \in \mathbb{N}\}$ be the set of uniquely identifiable events emitted during an execution. Given a run $w \in \Gamma_{\mathcal{Q}_T}$, $\uparrow w$ will return the set of *exploded traces*, whose elements have been converted into entry and exit events. This is defined recursively as

$$\uparrow \varepsilon \stackrel{\text{def}}{=} \{\varepsilon\}$$
$$\uparrow a_i w \stackrel{\text{def}}{=} \overrightarrow{a_i} \,.\, \overline{(\overleftarrow{a_i})}^{*} \,.\, \overleftarrow{a_i} \,.\, \uparrow w$$

where $\overrightarrow{a_i}$ and $\overleftarrow{a_i}$ are the entry and exit events related to a QCFSA event a_i, as described earlier. As the index is unique to every separate invocation of a monitored function, the split events are guaranteed to also be unique. Thus, it is certain that any entry event $\overrightarrow{a_j}$ preceding or succeeding an event $\overrightarrow{a_i}$ in $\uparrow w$ will have an index such that $i \neq j$, as will any pair of exit events $\overleftarrow{a_i}$ and $\overleftarrow{a_j}$ chosen from $\uparrow w$. This allows one to discriminate unambiguously between pointwise events of the same type, even if they are generated through a recursive call. The function $\Uparrow$ returns $\uparrow$ applied to each trace within a given set.

Conjecture 4.5.2.1 (**Preservation of negative trace set under translation**). *Given any total QuickCheck automaton* $\mathcal{Q}_T$, $\Uparrow N(\mathcal{Q}_T) = N(\Pi(\mathcal{Q}_T))$.

Functions associated with a QCFSA transition are always instrumented at their entry and exit points. Any system events received prior to the matching exit event are consumed. Exploding all the negative traces in $\mathcal{Q}_T$ will result in the equivalent set of negative traces of $\mathcal{Q}_T$ under transformation.

4.6 Writing Properties

4.6.1 Partitions

It is conceivable, and indeed, often the case that failures uncovered by multiple test cases stem from a single fault. More formally, when looking at the system's input space, there may exist a whole set of inputs, rather than a single instance, which triggers one specific type of failure. These sets of inputs are *partitions* of the input space, with each partition corresponding to a single concrete fault in the system.

Consider an arbitrary function $f :: \mathbb{N} \to \mathbb{N}$, which fails when given an input greater than 7. If all inputs above 7 fail for the same reason, then they form part of the same partition.

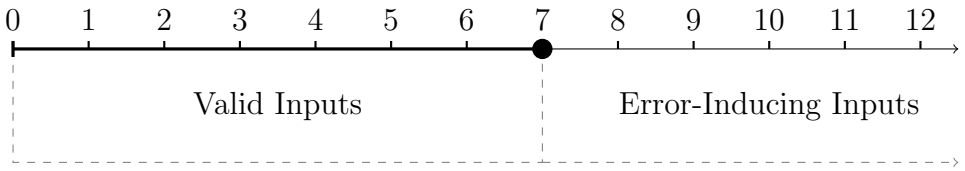

Figure 4.8: Number line depicting input partitions for function f

If the partitions have been identified with certainty, it would suffice to test just one element within that partition, as its result would be representative of the entire partition. Referring to function f, both 8 and 12 manage to characterise the partition equally, in that they both fulfill the condition of being larger than 7.

When testing, one could hasten the process whilst increasing test coverage by identifying such partitions of the input space, as the number of tests to be executed would be lowered. In an ideal setting, all partitions of the negative input space are identified, with the test set containing one candidate from each negative partition.

It is reasonable to assume that testers would like to be able to employ such partitioning techniques on QuickCheck automata in order to optimise the testing process, yet this may render the translation to runtime verification monitors ineffective unless it is implemented with certain considerations. When using QuickCheck automata, one may restrict the set of testable traces either by:

- **weakening the preconditions** in order to admit fewer traces
- **restricting the generator** so as to only produce and verify subsets of each partition

When weakening preconditions, one is losing information. For example, consider an arc within a QCFSA corresponding to function f. One could opt to set the precondition to only allow traces that call f with an input of 8, a representative value of the invalid partition, without losing coverage[3]. Yet the property would no longer be defined over the

[3]When testing, the odds of choosing a candidate from the partition under consideration would actually *decrease* unless the test set shrinks proportionally.

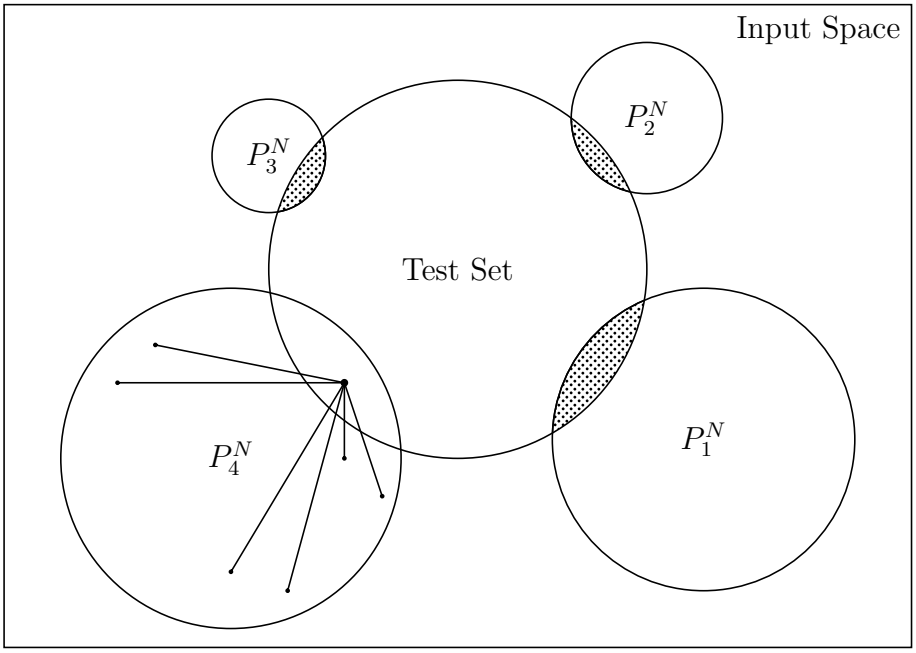

Figure 4.9: Example showing four partitions of the input space, and the test set. As each test case in a given partition is an equally valid candidate for uncovering that partition's associated bug, one can opt to only test a subset of each partition's intersection with the test set.

remainder of the partition, that is, it will not be able to classify the other elements of the partition as error-inducing inputs. While this would not normally affect testing, it would limit the detection capabilities of a derived runtime monitor. Using the previous example, a runtime monitor which only checks traces containing $f(8)$ would be far less capable than one which can produce a correct verdict for all $f(\mathbf{N})$ with $\mathbf{N} > 7$.

Pruning the search space at the generation stage whilst leaving the preconditions strong would result in automata that are far more comprehensive in their fault detection, as their testable trace set will not be compromised. It also keeps the automaton's concerns separate, allowing the generators to be as specific as necessary in order to restrict the test set whilst allowing the property to remain defined in general terms. Finally, restricting traces at the generation phase will improve performance by only generating traces which will be checked. In the previous example, setting a precondition to only allow calls with $f(8)$ would result in a slew of traces being generated and discarded until an appropriate trace is encountered. On the other hand, modifying the generator to immediately produce interesting traces would speed up testing, as they would instantly be of the correct form.

Generators in QuickCheck automata are typically used to seed the automaton's initial state. Finer-grained pruning of traces would thus require restrictions over the individual preconditions, yet this would limit the automaton's applicability to the runtime monitor translation due to the reasons described. Alternatively, one may opt to embed properties within an `?IMPLIES` construct (Section 2.5.3) so as to only consider traces which satisfy some global property, yet this would result in a loss of performance.

4.6.2 Bridging Generation and Monitoring

QCFSA properties should be written in such a way as to function within QuickCheck whilst also serving as a description from which an equivalent runtime monitor can be derived. The translation should not require that the QCFSA be modified or rendered inoperable in order to accommodate the transformation, as this would limit the applicability of the method. Ideally, the translation can accept any arbitrary QCFSA as an input and produce a consistent runtime monitor (the case study presented in Section 6.7 investigates the transformation of an arbitrary property defined by a third party).

When defining a QCFSA, one must specify a series of state transitions as functions, as described in Section 2.5.4. Every transition corresponds to a function call within the system under test. These functions can be passed values that are stored in the state data structure as arguments, and these values can be updated by executing the associated `next_state_data` function following a successful transition.

Listing 4.4: Transitions for a single state QCFSA

```
1 -record(state, {a :: term(), b :: term()}).
2
3 s(#state{a=A, b=B}) ->
4   [{s, {call, ?MODULE, f, [A]}},
5    {s, {call, ?MODULE, g, [B]}}].
```

Listing 4.4 specifies the transitions for a simple automaton that invokes two arbitrary functions `f` and `g`, both of which accept a single argument. In the example, each call is passed a value stored within the **#state** record. The actual values passed to the functions are determined based on the initial values of the state data and the definition of the `next_state_data` function. Using these two constructs, one may specify how and which values will be used within the generated traces.

In contrast, runtime monitors operate on live variable values that are resolved dynamically at runtime. With reference to the previous example, the values of the arguments of `f` and `g` are set by the process invoking them, irrespective of the automaton's state data. Thus, the translator ignores the bindings between the state data and arguments, and only uses the QCFSA transition structure to derive the relationship between states. This allows one to "hide" mechanisms that guide generation in the `next_state_data` function, with the generated values passed as arguments. As the state data values cannot be modified directly from within the tuple defining the transition, omitting the arguments will not lead to state-altering operations being left out[4].

For example, consider the case where one would like to test `f` using only odd natural numbers. One way of doing so would be to create a precondition which disallows calls to `f` with even numbers, yet this is inefficient and detrimental to the effectiveness of the runtime monitor.

[4]An exception to this would be if a transition is defined with one of the arguments consisting of a call to a function which alters the state of some server process, in which case the operation would not be performed at runtime. In general, code that has side-effects should not be embedded within an argument.

Listing 4.5: Testing `f` with odd-valued parameters

```
1 initial_state_data() ->
2   #state{a=1,b=0}.
3
4 next_state_data(s,s,D= #state{a=A},_,{call,?MODULE,Op,_}) ->
5   case Op of
6     f -> D#{a=(A+2)};
7     _ -> D
8   end.
```

Alternatively, one would opt to only generate odd-valued parameters. Listing 4.5 defines the initial values and the `next_state_data` function, which increments the **a** counter by 2 following every invocation of `f`. In this case, `f` is being tested with regularly-incremented values, yet this method allows full control over the form of the data that will be passed to the functions.

When translating to a runtime monitor, the resultant automaton always executes the `next_state_data` function as a state changing action once the associated transition completes successfully. In the example presented, the state data is being used exclusively for determining the argument values during the QCFSA's traversal in QuickCheck, and in theory need not be updated during runtime, as its values will not be used. Yet as the translation process must consider the general case, the function will still be invoked, so as to ensure that the automaton's intended behaviour is preserved.

4.7 Conclusion

The fundamental result of this chapter is the reconceptualisation of QCFSA functions as system events and the illustration of a method for transforming arbitrary QCFSAs into DATEs. The translation contains several inherent assumptions. The key assumption made is that all generated monitors are implicitly global, and by default consider that any relevant events generated by a system under test will be forwarded to that monitor. This assumption will not necessarily hold in the case of multiple concurrent monitors listening on a shared event stream if the original property does not itself cater for any event interleavings which can arise. Another assumption is that a QCFSA property is self-contained, that is it does not depend on the execution of any particular setup or initialisation functions prior to verification. External initialisation functions should be incorporated into the property, either within the `initial_state_data` function or by adding an explicit initialisation transition within the automaton.

While still sound, runtime monitors generated from properties with markedly restrictive preconditions may not be very effective. For runtime verification, preconditions should be as broad as possible, only discounting traces which are known to be unproblematic. QCFSAs should only attempt to direct searches by restricting the generation logic of the automaton, leaving the set of testable traces large enough to encompass any interesting

event sequences which a system may emit.

This chapter focused primarily on the translation's underlying theory, whereas the next chapter describes the implementation of the translation and monitoring framework. The analysis will also endeavour to address the issue of concurrent monitors through the use of several monitoring policies that attempt to introduce *context*.

Chapter 5

Runtime Monitoring in Erlang

5.1 Introduction

The rationale behind this project is the automated integration of testing and runtime verification. It is thus essential to demonstrate that the theoretical concepts presented earlier for transforming QCFSAs into DATEs are not confined to models, and can be used on the implemented property artefacts. In addition, translating QCFSAs into DATEs is of little use if the resultant automata cannot be verified. Thus, a runtime monitoring framework must also be developed to monitor DATEs in Erlang.

The following chapter details the implementation of a tool for automatically translating QCFSAs defined as Erlang modules into DATE scripts. The DATE script structure is then elucidated, and variations between the DATE notations used within this project and that of [Col08] are also identified. This is then followed by a description of the *E-LARVA* runtime monitoring framework developed for monitoring DATEs in Erlang. When presented with a DATE script and the source code of the system under test, E-LARVA interleaves the instrumentation code and generates all the necessary Erlang modules required for runtime monitoring, including one which implements the DATE monitor as a server process. The runtime monitoring framework's topology and use of *arbiter* processes are explained, along with a definition of the *object binding* (or *context*) problem and the mechanisms implemented to overcome it.

5.2 Overview

The translation of QCFSA properties into runtime monitors is carried out in two distinct and consecutive phases, namely the

1. **translation** phase, where an Erlang module defining a QCFSA is converted into a DATE script, and the
2. **generation** phase, where the modules implementing the runtime verification system are generated from a DATE script and the source code of the system under test.

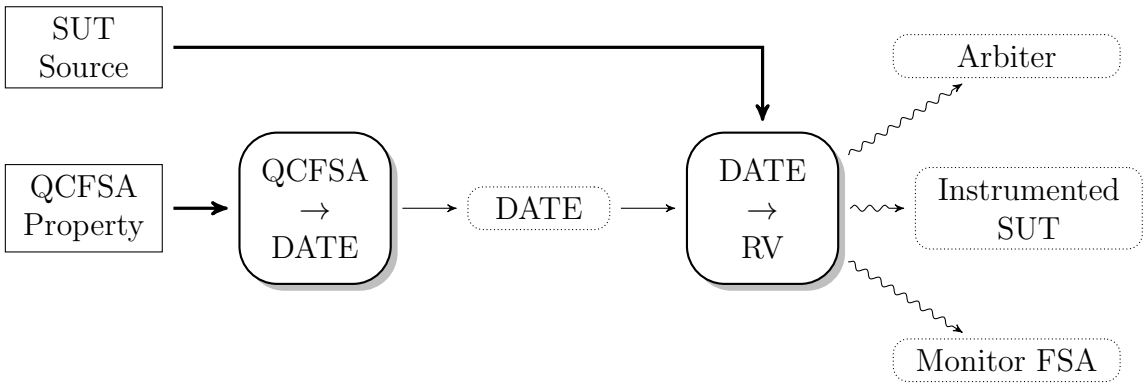

Figure 5.1: The translation process

Both phases are carried out using a tool developed in Java for this project. The translation phase implements the constructions defined in Chapter 4 for translating QCFSA models into DATEs. The generation phase serves a similar purpose as the LARVA [Col08] runtime verification tool, instrumenting the system under test and generating executable monitors from DATE properties. Thus, the generation tool will henceforth be referred to as *E-LARVA* whenever a distinction between the two phases has to be made.

Although this discussion focuses on monitoring properties which are initially expressed as QCFSAs, it should be emphasised that both phases can be performed independently of each other. The translation tool can be applied to a QCFSA to generate an equivalent E-LARVA DATE script, which could then be used as an input to other tools. Similarly, E-LARVA is designed to accept any valid E-LARVA DATE script, and does not limit itself to monitoring DATEs produced by the translation tool.

5.3 Translating Scripts

The following section describes the pertinent implementation details of a tool for converting Erlang modules defining QCFSAs into DATE scripts targeted for monitoring within E-LARVA. It also gives an overview of the blocks that constitute a DATE script, and highlights any differences between LARVA and E-LARVA script notations. Apart from a

textual description, the translator can also be configured to output a visualisation of the DATE automaton using *GraphViz*[1].

5.3.1 Variations Between LARVA and E-LARVA DATEs

The reference implementation of the LARVA runtime monitoring framework for DATEs described in [Col08] was designed for verification within a Java execution environment. Consequently, certain language-specific design choices had to be modified in order to facilitate monitoring within Erlang, as follows:

- Aside from the `GLOBAL` block, object-level contexts are not supported by Erlang DATEs. In testing, and especially runtime verification, it is often desirable to enforce properties for all instances of a given system entity or component [Col08]. For example, one would often want to verify statements such as "the memory footprint of an array's elements must not exceed its allocation" over all data structures of the type concerned, in this case, arrays. Through encapsulation, determining which entities are affected by a system event in an object-oriented environment is simplified. In contrast, keeping track of the system entities' progress within an environment lacking object-orientation, as in the case of Erlang, necessitates the use of additional monitoring information.

  Although the binding of monitors to individual system entities is not supported directly by the Erlang DATEs notation, Section 6.5 describes mechanisms implemented into E-LARVA for adding object tracking through enhanced instrumentation and logging, with Section 6.5 detailing several experiments gauging their efficacy.

- Since Erlang variables are single-assignment, global variables are not allowed. The omission of variables partially stems from the fact that Erlang lacks the notion of globally-scoped variables. Nevertheless, support for module-level constants could easily be added to an automaton. A straightforward way of doing this would be to bind each constant to the return value of a function. Accesses to the constants from within the code would then be replaced by calls to their respective function.

- Erlang DATEs do not support *timer events*, as the translation never produces them. Initial iterations of QuickCheck did not support the checking of temporal properties (other than event ordering, as dictated by the property's transitions), although recent versions[2] include a `temporal` module which allows the placement of time constraints on generated traces. Nevertheless, these properties exist outside the QCFSA structure and operate on a complete trace, making them less conducive to synthesis into runtime monitors.

[1] Project website: `http://www.graphviz.org/` (last accessed July 2011)
[2] Version 1.22

5.3.2 DATE Script Structure

The following is a description of the blocks making up DATE scripts, highlighting any differences between LARVA and E-LARVA script notations. Listing 5.1 is a complete example of an E-LARVA DATE script that defines the automaton illustrated in Figure 5.2. The example is derived from the light-switch controller QCFSA analysed in Section 2.5.4, and contains all of the essential DATE script blocks.

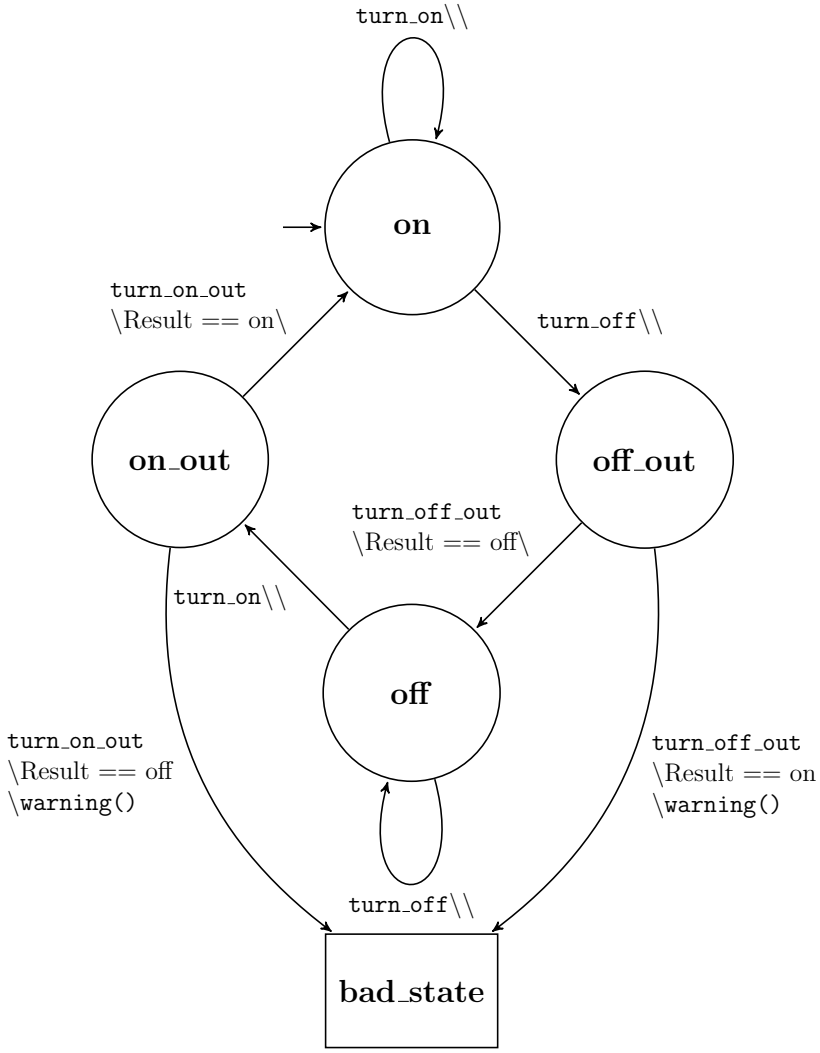

Figure 5.2: LARVA DATE automaton for the script defined in Listing 5.1. Arcs consist of event\condition\action triples.

The light switch is implemented by a module `light`, which exposes the `turn_on()` and `turn_off()` functions. Each of these functions returns the light's state after execution, this being represented by the atoms **on** and **off**. The DATE property ensures that calling either function will set the light to its correct state of illumination. So as to make the automaton clearer, transitions that loop onto the same state are not being verified, the precondition function's implementation has been moved directly into the arc's condition clause, and the idle state has been removed since the initial QCFSA was total. More

complex DATEs translated from QCFSAs can be seen in the case studies presented in Chapter 6.

IMPORTS

The `IMPORTS` block may contain one or more include directives. These directives will be inserted within the imports list of the system under test.

EVENTS

The events block maps functions implemented in the system under test to a local event name. Events may either be trigged on function *entry* or *exit*, the latter being triggered just after a function generates a result but before control is returned to the calling process.

Event definitions are of the form

$$\texttt{event\_name}() = \{\texttt{MODULE} : \texttt{func}(\textbf{arg1}, \textbf{arg2}, \dots)\}$$

where `func` is the name of the implemented function being monitored in `MODULE`. A function may take one or more arguments, the value of which will be determined at runtime. The function's arguments can be accessed from within a transition's condition and action fields using the reserved word **Args**, which is a variable to which the argument list is assigned. The use of the **Args** variable will be elaborated on further in the forthcoming exposition of the `TRANSITIONS` block.

To trigger an event on a function's exit, the `uponReturning(`**Result**`)` construct is used. Thus, an exit event would be of the form

$$\texttt{event\_name}(\textbf{Result}) = \{\texttt{MODULE} : \texttt{func}(\textbf{arg1}, \textbf{arg2}, \dots)\texttt{uponReturning}(\textbf{Result})\}$$

The function's result will be bound to the **Result** variable by the runtime verification engine, where **Result** is a reserved word used expressly for this purpose. For an exit event, the **Result** variable's value will be known and can be accessed from within a triggered transition's condition and action fields.

PROPERTY

A property is a block that defines an automaton which detects and classifies system states. It is formulated using two components, namely *states* and *transitions*. An automaton's states can be *accepting*, *bad*, *normal* or *starting*. A state's type is set by defining that state's name within the relevant type's block. Every state may also be assigned a *state action*, which is executed as soon as the automaton enters that state. For example, on entering the starting state defined on Line 27 of Listing 5.1, the system calls the `turn_on()` function to ensure a consistent starting state. State names must be unique throughout the state block, and cannot be of more than one type.

Listing 5.1: DATE script for verifying the light-switch controller

```
IMPORTS {
  -include_lib("eqc/include/eqc_fsm.hrl").
}

GLOBAL {
  EVENTS {
    turn_on() = {light:turn_on()}
    turn_on_out(Result) = {light:turn_on()uponReturning(Result)}
    turn_off() = {light:turn_off()}
    turn_off_out(Result) = {light:turn_off()uponReturning(Result)}
  }

  PROPERTY light {
    STATES {
      ACCEPTING {
      }
      BAD {
        bad_state {}
      }
      NORMAL {
        on_out {}
        off_out {}
        off {}
      }
      STARTING {
        on { light:turn_on() }
      }
    }

    TRANSITIONS {
      on -> on [turn_on \ \ ]
      on -> off_out [turn_off \ \  ]
      off_out -> off  [turn_off_out \ Result == off \ ]
      off_out -> bad_state [turn_off_out \ Result == on \ warning() ]
      off -> off [turn_off \ \ ]
      off -> on_out [turn_on \ \  ]
      on_out -> on  [turn_on_out \ Result == on \ ]
      on_out -> bad_state [turn_on_out \ Result == off \ warning() ]
    }
  }

  METHODS {
    warning() -> io:format("Light malfunction!").
  }
}
```

An automaton must define a single starting state. The other valid state types and their purpose are enlisted below:

Type	Purpose
Normal	Signifies a normal system state. No additional action is taken.
Accepting	Automaton is in a valid end state and can be terminated. This state type serves primarily as a hint for the underlying runtime verification system, and otherwise behaves identically to a normal state.
Bad	The automaton's property has been violated, and execution stops. Any associated state actions are called prior to the automaton's termination.

Transitions are defined within the `TRANSITIONS` block. The general form of a transition is given as

$$\mathbf{S} \rightarrow \mathbf{S'}\,[\mathbf{event}\backslash\mathbf{condition}\backslash\mathbf{action}]$$

which means that if the runtime monitor receives **event** when in state **S**, then the automaton will execute **action** and move to the next state $\mathbf{S'}$, provided that **condition** returns **true**. The condition field can be left out, in which case the default value of **true** is assumed. Similarly, omitting the **action** field will result in no additional state-changing operations being carried out prior to performing the transition. The **event** field must specify the event name, and takes no arguments.

The **condition** and **action** fields have access to the arguments with which the related event's function was invoked. Arguments are stored within a list **Args**, whose scope is local to that transition. In addition, events that are triggered on a function's exit (specified using the `uponReturning()` construct) will have the function's return value bound to the **Result** variable, which may also be accessed from within the condition and action blocks.

METHODS

The methods[3] block allows one to implement any necessary ancillary functions, which can then be invoked from within the automaton.

5.3.3 The Implemented Translation

Many elements within QCFSAs correspond directly to DATE script elements. Given that the data store and retrieve function `stateData()` operates as defined in Section 4.4.2, DATE script elements are built as follows:

[3]Erlang uses *functions*, not methods. The latter name was chosen so as to directly mirror its counterpart in Java DATE scripts.

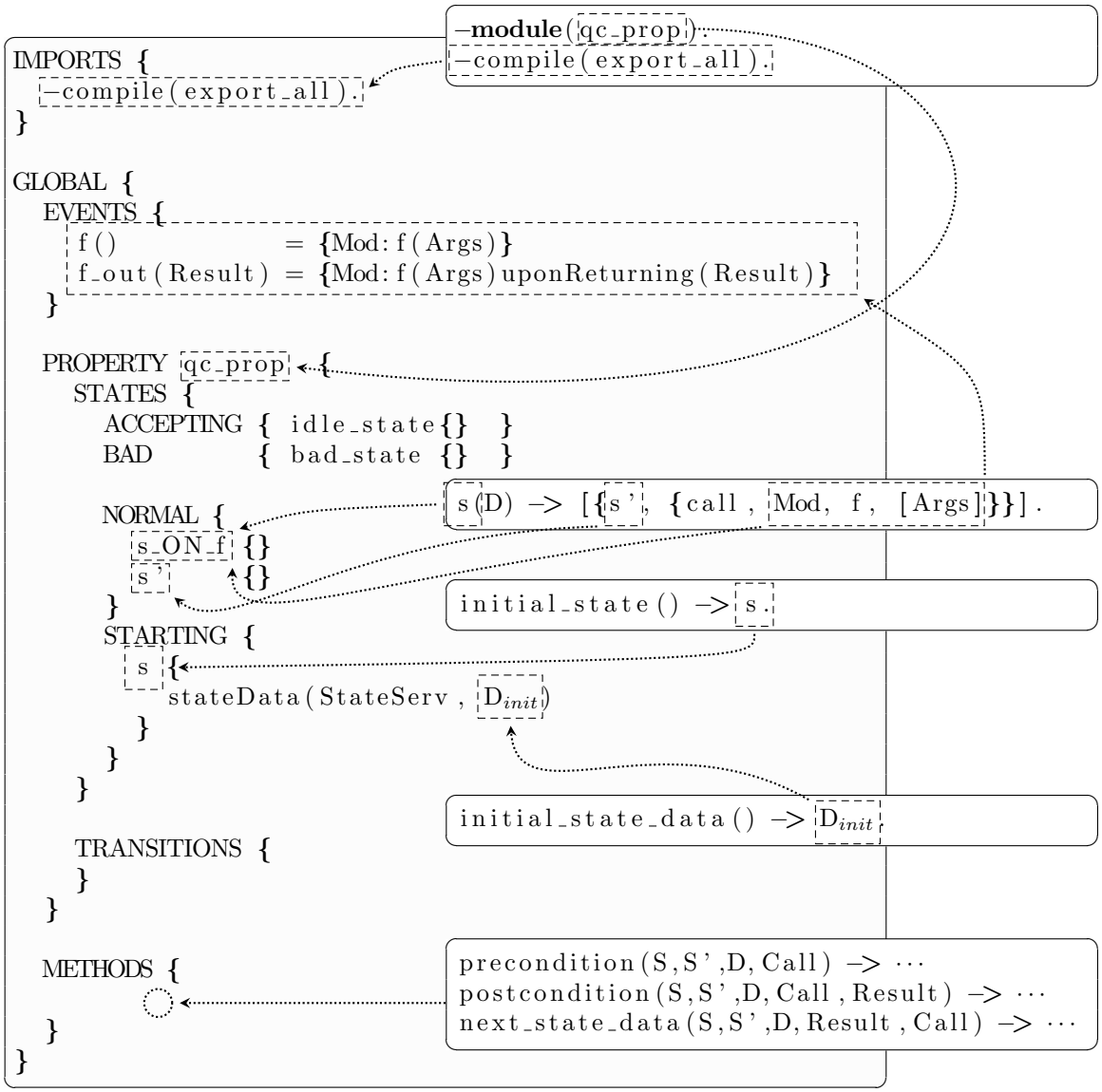

Figure 5.3: Mapping QCFSA script elements to their equivalents in DATEs (transition block is omitted)

- The *property name* is taken from the name of the module within which the QCFSA is defined.
- The *include* and *compiler directives list* is copied into the `IMPORTS` block.
- The list of events is extracted from the function call tuple defined for each transition. An entry and exit event is generated for every unique function and argument pair. The return value is assigned to **Result**.
- The *start state* is taken from the `initial_state()` function's return value.
- The *initial state data* is set from within the starting state's state-entry action. On entering the start state, the automaton invokes the `stateData` function with the

return value of the `initial_state_data()` function as an argument. An alternative design could invoke the `initial_state_data()` function directly from within `stateData`'s parameter list.

- The `precondition`, `postcondition` and `next_state_data` functions are copied into the `METHODS` block. Any additional functions defined within the QCFSA module that do not form part of the automaton structure are also copied.
- The `ACCEPTING` and `BAD` state blocks are automatically populated by the *idle* and *bad* states produced by the translation, respectively.
- Every state which appears within the right-hand side of a transition is added to the `NORMAL` state block.
- Intermediate states created when splitting events (Section 4.4.2) are automatically created for each outgoing arc in the QCFSA and are named using a combination of the source state and event names.

As described in Chapter 4, the initial QCFSA is first rendered total, after which transitions are split using intermediate states. The `TRANSITIONS` block is then populated by mapping each transition in the resultant QCFSA to the general DATE transition form described in Section 5.3.2.

5.4 Runtime Monitoring DATEs in Erlang

Given the source code of the system under test and the property being verified, the translator will automatically generate three artefacts, namely:

- The *instrumented system under test*, with instrumentation code inserted at the entry and exit point of each function. Events are sent to the runtime monitoring system using Erlang messages.
- An implementation of the DATE *monitor* as a server process which listens for events in the form of Erlang messages and updates its state as necessary.
- An *arbiter* program which mediates between the instrumentation points and the monitor processes.

On startup, named arbiter processes are spawned for every DATE property that must hold during the program's execution. For example, if a system is being monitored for the properties ϕ and φ, two arbiter processes ϕ_{ARB} and φ_{ARB} will be spawned, with each arbiter handling one of the property types. Which functions are instrumented depends on the traces with which the properties are concerned. Given that ϕ_f is the set of functions that appear at least once within any trace in ϕ's set of testable traces, every function in ϕ_f would forward its entry and exit events to ϕ_{ARB}, and the same would apply for property

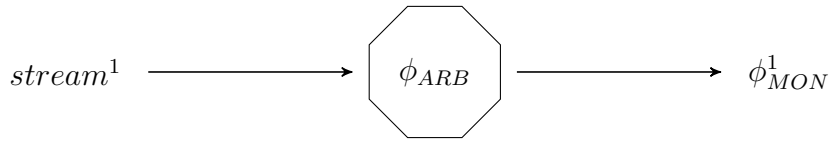

Figure 5.4: Arbiter mediating between events generated by the system under test and the property monitor

φ. Any functions in $\phi_f \cap \varphi_f$ would forward messages to both arbiters, although the system does not place any formal guarantees as to which arbiter would receive a message first.

It is conceivable that multiple instances of a given monitor type are required to execute concurrently. These monitors tend to be concerned with the verification of a system element of which multiple instances exist. For example, one may want to verify that no bounded buffers in use by the system will overflow. Such a property could be written in at least two ways. The first approach would be to create an automaton which receives events from all buffer access operations and keeps track of their counts using its private state data, yet this approach suffers from scalability issues. The second, more natural way of writing this property would be to create a monitor that verifies a single buffer, and subsequently spawn a separate monitor for each buffer in the system. The property's arbiter would serve as an exchange, allocating and maintaining a mapping of buffers to monitors, forwarding messages and spawning new monitors as required, whilst keeping monitors oblivious to concurrency. The instrumentation code will always communicate with the relevant arbiter and lets it manage event re-routing. The process of binding and mapping clients to monitors is intricate and property-dependent, and will be discussed in greater detail in Section 5.4.3.

Other than entity binding and message routing, the arbiter also discards events that arrive between an entry and exit event based on their unique identifier, as described in Section 4.4.2. This allows the monitor's code to be written to only accept the correct exploded QCFSA event stream without having to cater for intermediate actions. Finally, arbiters maintain logs of all the events that they receive, listing the event details such as type (entry or exit) and the associated function's arguments, if it has any. The logs also show whether events have been forwarded to the relevant monitors, whether new monitors have been spawned and the result of a monitor taking a transition.

5.4.1 Instrumentation

Functions are instrumented through the use of a trampoline function, as described in Section 4.4.2. To demonstrate, consider the following function to be monitored which, given a record of type **data**, returns the **s** element for **n** times.

```
1 -record(data, {s :: string(), n :: integer(), id :: integer()}).
2
3 repeat(Data = #data{s=S, n=N}) ->
4   lists:concat(lists:duplicate(N, S)).
```

When instrumented, the function is replaced with the following code:

```
1  -record(data, {s :: string(), n :: integer(), id :: integer()}).
2
3  date_repeat(Data = #data{s=S, n=N}) ->
4    lists:concat(lists:duplicate(N, S)).
5
6  repeat(P1 = Data = #data{s=S, n=N}) ->
7    ID = ?ARBITER:nextID(),
8    proceed=?ARBITER:send_event(#event{from=self(),
9          event=repeat,args=[P1],result=no_var,id=ID}),
10
11   DATE_Result = date_repeat(P1),
12
13   proceed=?ARBITER:send_event(#event{from=self(),
14         event=repeat_out,args=[P1],result=DATE_Result,id=ID+1}),
15   DATE_Result.
```

The instrumentation of a given function is performed through a series of steps, as follows:

1. The arbiter to which an event should be forwarded should be identified. In this case, `?ARBITER` refers to the module within which it is defined.
2. The original `repeat` function is renamed to another unique name by prepending `date_` to the function header, leaving the body unchanged. An instrumented function with the original name will be generated, within which a call to the renamed function will be performed. Using the instrumented trampoline function instead of instrumenting the original function directly avoids having to analyse the function's control flow and exit points and simplifies the collection of its result. Renaming the function is necessary, as otherwise the program's calls to the original function will not be monitored.
3. The new trampoline is created. Executing the trampoline will
   (a) Poll a server process exposed by the arbiter that will return a new, unique integer. Calls to the `nextID()` function will cause the server's stored value to be incremented by 2.
   (b) Forward the entry event to the arbiter using a transmit operation implemented within the arbiter module. The **#event** record contains all of the necessary instrumented values, namely the event name, originating process' PID, the arguments list with which the function was invoked and the event's unique identifier.
   (c) Invoke the original function with the original arguments whilst storing the result in an intermediate value.
   (d) Send an exit event to the arbiter once the function terminates. Unlike the first event transmission, this event will also contain the function's return value. As

described earlier, the exit result's ID is equal to the entry event's identifier, plus one.

(e) Return the original function's result so as not to break compatibility with other parts of the program that make use of the function.

When creating the instrumented trampoline function, it is important that the function's arguments are bound to temporary values within the function's signature, and that references to arguments within the instrumentation code (when setting the **#event** record's **args** field) are made using these new references.

Consider the following alternative implementation of the trampoline, where the use of the temporary variable **P1** is omitted, and instead references to the arguments are performed by simply copying the function's parameter list.

```
1  repeat(Data = #data{s=S, n=N}) ->
2    ID=?ARBITER:nextID(),
3    proceed=?ARBITER:send_event(#event{from=self(),
4          event=repeat,args=[Data=#data{s=S, n=N}],
5          result=no_var,id=ID}),
6
7    DATE_Result = date_repeat(Data=#data{s=S, n=N}),
8
9    proceed=?ARBITER:send_event(#event{from=self(),
10         event=repeat_out,args=[Data=#data{s=S, n=N}],
11         result=DATE_Result,id=ID+1}),
12   DATE_Result.
```

If the function were to be called with a **#data** record with all three fields defined (**s**, **n** and **id**), the system would bind **S** and **N** to the structure's value, and would bind the structure with which the function was called to **Data**. The problem arises in line 4 due to Erlang's single assignment mechanism. All three fields of **Data** are defined, yet it is being assigned the value of **#data**{s=**S**, n=**N**}. As the latter record does not specify a value for **id**, Erlang will give this field the default value of **undefined**, causing the function to fail. The use of temporary values avoids such issues by always making use of the original argument value with which the function was invoked. These variables are assigned as the leftmost term within the argument. It should be noted that one cannot simply replace the function's existing argument list with corresponding temporary variables, as this could damage the function's intended control flow by altering its pattern matching logic.

Emitting events from the SUT is a *blocking* operation. The SUT sends an event to an arbiter and waits until a **proceed** message is received. This simple two-way barrier synchronisation ensures that the automaton will always evolve in tandem with the system.

5.4.2 The DATE Monitor

The DATE monitor is implemented using a server process that listens for messages containing events generated by the instrumentation code that has been inlined into the system

under test. The next state to which the automaton will move will depend on the outgoing transition chosen.

Listing 5.2: Outline of a DATE Monitor. **State** contains the current automaton state, while **StateServ** is the PID of the state-data store.

```
1  event_server(StateServ, State) ->
2    case (lists:member(State, [bad_state])) of
3      true -> bad_state_reached;
4      false -> true
5    end,
6
7    receive
8      E=#event{event=Event, args=Args, result=Result} -> true
9    end,
10
11   ?ARBITER:log(...),                    % Log Automaton State
12
13   case {State, Event} of
14     {S, E} ->
15       case C of
16         true ->
17           NSDn = α,
18           ?ARBITER:send_object(NSDn),
19           event_server(StateServ,S');
20         false ->
21           true
22       end
23
24       ...
25       _ -> ?ARBITER:log(...)            % Log Unexpected Event
26   end.
```

Listing 5.2 is an outline of the general form of a monitoring process. Every transition (S, E, C, α, S') is translated into a case clause, as in lines 14 – 22. Since the arbiter filters out intermediate events, the monitor will always be able to determine a next state given an event, as the automaton will have been rendered total prior to its translation. **NSDn** will contain the new state returned by the state-changing action α, and the variable name is unique to each case clause. The `send_object` function serves to signal to the arbiter process that the SUT may proceed with its execution. The outline presented takes no action on detecting a bad state. Other than as a signal, the function serves to communicate back information to the arbiter. The precise mechanics of how this information is used will be described in greater detail in the following sections.

5.4.3 Object Binding

Within a monitoring framework operating in an object-oriented environment, as that described in [Col08], one has the facility to bind a property to an entire class of objects. In general, one may write a a property ϕ that verifies a given single instance σ of class Σ, and have the runtime monitor check that $\forall \sigma : \Sigma \cdot \phi(\sigma)$. The system would associate a monitor to every instance of Σ, and would ensure that ϕ holds for that object with its private state.

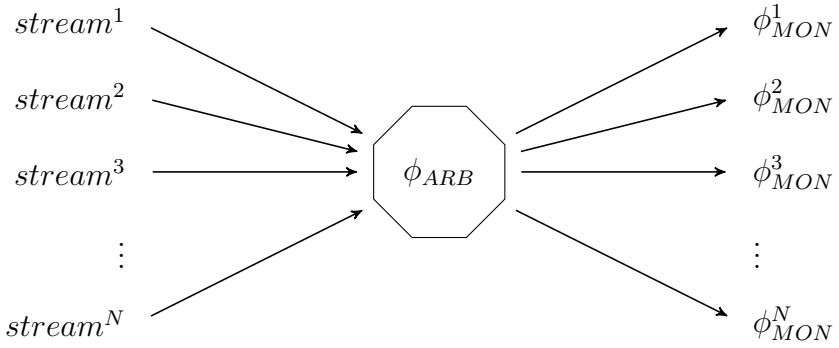

Figure 5.5: Multiple event streams to multiple monitors. Multiple streams may flow from a single process, and a single logical stream may have events that originate from multiple processes.

It would be desirable for such a mechanism to exist within the Erlang DATE monitoring framework, primarily for monitoring operations on *data structures*, yet Erlang is not object-oriented. Since operations are implemented as functions rather than methods, one can only relate a function to a data structure by passing the object being considered within the arguments list. Unlike instance methods, where the object on which the action is being performed is evident, the idea of a function as an operation relative to a data structure is conceptual and is not directly enforced by the Erlang system.

For example, consider the monitoring of a simple incrementing counter. Using object-oriented programming, one would define a *Counter* class which exposes an `increment()` method. An analogue in Erlang would be the creation of a **#counter** record and an `increment`:: **#counter** → **#counter** function that returns an updated version of the structure passed. In the former scenario, calling `increment()` would simply update the object's internal state. References to the object within the system would remain unchanged, and any handles or object identifiers would remain consistent after the state change. Conversely, the Erlang increment function does not explicitly specify the subject of the operation, and modifications to the data structure are not manifested as updates to the passed argument, rather as a newly returned structure. Keeping track of an object within the system over its lifetime thus requires additional information or meta-data.

For the arbiter to be able to bind unique monitors to unique objects, it is essential that a mechanism for *tracking* the objects over transformations is implemented. One effective way of achieving this would be to encapsulate each monitorable datum within a unique process and only allow modifications to the stored values through a well-defined interface. In essence, this would solve the tracking issue by giving data structures a unique identifier

that persists over transformations. Such an approach, while simple to manage, would be cumbersome to apply retroactively to a system that already exists, and would only be viable when writing programs from scratch.

To overcome these issues, the arbiter launches and maps new monitors to objects based on the variation of two policies, as follows:

One monitor per process from which an event has originated. On receiving an event from a process P, the arbiter launches and initialises an automaton P_M. Subsequent events from P that concern P_M are always sent to its associated monitor. This mode can be used in the case of properties that must hold at the process level, or in cases where the property being monitored is built to accept and handle events originating from coexisting data structures. The downside of this mode is that it does not allow multiple monitors of the same type to be bound to individual structures used within the same process.

Monitor for every new structure of a given type. The function headers in the system under test are augmented using *tags*, whereby the object variable is identified and tagged within the arguments list (this requires that a function header always has the object on which it is acting as one of the arguments). When sending an event, the system under test also sends the value of the tagged parameter. Alternatively, the arbiter must be able to deduce the bound object from the event's arguments list. When the arbiter receives an event from a process regarding an object for which it has not yet spawned a monitor, it will launch a monitor and register the mapping.

So as to track updates to the object, the `next_state_data` function must compute the object's new value and store it within a defined field within the automaton's state data. This value is then transmitted from the monitor back to the arbiter, which then updates the mapping of the object to the monitor. Transmitting **no_obj** to the arbiter signifies that the object mapping should not be updated for that event.

The point at which new monitors are spawned is based on two possible interpretations of the *for all* quantifier. The first interpretation prohibits an arbiter from spawning new monitors if it has received an entry event and is currently waiting for the matching exit event. For example, if a call to `increment` makes internal use of other **#counters** prior to returning, then these intermediate counters will not be tracked. The second interpretation will launch a monitor for any previously-unseen object of a given monitored data type, even if the event is emitted from within a nested call.

As will be seen in Chapter 6, object binding policies can only be applied successfully if the program's behaviour is known entirely. For example, arbiters cannot discriminate between streams using process IDs when actions on an object are carried out in separate processes. Object binding also assumes that several events forming a stream can be related to each other by a common data structure, and does not automatically isolate streams based on the logical relationship between their events.

5.5 Conclusion

This chapter has described the implementation of the property translation tool and the E-LARVA generation and runtime monitoring framework, which is designed to monitor any DATE script that conforms to its specification. The function of the elements making up a DATE script is described, and differences between DATEs as used within this project and LARVA outlined. The theory behind the addition of context to properties is discussed. In the next chapter, the translation and monitoring framework is examined and tested extensively through a case study, with an emphasis on the evaluation of E-LARVA's context binding mechanisms.

Chapter 6

Case Study

6.1 Introduction

The following chapter documents various experiments conducted on the implementation of the translation and monitoring framework outlined earlier. The aim of this exercise is to assert that the translation works within the specified parameters, to gauge the approach's effectiveness and to determine any pitfalls which may arise when moving from a formal construction to its concrete implementation. In particular, the case study analyses the issue of concurrently-executing monitors and attempts to introduce mechanisms for handling context. It also serves to determine whether the translation is general enough to accept any valid QCFSA as an input, and any considerations which have to be made if it is not.

The case study consists of the translation and verification of *four* QCFSA properties on *Riak*, an open-source distributed database written in Erlang. Each property attempts to analyse some aspect of the property creation and translation process, investigating the effects of event granularity, event interleavings, object binding and incoherence between an automaton's state data and the system's state.

6.2 Riak

The Riak[1] fault-tolerant distributed database was chosen as the system to be tested. Riak is an open-source project written in Erlang, and it is used by several large companies and organisations [Bas11]. Its implementation incorporates many different programming concepts, which offer varying scenarios for verifying data structures, control flow and interprocess communication. The following is an overview of Riak's core functionality and its high-level operations, continuing with a discussion of its topology and its mechanisms for robust data storage.

[1]Project website: `http://www.basho.com/` (last accessed July 2011)

6.2.1 Core Functionality

At a high level, Riak is a database which functions as an *associative array* (or *dictionary*), mapping keys to data items. Every data element stored within the database has a corresponding unique key. Given a key, the system will search for the element to which that key value is mapped, returning the object if it exists within the database. 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. This structure also contains other system-managed fields such as timestamps.

At its core, the system gives a connecting client the ability to

- **create** an object, associating a $\langle bucket, key \rangle$ with a value
- **insert** an object into the database
- **retrieve** or **delete** an object given a $\langle bucket, key \rangle$
- **update** or **extract** a Riak object's data value

Riak also provides additional functionality, such as the ability to list all the keys associated with a given bucket, or to list all registered buckets.

6.2.2 Topology

Riak differs from a simple dictionary in that it is designed to operate within a distributed environment. The system is hosted over one or more physical servers, or *nodes*, with the database's elements residing across these nodes. The distributed nature of the database is transparent to the accessing process, and requests to the database can be sent to any of the participating nodes, with Riak mediating communications between nodes. When a node receives a request to insert an object into the database, Riak must decide where the object will be stored. As Riak aims for fault tolerance through *replication*, multiple copies of the object are made over a number of nodes. Similarly, when a request to retrieve an item is received, the system must be able to determine which nodes host the object under consideration. Several system variables, such as the number of nodes to which objects must be replicated, can be fine-tuned depending on the underlying infrastructure and the system requirements.

The location of objects is resolved through the use of *hashing* and key-space *partitioning*. Each node is allocated a number of equally-sized and non-intersecting ranges, or *partitions*, of a 160-bit integer space, with the entire integer space being allocated. When performing a database operation, an object's bucket and key value pair is hashed to a value within the integer-space. If the resultant value falls within a given node's range,

then that node must handle the operation, storing or retrieving the object in question. The partitioning of the integer-space and the mapping of partitions to nodes is stored within a data structure known as a *ring*.

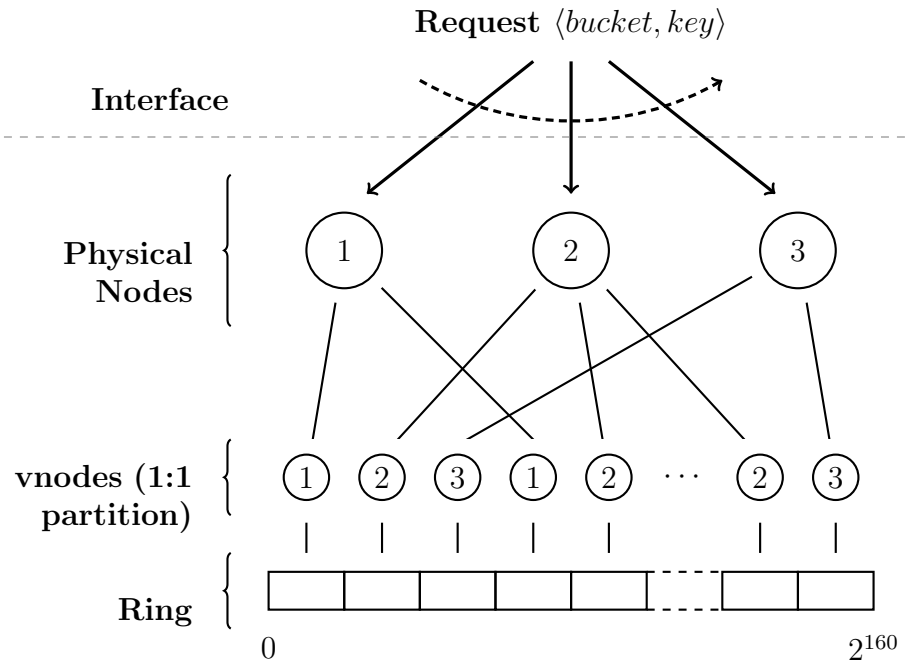

Figure 6.1: Riak topology for three nodes

Figure 6.1 illustrates the topology of a Riak cluster running over three physical nodes. A request can be sent to any of the three nodes. Ring partitions are not controlled directly by physical nodes. Instead, every partition is bound to a virtual node, or *vnode*, which is in turn owned by a physical node.

Nodes propagate their local view of the ring to other nodes via a *gossip* protocol [vRDGT08]. When a node changes its allocation of partitions over the ring, it communicates the local updated ring structure to other nodes in the cluster, which in turn reconcile their view with the new ring. Nodes also periodically advertise their ring so as to increase the probability of convergence to a globally-shared ring view.

Ideally, partitions should be distributed evenly amongst physical nodes, with consecutive partitions being owned by different nodes. Thus for **N** nodes, partitions should be spaced with a periodicity of **N**, that is, partitions belonging to the same node should be distanced by **N-1** intermediate partitions. In general,

$$\text{vnodes/node} \approx \frac{\#\text{ partitions}}{\#\text{ nodes}}$$

6.2.3 Replication

As mentioned previously, Riak attempts to provide fault tolerance through replication. The degree of replication is controlled by the **N-value**, which specifies the number of

partitions to which an object must be written. The **N-value** is set at either the node or bucket level. If the ring is evenly partitioned over a multi-node cluster, then the system's resilience to failing nodes will increase, as there will still be multiple copies of an object available should some of the nodes go offline.

Whenever a node executes a read or write request, it reports the operation's result back to the process managing that transaction. This allows database operations to specify a minimum number of reported successes for the operation to be considered as having executed correctly. For insert operations, this is known as the **W-value**. Similarly, fetch operations may specify a minimum **R-value**. Subtracting the **W-value** or **R-value** from the **N-value** gives the number of nodes that can fail before the operation becomes unreliable.

Operation	Value Type	Tolerance
Get (Read)	**R**	**N−R**
Put (Write)	**W**	**N−W**

Table 6.1: Value types and related operation. Tolerance refers to the number of nodes which can fail before the transition fails.

Once nodes receive and process a write request, they commit the final values to permanent storage. This operation also results in a reply being generated to the transmitting process. Thus, insert operations may specify an additional *durable-write* threshold, or **DW-value**, to set the minimum number of reported successful commits.

6.3 Translated Properties

The following section details the translation and performance of four QCFSA automata into equivalent DATE monitors. Each automaton attempts to gauge the translation's effectiveness under different scenarios, with a complete analysis of the generated results presented in Chapter 7. To avoid cluttering automata unnecessarily, any DATEs described forego the splitting of self-looping transitions on the idle state. This will not affect the monitor's set of testable traces or its classification capabilities.

Different modes of object tracking and variations on the approaches presented will be analysed. With object tracking disabled, the system defaults to launching a monitor per individual process. Alternatively, the arbiter can be set to maintain only one automaton for every property type, and forward all of the pertinent events to it irrespective of the point of origin.

The choice of policy depends greatly on the executing program's behaviour. For example, certain procedures in Riak execute individual steps within separate processes. An automaton describing such a procedure must be set to receive events from multiple instrumentation points within different processes[2]. On the other hand, maintaining a monitor

[2]If events originate from multiple processes, then the guarantees on event ordering no longer hold. To ensure correct monitoring, the system under test should avoid calling monitored functions pertaining to

for every separate process could allow coexisting processes to each be monitored. Such a scenario would be particularly common when monitoring systems that allow multiple concurrent connections (as does Riak) managed by separate dedicated processes.

All test cases were executed on Riak running across three Erlang nodes (**dev1**, **dev2** and **dev3**) hosted within the same physical system (**localhost** or 127.0.0.1). All Riak operations performed during the test are initiated from **dev1**.

6.4 Case 1: Coarse-grained Operations

Analyses: basic monitoring scenario, application of high-level Riak operations

6.4.1 Property Description

The first property serves primarily as a sanity test, and investigates the permanence of Riak objects within the database by repeatedly inserting and retrieving objects and confirming their consistency. Operations are performed using two high level functions, `put2(Obj)` and `get2(Bucket, Key)`, which connect to the database and invoke `put` and `get`, respectively. 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. Section 7.2.1 exposes a more complete analysis of the issue.

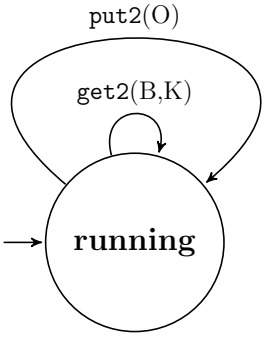

Figure 6.2: QCFSA for coarse-grained object insertion and retrieval

Figure 6.2 is a single-state QuickCheck automaton that generates sequences of insertion and retrieval operations (Listing 6.1 describes the complete automaton). The automaton keeps track of which objects have already been inserted through the use of its private *state data*, and ensures that fetch operations for objects that have previously been inserted will succeed. The property will only fail when an attempt to fetch an object

the same monitor from within separate processes, or alternatively, it should enforce a strict ordering on event generation.

which has been inserted into the system returns a different object. In essence, the property keeps track of insertion and retrieval operations on a simple, local data store model and ensures that operations are mirrored correctly by the database.

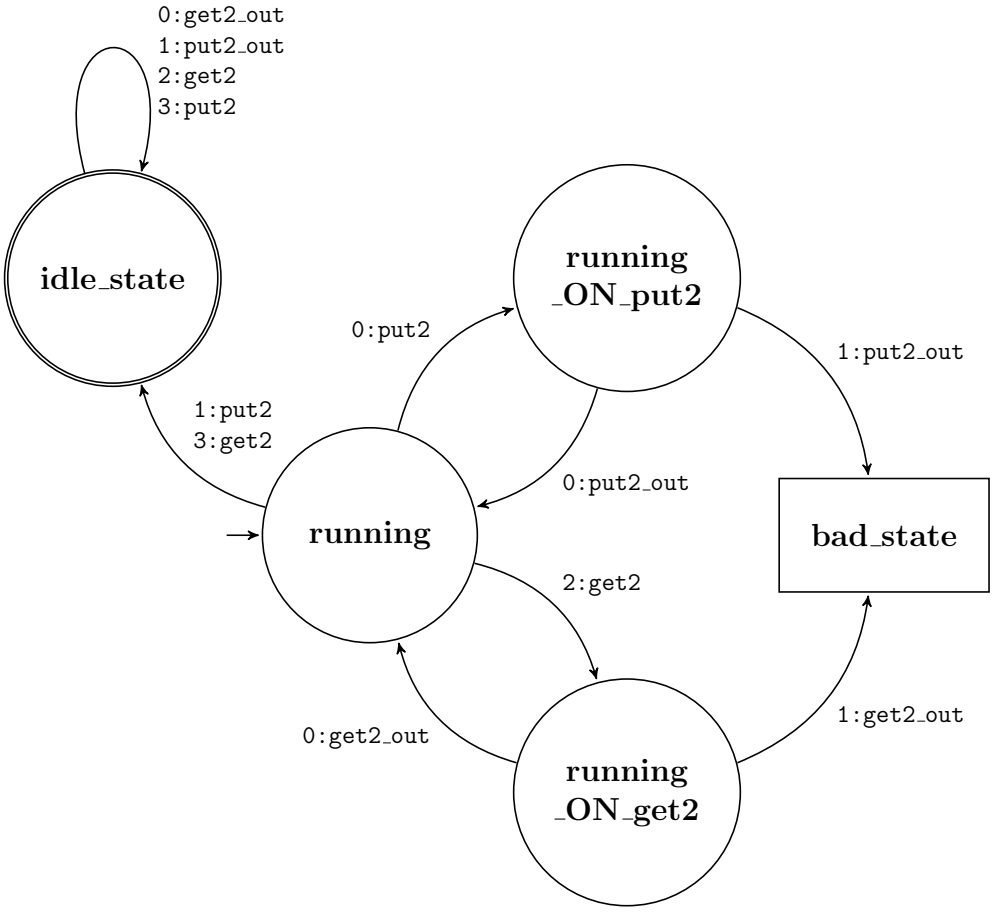

Figure 6.3: DATE for coarse-grained object insertion and retrieval. The index accompanying each event specifies the order in which transitions are checked. This ranking is derived from the order in which transition declarations appear in the DATE script [Col08].

The automaton maintains two fields within its state data record, the **objects** dictionary and **gen_obs** list. The former is used to maintain a list of keys that have already been bound, and is used to ensure that if the object was inserted, then it must be returned by a fetch operation with that object's bucket and key pair. The latter specifies a pool of pre-generated Riak database objects which will be used as arguments for the Riak operations. As described in Section 4.6.2, this keeps the generation aspect of the automaton separate from its functions as a recogniser, which facilitates the translation process.

When operating as a QuickCheck automaton, it is necessary that the pool of objects be pre-determined rather than built during traversal, so as to ensure that the preconditions and state data transformations produce matching results during the generation and execution passes (Section 2.5.4). Naturally, hard-coding initial state conditions may be limiting, as it restricts the set of values with which the tests can be run. The case study

Listing 6.1: QCFSA property for coarse-grained operations

```
-module(kv_putcoarse).
-include_lib("eqc/include/eqc_fsm.hrl").
-compile(export_all).

-define(SUT, riak_client).
-record(state,    {objects         :: term(),  gen_obs     :: list()}).
-record(r_object, {bucket          :: term(),  key         :: term(),
                   contents        :: term(),  vclock      :: term(),
                   updatemetadata  :: term(),  updatevalue :: term()}).

put2(O) ->   {ok, C} = riak:client_connect('dev1@127.0.0.1'), C:put(O).
get2(B,K) -> {ok, C} = riak:client_connect('dev1@127.0.0.1'), C:get(B, K).

initial_state() -> running.

initial_state_data() ->
  #state{objects=dict:new(),
    gen_obs=[riak_object:new(<<"testbucket">>, <<"testkey1">>, test1),
             riak_object:new(<<"testbucket">>, <<"testkey2">>, test2),
             riak_object:new(<<"testbucket">>, <<"testkey3">>, test3),
             riak_object:new(<<"testbucket">>, <<"testkey4">>, test4)]}.

running(#state{gen_obs=[#r_object{bucket=B, key=K}=G|_]}) ->
  [ {running, {call, ?MODULE, put2, [G]}},
    {running, {call, ?MODULE, get2, [B, K]}}].

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

postcondition(running,running,#state{objects=Os},{call,_,Op,Args},Res) ->
  case Op of
    put2 -> true;
    get2 ->
      [B, K] = Args,
      Key= {B,K},
      {Code, Obj} = Res,
      case dict:is_key(Key, Os) of
        true  -> (Code == ok) and
                 (dict:fetch(Key, Os) ==
                  riak_object:get_value(Obj));
        false -> true % may have been added through some other process
      end
  end.

next_state_data(_,_,D=#state{objects=Os,gen_obs=Gs},_,{call,_,Op,Args}) ->
  case Op of
    put2 ->
      [New = #r_object{bucket=Bucket, key=Key}] = Args,
      [G|Gen] = Gs,
      D#state{objects=dict:store({Bucket,Key},
                riak_object:get_value(New), Os),
                gen_obs=(Gen++[G])};
    get2 -> D
  end.
```

presented in Section 6.6 avoids setting fixed initial conditions within the automaton. Recall that QCFSA automata are used as arguments to a generator function `commands()`, which is in turn used within a QuickCheck property. In addition to a QCFSA, the `commands()` generator can also accept a list of arguments to be passed to the automaton's `initial_state_data` function. This allows object pools to be generated using QuickCheck generators, with the results being used as initial conditions to an automaton. The advantages of the latter approach are numerous, as will be discussed in Section 7.2.2.

6.4.2 Operation and Results

Listing 6.2 shows the DATE derived from the QCFSA property. Although the QCFSA consists of merely one state and two transitions, the blow-up experienced under transformation is appreciable. As mentioned earlier, the number of states in an automaton would not in itself affect the performance of runtime monitoring, as the monitor is employed as a recogniser and the system does not attempt to explore the state space in advance. On the other hand, the out-degree of a node will determine the number of comparisons that the system would have to perform in order to choose the next state (at worst, all the outgoing transitions would have to be checked).

Figure 6.4 is a sequence diagram showing the communication that occurs between the monitoring entities when insert and retrieve operations are executed consecutively. The monitor spawning policy is to bind a new monitor to every unique PID from which events originate. The original call to `put2` is passed a Riak object **O**. The first call inserts a Riak object **O** into the system, whereas the second call attempts to retrieve an inexistent object at the bucket and key location $\langle \ll$ "b" $\gg, \ll$ "notexist" $\gg \rangle$[3], which returns an error. Events from the System Under Test to the arbiter are uniquely numbered, and the arbiter only forwards paired entry and exit events to the monitor. Entry events have an unknown return value **no_var**, as the function will not yet have executed.

On the first entry event, the arbiter spawns a new monitor to handle subsequent events that originate from the System Under Test (SUT). Calls from the SUT to the arbiter process are *blocking*, and the program will only continue with its execution once a **proceed** message originating from the arbiter is received. On being spawned, the monitor is automatically initialised to its correct starting state as dictated by the property. Likewise, its state data server process is launched and assigned the monitor's initial state data. If the object's inexistence within the system were to signify a system failure, then the final exit event would have placed the automaton into a bad state.

As formulated, the precondition function always evaluates to true, thus allowing the automaton to monitor any interleaving of `get2` and `put2` operations. The property only checks that a `get2` operation retrieves the correct object if the object being fetched has been inserted during the monitor's lifetime. This is because the automaton stores the inserted object list within its own private data. Thus, the system will not check the validity of objects that have been inserted by another process or by the same process

[3]The $\ll \gg$ operator converts the given value into its binary representation. Buckets and keys must be binary values.

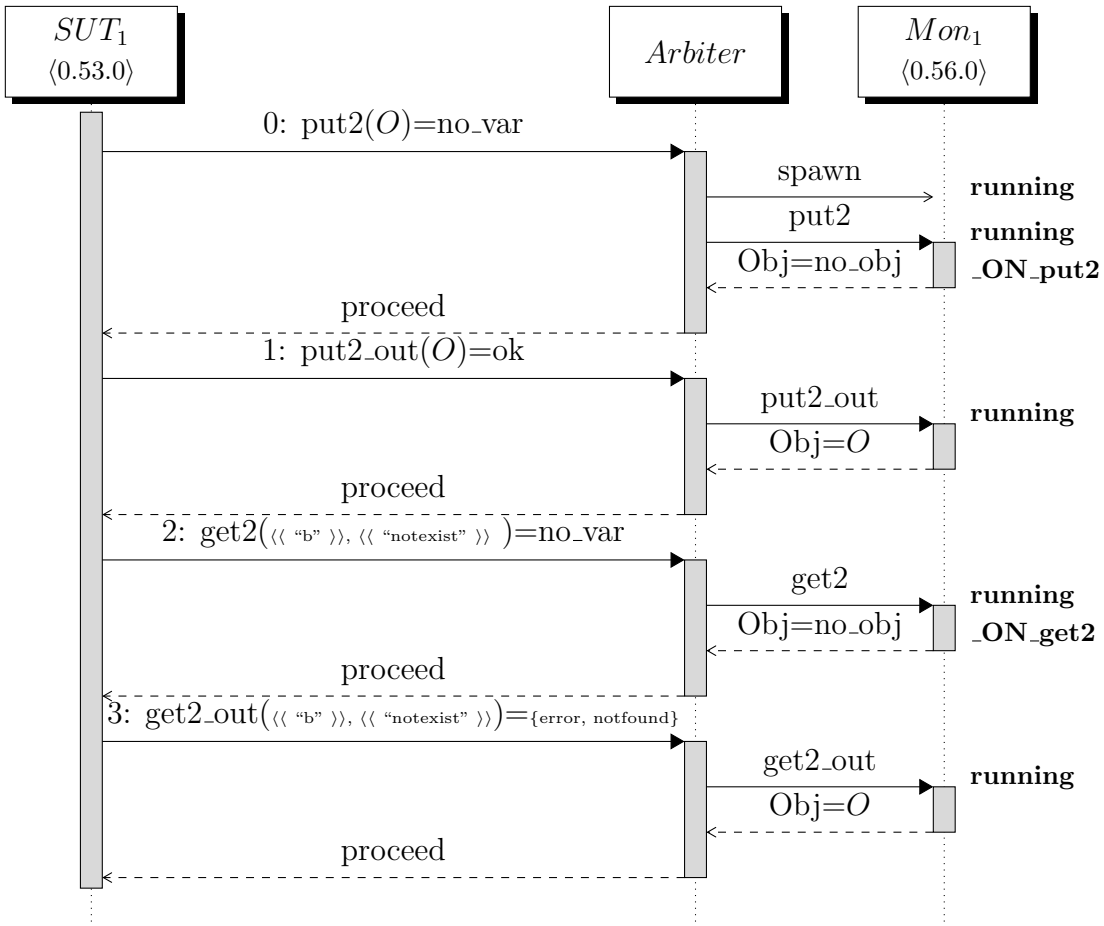

Figure 6.4: Sequence diagram for `put()` followed by a `get()` for a non-existing object

prior to the commencement of runtime monitoring. Ensuring that the model of the data store is coherent across processes would require that operations performed from any process also operate on a shared data store. This would complicate the data store model, introducing concurrency and ordering issues, which could in turn lead to the monitor malfunctioning as a classifier by allowing unwanted behaviours to remain undetected. In addition, the property assumes that objects inserted in the system are immutable, and can only be updated through a fresh call to `put2`. This assumption would have to be revised if updates are to be allowed, as the process would not be atomic.

Listing 6.2: DATE script for coarse-grained operations

```
IMPORTS {
  -compile(export_all).
}

GLOBAL {
  EVENTS {
    put2() = {kv_putcoarse:put2(A)}
    get2() = {kv_putcoarse:get2(A,B)}
    get2_out(Result) = {kv_putcoarse:get2(A,B)uponReturning(Result)}
    put2_out(Result) = {kv_putcoarse:put2(A)uponReturning(Result)}
  }

  PROPERTY pc {
    STATES {
      ACCEPTING {
        idle_state {}
      }
      BAD {
        bad_state {}
      }
      NORMAL {
        running_ON_get2 {}
        running_ON_put2 {}
      }
      STARTING {
        running {
          stateData(StateServ,
          #state{objects=dict:new(), gen_obs=[
            riak_object:new(<<"testbucket">>, <<"testkey1">>, test1),
            riak_object:new(<<"testbucket">>, <<"testkey2"test2),
            riak_object:new(<<"testbucket">>, <<"testkey3"test3),
            riak_object:new(<<"testbucket">>, <<"testkey4">>, test4)]})
        }
      }
    }

    TRANSITIONS {
      running -> running_ON_put2 [put2 \
          precondition(running,running,stateData(StateServ),
                       {call, kv_putcoarse, put2, Args}) \ ]
      running_ON_put2 -> running [put2_out \
          postcondition(running,running,stateData(StateServ),
                       {call, kv_putcoarse, put2, Args}, Result) \
          stateData(StateServ, next_state_data(running,running,
                       stateData(StateServ), Result,
                       {call, kv_putcoarse, put2, Args}))]
      running_ON_put2 -> bad_state [put2_out \
          !postcondition(running,running,stateData(StateServ),
                       {call, kv_putcoarse, put2, Args}, Result) \ ]
      running -> idle_state [put2 \
          !precondition(running,running,stateData(StateServ),
                       {call, kv_putcoarse, put2, Args}) \ ]
      running -> running_ON_get2 [get2 \
          precondition(running,running,stateData(StateServ),
```

```
55                          {call, kv_putcoarse, get2, Args}) \ ]
56        running_ON_get2 -> running [get2_out \
57            postcondition(running,running,stateData(StateServ),
58                          {call, kv_putcoarse, get2, Args}, Result) \
59            stateData(StateServ, next_state_data(running,running,
60                          stateData(StateServ), Result,
61                          {call, kv_putcoarse, get2, Args}))]
62        running_ON_get2 -> bad_state [get2_out \
63            !postcondition(running,running,stateData(StateServ),
64                          {call, kv_putcoarse, get2, Args}, Result) \ ]
65        running -> idle_state [get2 \
66            !precondition(running,running,stateData(StateServ),
67                          {call, kv_putcoarse, get2, Args}) \ ]
68        idle_state -> idle_state [get2_out \  \ ]
69        idle_state -> idle_state [put2_out \  \ ]
70        idle_state -> idle_state [get2 \  \ ]
71        idle_state -> idle_state [put2 \  \ ]
72      }
73    }
74
75    METHODS {
76      precondition(_,_,_,_) -> true.
77
78      postcondition(running, running,#state{objects=Obs},
79                    {call, _, Op, Args}, Result) ->
80        case Op of
81          put2 ->
82              true;
83          get2 ->
84              [B, K] = Args,
85              Key= {B,K},
86              {Code, Obj} = Result,
87              case dict:is_key(Key, Obs) of
88                true  -> (Code == ok) and (dict:fetch(Key, Obs) ==
89                                           riak_object:get_value(Obj));
90                false -> true
91              end
92        end.
93
94      next_state_data(_, _, D=#state{objects=Obs, gen_obs=Gens}, _,
95              {call, _, Op, Args}) ->
96        case Op of
97          put2 ->
98              [New = #r_object{bucket=Bucket, key=Key}] = Args,
99              [G|Gen] = Gens,
100             D#state{objects=dict:store({Bucket,Key},
101                     riak_object:get_value(New), Obs), gen_obs=(Gen++[G])};
102         get2 -> D
103       end.
104
105     put2(O) -> {ok,C} = riak:client_connect('dev1@127.0.0.1'),C:put(O).
106
107     get2(B,K) -> {ok,C} = riak:client_connect('dev1@127.0.0.1'),C:get(B, K).
108   }
109 }
```

6.5 Case 2: Vector Clocks

Analyses: monitoring data structures, object binding, single threaded monitoring

6.5.1 Property Description

Vector clocks (or *vclocks*) are used in Riak as part of the mechanism that enforces coherence between objects residing over different nodes [Bas11]. They allow the creation of a partial ordering amongst object update operations by 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*.

The property under investigation 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.
2. Merging v with another vclock will result in a vclock that descends from v.

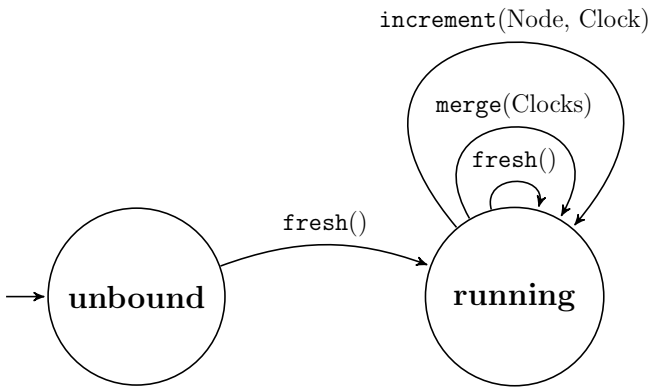

Figure 6.5: Vclock QCFSA property

Figure 6.5 describes the QCFSA of the property being checked. The property is designed to evaluate the different monitor spawning and binding policies described earlier. The automaton starts from an initial **unbound** state and moves to a **running** state on creating a new vclock through `fresh()`[4]. The vclock returned by the latter operation is taken to be the object to which the automaton is bound. Vclocks created after the first vclock has been bound are added to the **clocks** list in the automaton's state data.

Apart from creating new clocks, the automaton will either `increment` or `merge` clocks from the vclock pool. Clocks are chosen from the head of the vclock list, which is shuffled after every operation. The shuffling operation uses known seed values for its randomiser

[4]New vclocks are initialised as an empty list [].

so that the trace generation and execution phases do not produce different orderings. The `merge` operation merges the first two clocks in a list, and the `increment` operation increments the top vclock using a random node name from a fixed list of dummy nodes. In both cases, the original vclocks are removed from the clock pool and the resultant vclock is added.

Listing 6.3: Vector clock QCFSA

```
-module(vclockfsm).
-compile(export_all).
-define(SUT, vclock).

-record(state, {obj     :: vclock(),          clocks :: [vclock()],
                nodes   :: [vclock_node()], seed   :: term()}).

initial_state() ->
  unbound.

initial_state_data() ->
  #state{obj=[], clocks=[], nodes=[a,b,c,d,e], seed=random:seed0()}.

unbound(_D) ->
  [{running, {call, ?SUT, fresh, []}}].

running(_D=#state{clocks=[C|_]=Cl, nodes=[N|_]}) ->
  [{running, {call, ?SUT, fresh, []}},
   {running, {call, ?SUT, increment, [N, C]}},
   {running, {call, ?SUT, merge, [lists:sublist(Cl, 2)]}}].

% Pre and post conditions
precondition(unbound,running,_,_) ->
  true;
precondition(running,running,_D=#state{clocks=Cs},{call,?SUT,Op,Args}) ->
  case Op of
    fresh ->
      true;
    increment ->
      [_Node, Clock] = Args,
      lists:member(Clock, Cs);
    merge ->
      [Ar2] = Args,
      (lists:usort(Ar2++Cs) == lists:usort(Cs))
  end.

postcondition(unbound,running,_,_,C) ->
  C == [];
postcondition(running,running,_D=#state{obj=Obj},{call,?SUT,Op,Args},Res)->
  case Op of
    fresh ->
      (Obj == []) or not ?SUT:descends(Res, Obj);
    increment ->
      [_Node, Vclock] = Args,
      ?SUT:descends(Res, Vclock);
    merge ->
      [A1] = Args,
```

```
48        lists:all(fun({Rz, E}) -> ?SUT:descends(Rz, E) end,
49                  [{Res, A} || A <- A1])
50    end.
51
52  next_state_data(unbound,running,D,Res,_) ->
53    D#state{obj=Res, clocks=[Res]};
54
55  next_state_data(running,running,D=#state{obj=Obj, clocks=Cs,
56                          nodes=Ns, seed=Seed}, Res, {call, ?SUT, Op, Args}) ->
57    case Op of
58      fresh ->
59        Cprime = Cs ++ [Res],
60        {S1, NewCs, NewNs} = shuffle(Cprime, Ns, Seed),
61        D#state{seed=S1, clocks=NewCs, nodes=NewNs};
62
63      increment ->
64        [_Node, Clock] = Args,
65        NewClocks= (Cs -- [Clock]) ++ [Res],
66        {S1, NewCs, NewNs} = shuffle(NewClocks, Ns, Seed),
67        case Clock of
68          Obj -> D#state{seed=S1, clocks=NewCs, nodes=NewNs, obj=Res};
69          _   -> D#state{seed=S1, clocks=NewCs, nodes=NewNs}
70        end;
71
72      merge ->
73        [A1] = Args,
74        Unmerged = lists:sublist(Cs, length(A1) + 1, length(Cs)),
75        Complete = Unmerged ++ [Res],
76        {S1, NewCs, NewNs} = shuffle(Complete, Ns, Seed),
77        case lists:member(Obj, A1) of
78          true  -> D#state{seed=S1, clocks=NewCs, nodes=NewNs, obj=Res};
79          _     -> D#state{seed=S1, clocks=NewCs, nodes=NewNs}
80        end
81    end.
```

At any one time, the system can make use of multiple vclocks, making the property a good candidate for deployment with object[5] tracking. As a vclock's lifetime begins with an invocation to `fresh()`, calls to it trigger the creation of new monitors. A potential issue arises due to the fact that events tied to `fresh()` can appear anywhere within a trace, due to the existence of the self-loop on the **running** state. Thus, the system must decide whether or not to spawn a new monitor with every call to fresh, and which monitor should receive subsequent events. This is further complicated by the fact that `fresh()` takes no arguments and exists as an independent action, not being inherently related to any single object's lifetime other than that which it returns as a result of its execution.

6.5.2 Operation and Results

The following tests evaluate the application of the different monitor creation and binding policies described in Section 5.4.3. In all cases, the runtime monitor was deployed in a live

[5]The term *object* is being used generically to refer to a system element, in this case a vclock data structure.

system. Thus, the solution chosen also has to cater for events fired by the vclocks used internally by the system. Object tracking is managed by the `next_state_data` function, with the initial object's value being the return value of `fresh()`.

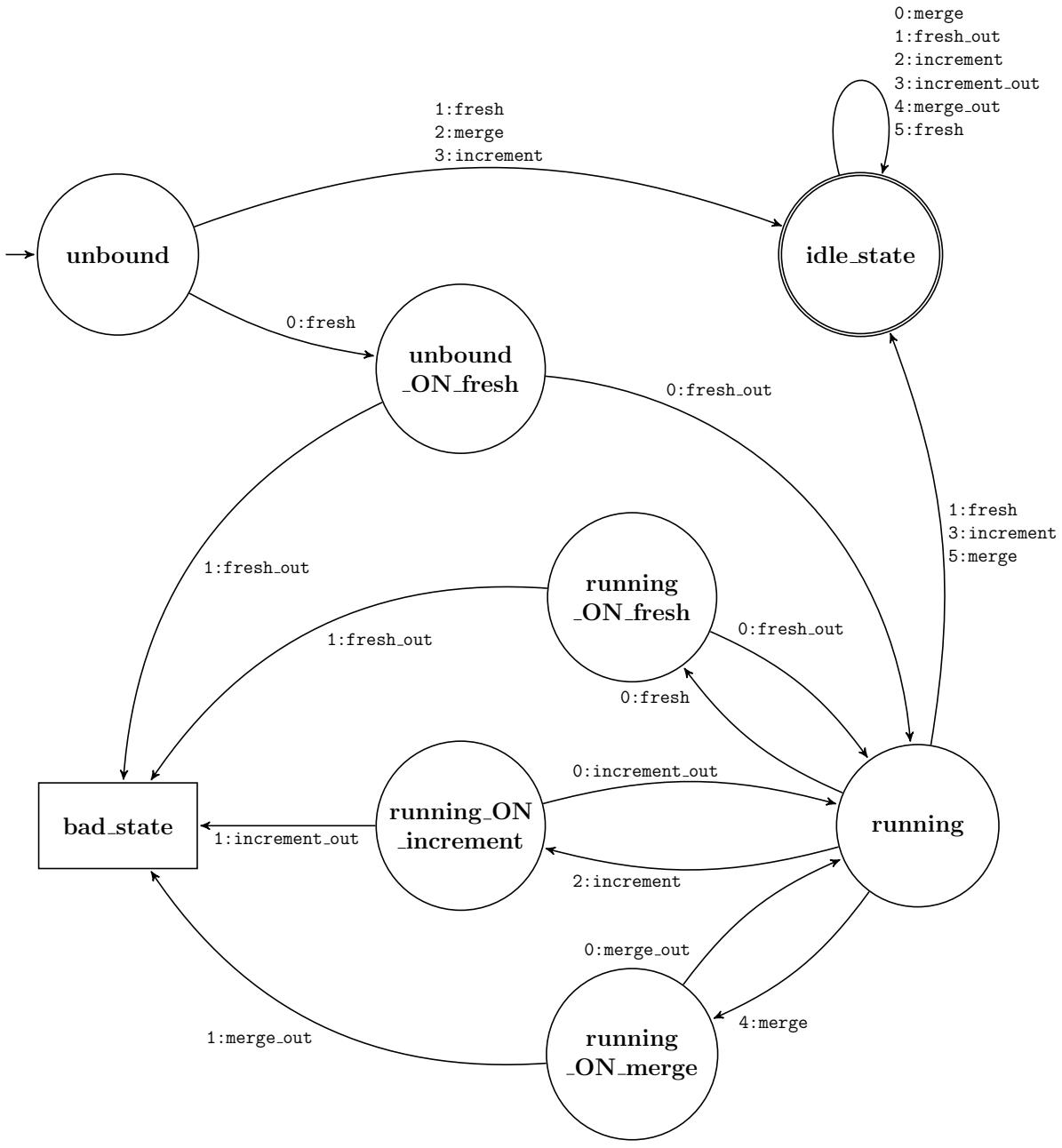

Figure 6.6: DATE monitoring vclock operations

Single Monitor, No Object Tracking

On receiving the first event from process SUT_1, the arbiter is set to launch a monitor and bind it to that process. Events that originate from different processes are not forwarded.

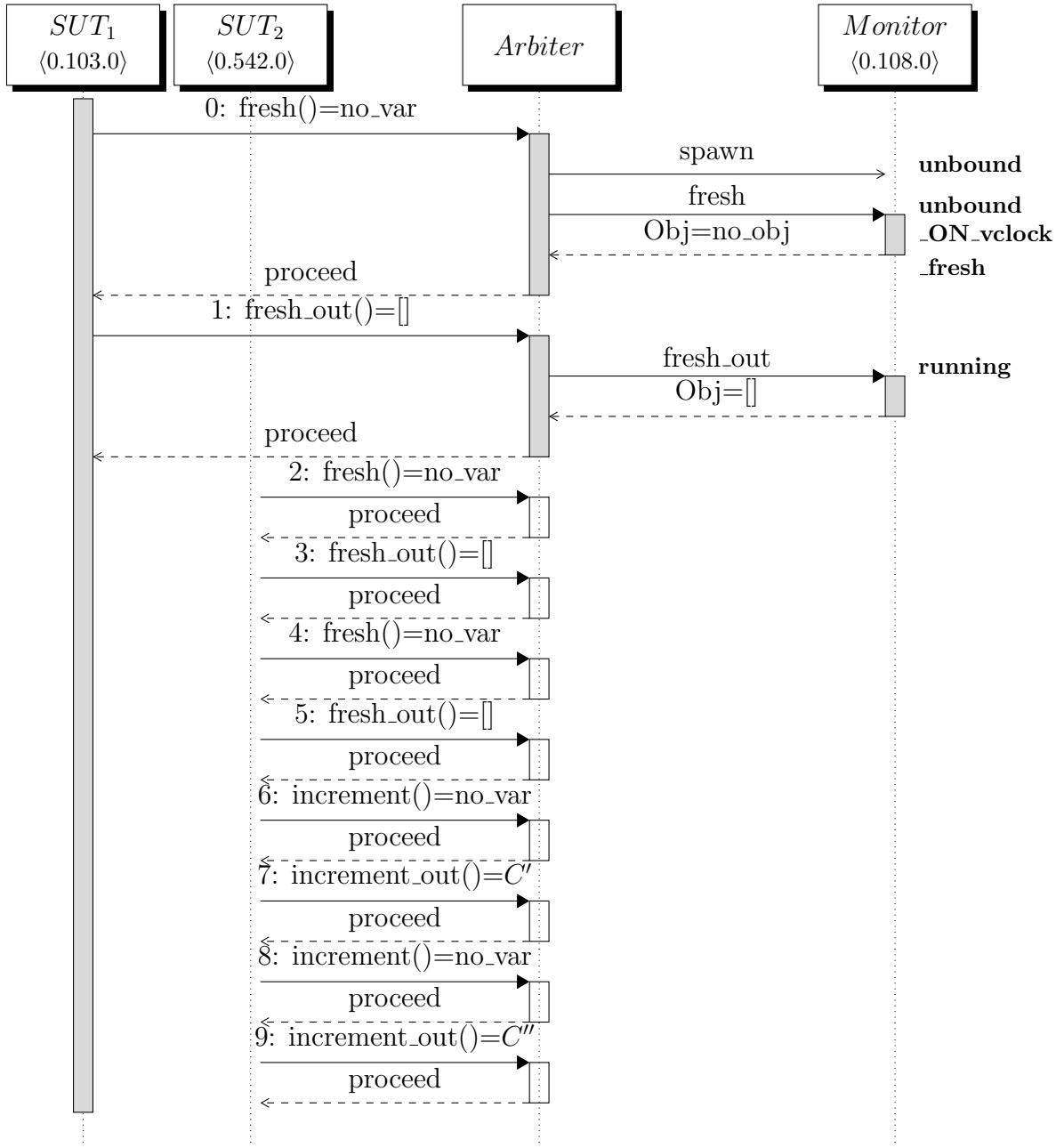

Figure 6.7: Inserting two objects into a database. Events 0 and 1 are generated by an internal vclock, and cause the monitoring of the actual insertion process to be ignored (events 2-9).

Figure 6.7 shows the result of creating and inserting two objects into the database. As SUT_1 fires the first vclock event, it causes the arbiter to clamp onto that source PID, ignoring events from SUT_2, which is the process that is executing the actual insertions. This policy is very fragile, and would only work if all events originate from a single process or if the process of interest can be identified beforehand.

Single Monitor, Object Tracking

The arbiter is set to spawn a single monitor and maintain an object mapping to it. All events that do not operate on a value that matches the mapped object are discarded.

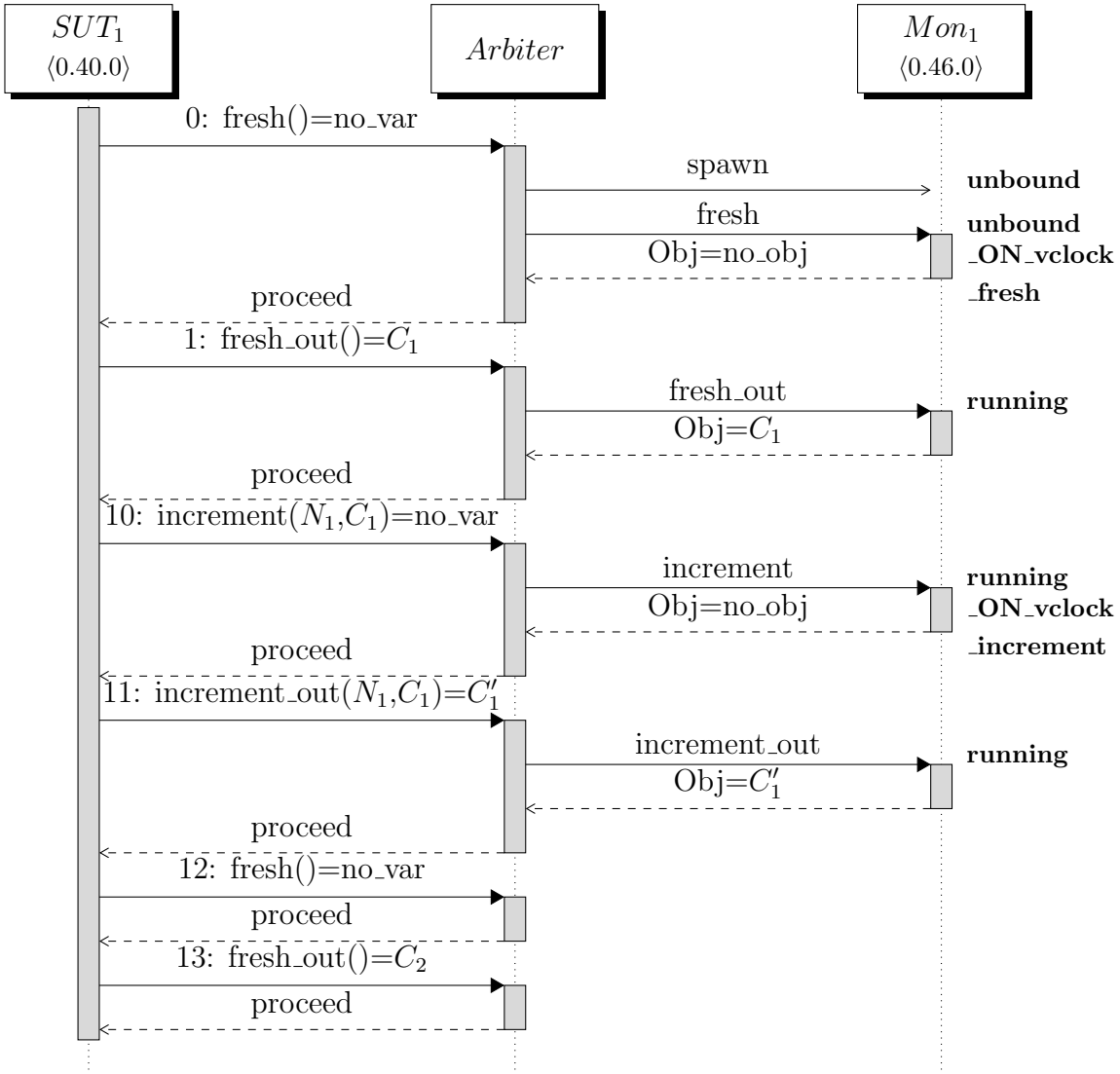

Figure 6.8: Single monitor, object tracking enabled

Figure 6.8 shows the event sequence produced on creating a Riak object. As the first event received is produced by the test, the monitor rejects all other events stemming from operations on vclocks maintained by the system (events 2-9, omitted in the diagram). Event 1 sets the object mapping, which is then updated by the increment operation that

triggers events 10 and 11. Events 12 and 13 do not contain the tracked object within their arguments list and are discarded, as the mapping is not stated. As mentioned earlier, one cannot through inspection determine the objects with which `fresh` events are concerned, and in this case, the event should be broadcast to all monitors of this property type whilst spawning a new monitor bound to its return value.

Single Monitor Per Source PID, No Object Tracking

For this scenario, the arbiter spawns a single monitor for every unique PID from which events originate. Although the tracked object field within the state data structure is being updated correctly, it is not used during this exercise.

Figure 6.9 shows the previous test executed with this policy. Unlike in the previous example, events fired through the use of internal vclocks do not cause subsequent events to be lost. The assumptions underlying this policy are that events concerning a given monitor will always originate from a single process, and that a process will only have at most one automaton for every property being monitored.

Figure 6.10 highlights an event sequence where these assumptions do not hold. The `merge` and `increment` operations (events 64 and above) are handled by a separate process, which causes a new automaton to be launched and bound. As this automaton is initialised as **unbound**, a `fresh` operation is expected, causing `merge` to move the automaton to an idle state.

As shown in Figure 6.10, although object tracking is not being used for monitor binding, the object mapping and update process is operating correctly. Events 60 and 61 update the monitor's tracked object field from a fresh vclock to C_1'. As the next event operates on C_2, the object mapping to the monitor is left unaltered as the function is taken not to pertain to the vclock C_1'.

Monitor Per Unique Source PID and Object Pair

As an extension to the previous cases, the arbiter is set to bind a monitor to every unique $\langle SourcePID, Object \rangle$ pair. Requests originating from different PIDs that address the same object will result in the arbiter launching a new monitor for that event stream. For correct operation, one should thus serialise requests related to the same object and ensure that they originate from the same process.

The following is a simple test that checks whether the system will move to a bad state when a postcondition fails. The postcondition result for a transition to the **running** state when executing **fresh** was inverted, and a new Riak object was created, giving the following output and logged events:

```
(myclient@127.0.0.1)2> riak_object:new(<<"buck">>, <<"key">>, [data]).
Bad State bad_state Reached!
```

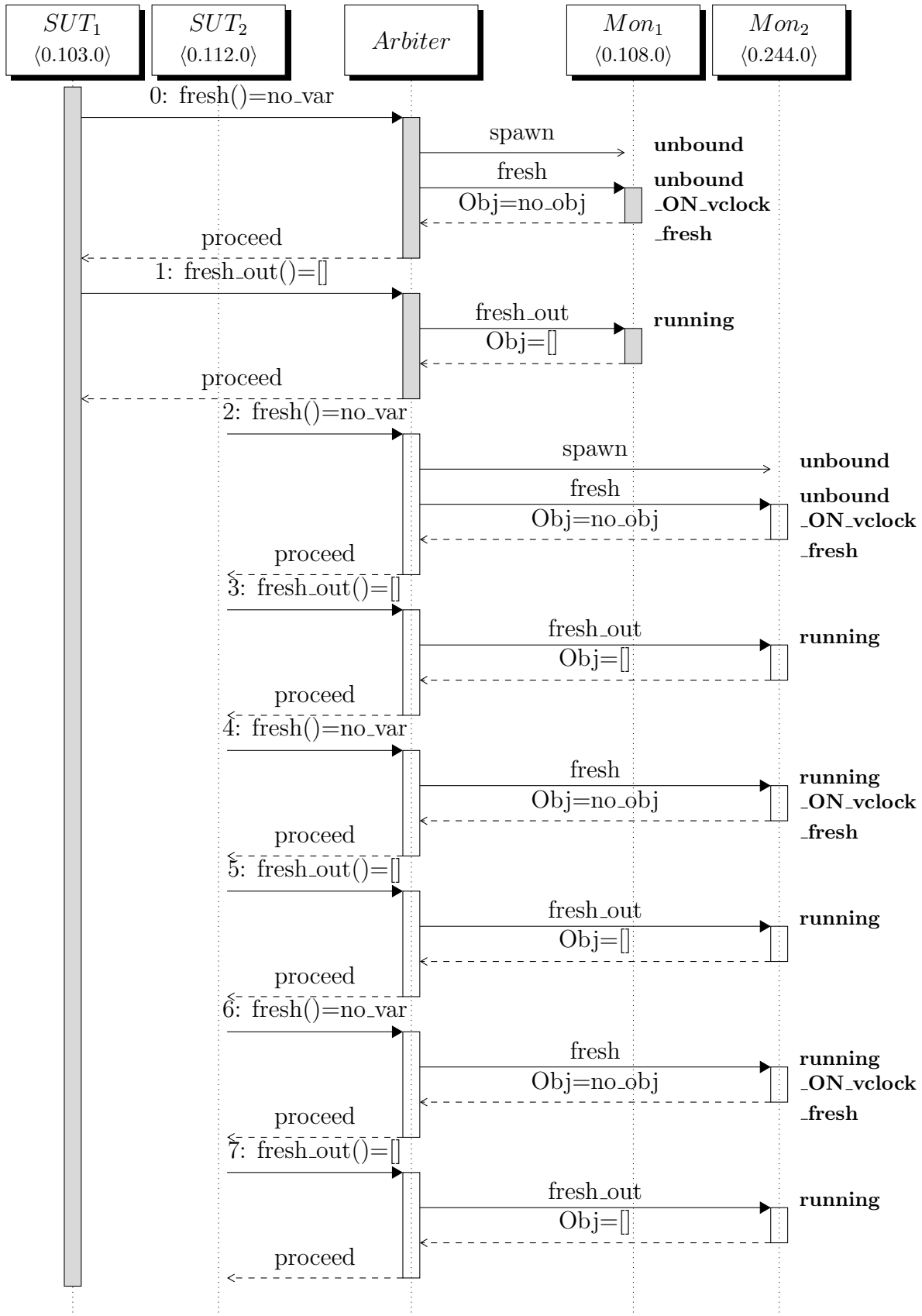

Figure 6.9: Inserting two objects into a database, monitor per unique source PID

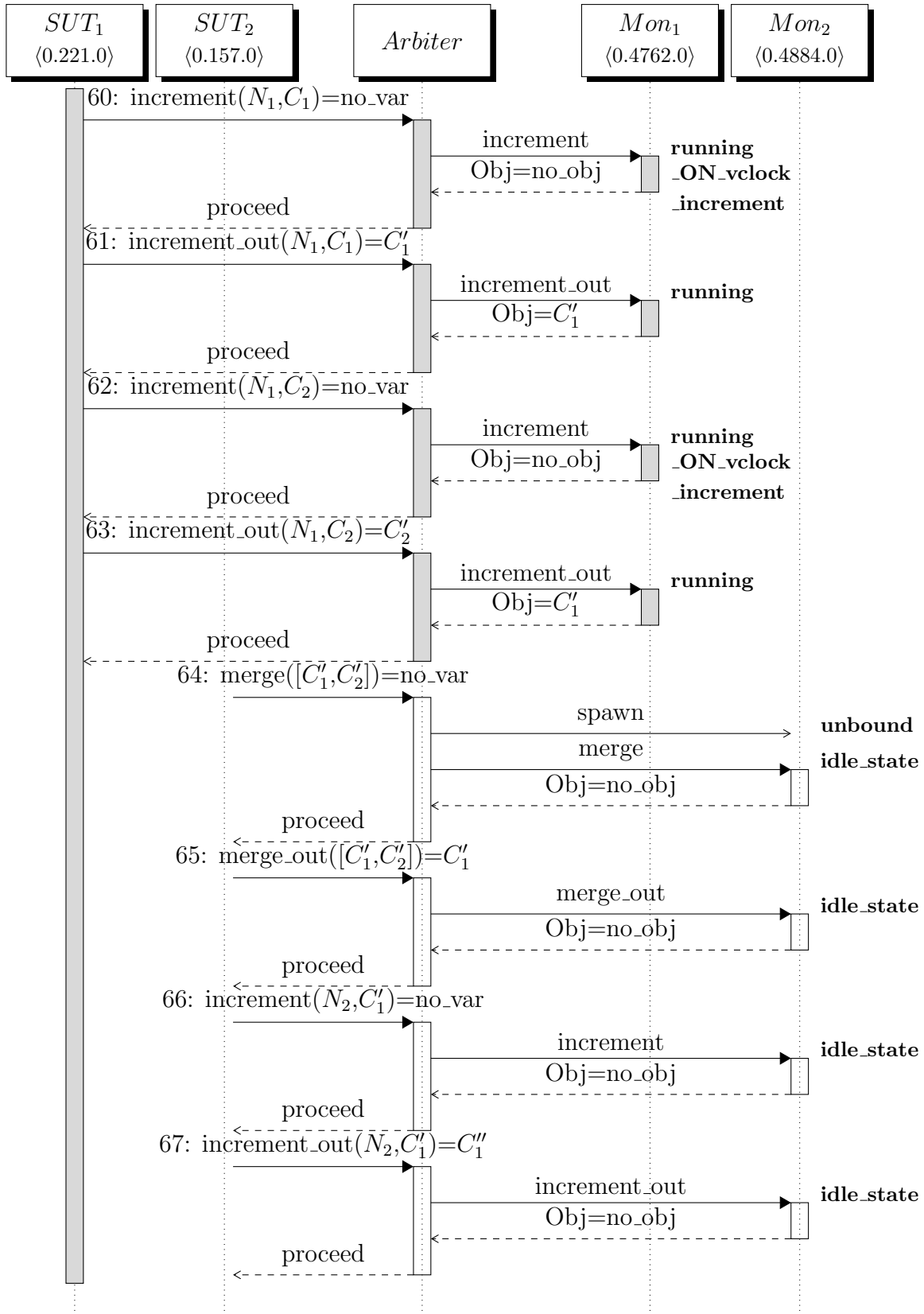

Figure 6.10: Failed monitoring of insertion. C_1 and C_2 have been declared within process SUT_1 and are set to []. Mon_2 is attempting to merge the vclocks, causing monitoring to suspend.

```
(myclient@127.0.0.1)3> arb_vclockfsm:log_print().
ARB: New Automaton [<0.45.0>]
ARB: [<0.38.0>] --> [<0.45.0>] 0:fresh()=no_var
     (<0.45.0> , unbound)   fresh()=no_var
ARB: [<0.45.0>] --> Object: no_obj

ARB: [<0.38.0>] --> [<0.45.0>] 1:fresh_out()=[]
     (<0.45.0> , unbound_ON_vclock_fresh) fresh_out()=[]
ARB: [<0.45.0>] --> Object: no_obj
(myclient@127.0.0.1)4>
```

Lines beginning with `ARB` refer to events handled by the arbiter process, with the remaining lines showing the current monitor state and the received event. The `fresh_out` event's condition succeeds for the arc leading to the bad state, causing the monitor to report a failure.

Figure 6.11 shows the event trace generated when a Riak object is created and inserted into the system twice, with a vclock object created between insertions. More specifically:

- Events 0-3 are fired on creating object **O**.
- Events 4-5 are fired on manually creating a separate vclock. This forces the arbiter into spawning a new monitor for the new object.
- Events 6-7 are fired on re-inserting object **O**. Riak calls `increment` as part of its insert procedure, with **O** already existing within the database. The arbiter has a monitor bound to an empty vclock C_2. The increment operation is invoked with an empty vclock as an argument, which is taken to be C_2 by the arbiter, causing the associated update in its mapping.

In some instances, two tracked objects with separate monitors may converge onto the same value. If the objects are associated with the same source PID, then the arbiter must decide which automaton should receive events for that object. As an example, consider the execution of the following commands

```
(myclient@127.0.0.1)1> O1 = vclock:fresh().
[]
(myclient@127.0.0.1)2> O2 = vclock:fresh().
[]
(myclient@127.0.0.1)3> vclock:increment(node(), O1).
[‘myclient@127.0.0.1’,1,63463461829]
(myclient@127.0.0.1)4> vclock:increment(node(), O2).
[‘myclient@127.0.0.1’,1,63463461846]
```

Figure 6.12 illustrates the event sequence produced. Events 0 and 2 induce the arbiter into creating and binding new instances of the property monitor, as the `fresh()` operation

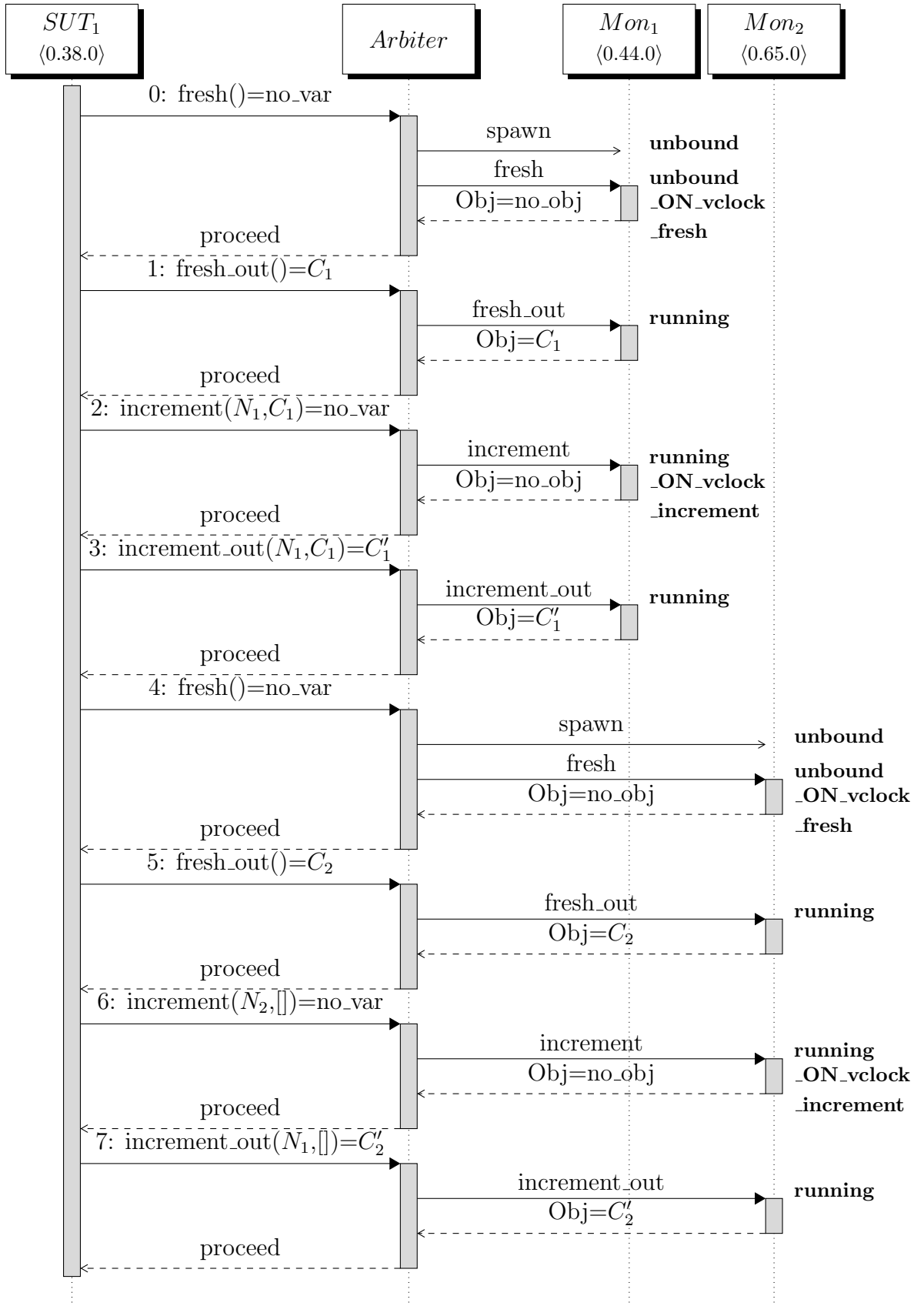

Figure 6.11: Object tracking with monitors per object and process pair

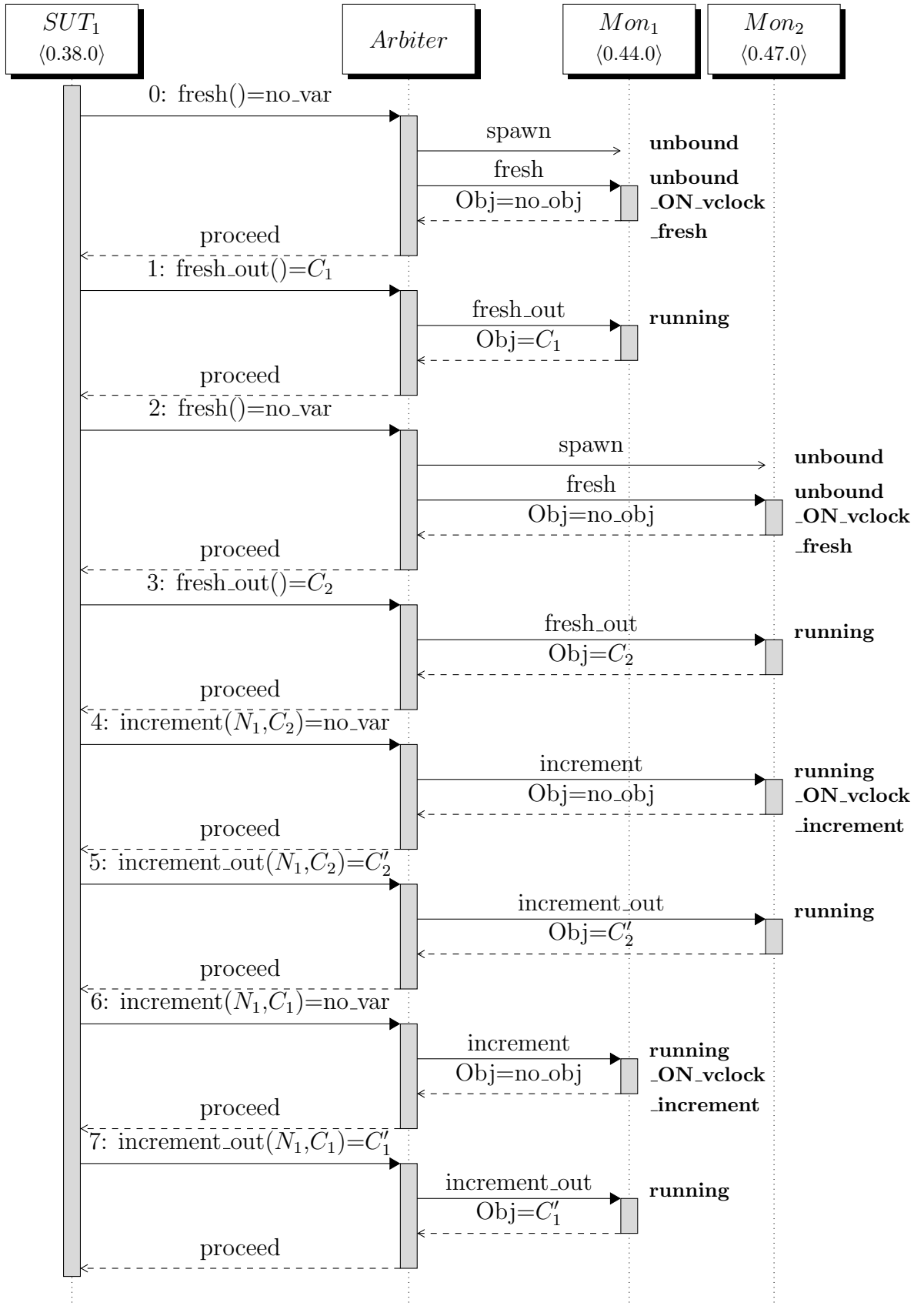

Figure 6.12: Object tracking with conflicting objects

is not bound to a particular object. This will result in two separate bindings to the same object value [] being maintained. As both monitors are bound to an empty vclock, the arbiter cannot determine to which automaton event 4 should be forwarded, and chooses one arbitrarily. When the second `increment` event 6 is received, the conflict no longer exists and the arbiter forwards the event accordingly. This ambiguity arises due to the non-uniqueness of the monitored objects, and is discussed in further detail in Section 7.2.2.

6.6 Case 3: Fine-Grained Insertion

Analyses: effects of test granularity and initial state data

6.6.1 Property Description

Section 6.4 presents a case that tests coarse-grained database operations. The test inserts and retrieves objects using Riak's high-level interface, and verifies the system based on the operations' results. While this approach is simple and effective, 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.

Object insertion and retrieval operations are handled using *Generic Erlang Finite State Machines*, or `gen_fsms` (Erlang behaviours are covered in Appendix A.4.4). The logic and control flow that manages either operation is implemented as a finite state automaton. The following case is concerned with the verification of the object *insertion* (put) procedure. Broadly, the automaton goes through three stages.

1. **Initialisation**, where the automaton's internal state values are set. These values define the parameters of the operation, and include the object that is to be inserted, the **W** and **DW** values (Section 6.2.3) and the compilation of an ordered list of eligible vnodes to which the object is to be committed.
2. **Transmission**, where the object under question is sent to the identified vnodes.
3. **Confirmation**, where the transmitting `gen_fsm` waits until the expected number of write and durable-write confirmations are received.

Every stage of the transmission protocol is implemented as a state transition. On completing the initialisation and transmission stage, the automaton enters a `wait_w` state. When the target vnodes reply with a successful write or durable write notification, an event is received by the automaton, which unblocks and increments locally-maintained counters. Once the **W** quota has been reached, the system moves to a `wait_dw` state,

in which it begins to harvest durable-write confirmations. If the **DW** quota had already been met while the automaton was still in the prior state, then the automaton moves directly to the end state, otherwise it must wait for the remaining **DW** signals to arrive.

Figure 6.13 details a QCFSA which replicates the control flow and logic of the `gen_fsm` that implements the `put` operation, allowing the generation of traces of the send protocol with checks on each stage of the operation. The QCFSA differs slightly from the implemented `gen_fsm` in that the initialisation phase is broken down into a sequence of multiple states. In the original automaton, the states **s_ready** and **s_sent** are merged into a single state, with the transition's associated functions being called from within a single function. Splitting transitions into intermediate steps allows the property to inspect the internal workings of the operation, thus leading to finer-grained testing. As the system only allows the instrumentation of function entry and exit points, intermediate values of functions can only be inspected through *decomposition*. When decomposing functions, it is imperative that the original function's entry and exit points remain unaltered, as otherwise the module's compatibility with the rest of the program will be violated. Decomposition also requires modifications to the system under test. Section 7.2.1 elaborates on the issue of decomposition, and its relation to test granularity.

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. This is because its events would not be correlated directly to the implemented functions, and would subsequently cause them to 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. Ultimately, this approach would be more focused towards testing the operation of Erlang `gen_fsm`s rather than the implemented callback functions.

Initial QCFSA Conditions

The case presented in Section 6.4 suffers from the fact that certain test parameters are hard-coded into the QCFSA's initial state. In the aforementioned example, the property only performed insertions and retrievals using a fixed pool of pre-set Riak objects. Ideally, the pool's size and contents are changed between tests, verifying the system under a multitude of conditions. QuickCheck allows properties to override an automaton's default QCFSA initial state by specifying the initial conditions as parameters to the generator.

Listing 6.4 details the QCFSA's initial state data function. Unlike in the cases analysed previously, the initial state data function accepts a set of arguments. The arguments are generated using QuickCheck's data generation functions prior to generating traces from the QCFSA. The state data fields are the same as those used within the `gen_fsm`'s state data structure, apart from the **q_replies** field which is used by the QCFSA. Instead of replicating these fields, one could opt to define a single field within the QCFSA's state data

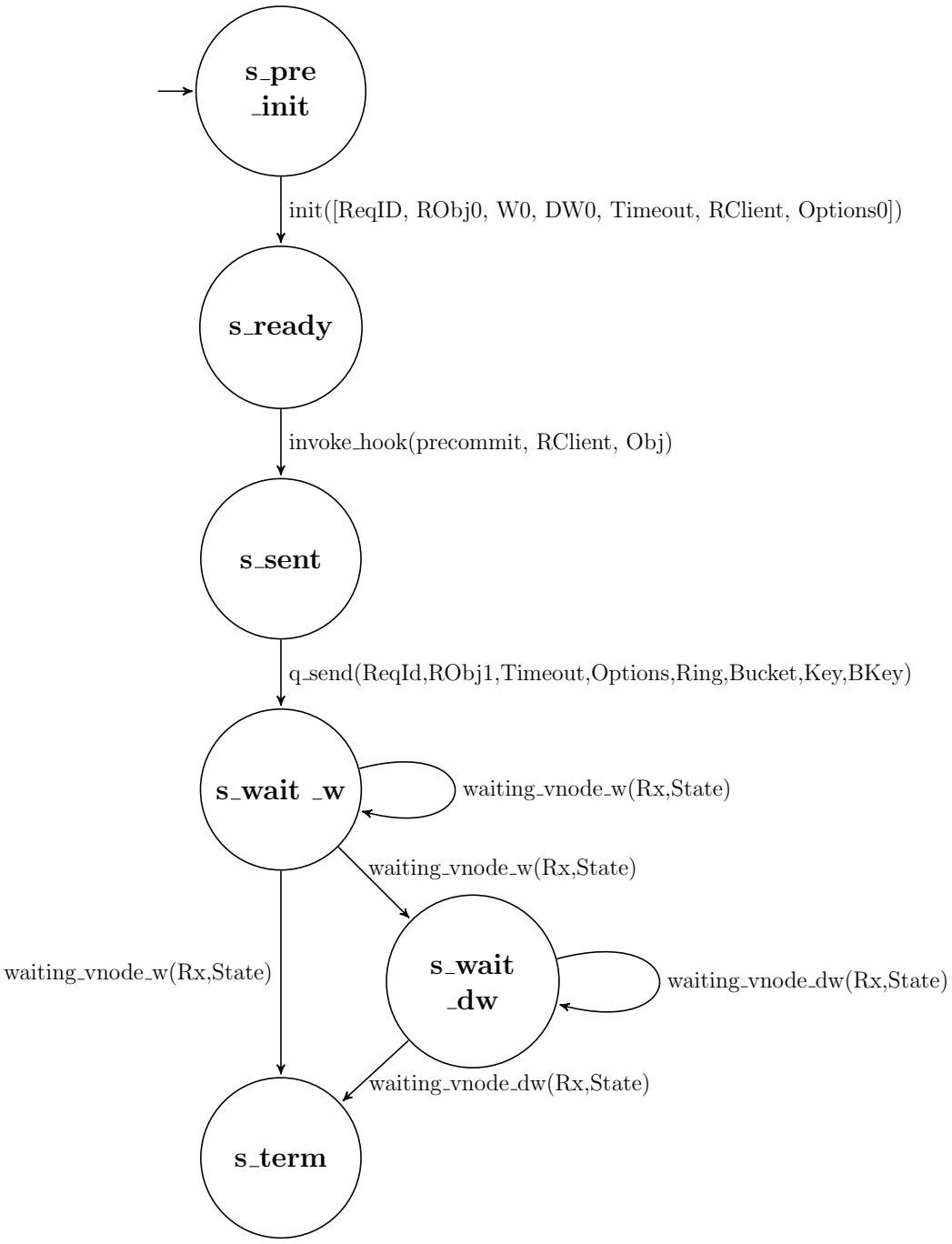

Figure 6.13: QCFSA property that generates a command sequence for inserting objects into Riak

Listing 6.4: Initial state data function for `put` QCFSA

```
1  initial_state_data(RC, Rng, Bk, Key, Value, RID, W, DW, Reps) ->
2    #state{
3      robj=riak_object:new(Bk, Key, Value),
4      client=not_set,    %% update in next state data
5      rclient=RC,
6      n=proplists:get_value(n_val,
7                            riak_core_bucket:get_bucket(Bk, Rng)),
8      w=W,
9      dw=DW,
10     bkey={Bk,Key},
11     req_id=RID,        %% update in next state data
12     replied_w=[],
13     replied_dw=[],
14     replied_fail=[],
15     timeout=2000,
16     ring=Rng,
17     options=[{returnbody, false}],
18     allowmult=false,
19     reply_arity=1,
20     q_replies=Reps
21   }.
```

record and store the entire `gen_fsm`'s state within it using the original record definition, keeping the QCFSA and `gen_fsm` constructs separate.

Moving the initial state data generation logic out of the QCFSA provides a cleaner separation of the generation and verification aspects of the automaton, yet care must be taken when converting the automaton to its DATE equivalent, as the arguments with which the initial state data function is invoked will not be initialised. For example, the function detailed in Listing 6.4 stores the parameter **W** in the state data structure. The remainder of the property assumes that the state data's **W** field is defined, and makes use of it within the pre- and postcondition functions. In this case, the state data's **W** field should be updated at some point within the property during a transition's action so as to mirror the live value correctly. Caution must be employed when changing the initial state at runtime, as the QCFSA should not produce traces whose verdicts change between generation and execution (Section 2.5.4).

6.6.2 Operation and Results

The act of inserting Riak objects into the database involves two parties, namely the *transmitting* node which initiates the `put` sequence, and the *receiving* vnodes which store the object in question. The property is concerned solely with the function call sequences

that must be performed by the transmitting node, and does not invoke any functions which would normally be used by the receiving vnodes.

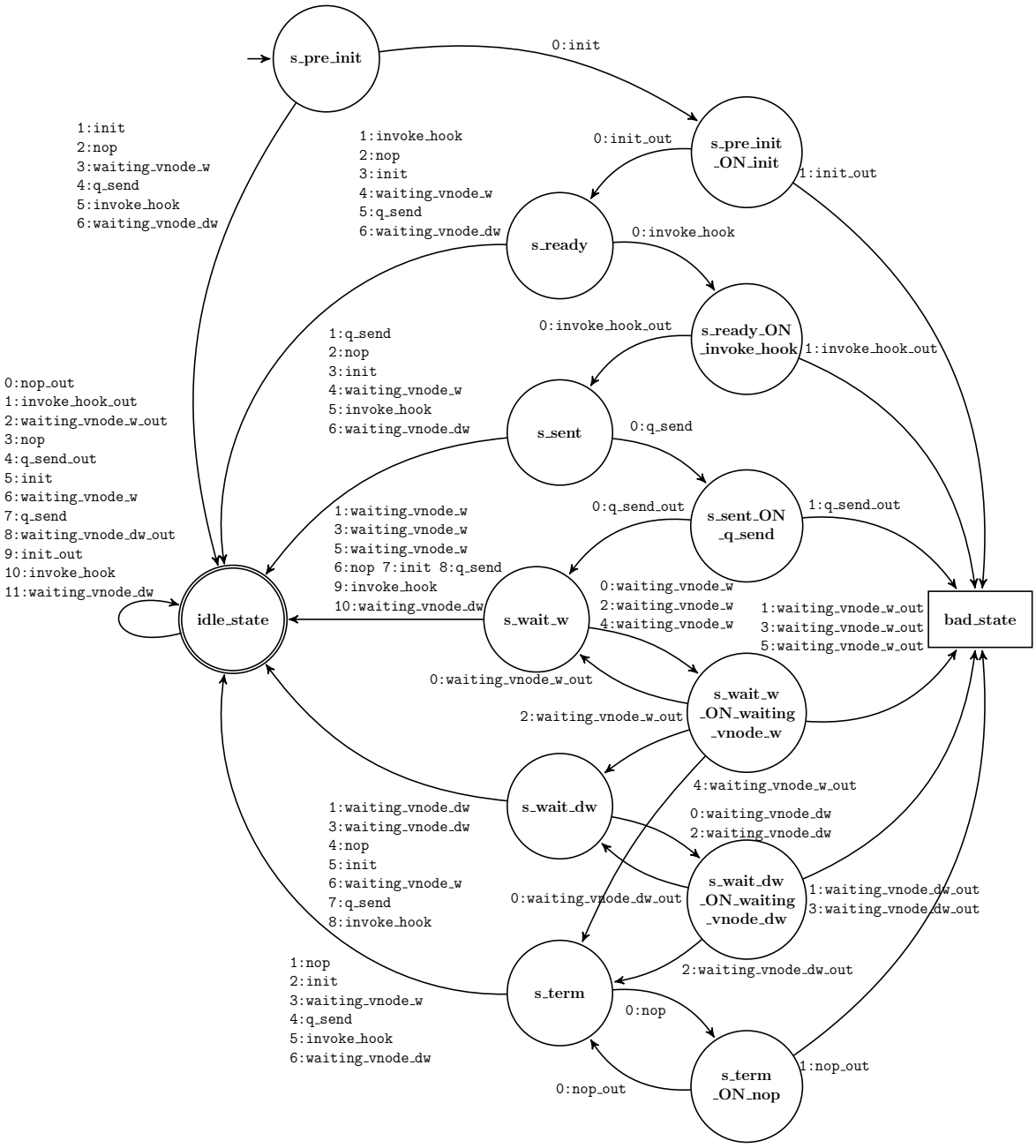

Figure 6.14: DATE for fine-grained Riak object insertion

As described previously, the transmitter's protocol is implemented in Riak through the use of a `gen_fsm`. After the transmitter broadcasts the object to be stored to the relevant vnodes, it blocks and waits for **W** and **DW** replies to arrive. These replies are generated by the receiving ends by sending an event to the transmitting `gen_fsm`. Each event consists of a tuple $\langle Type, Index, RequestID \rangle$, where $Type$ is either **w** or **dw**, $Index$ is an identifier for that vnode's communication and $RequestID$ is an identifier assigned to the entire insertion operation. These events are normally passed as arguments to the

state functions at **wait_w** and **wait_dw**, which add the messages to the **replied_w** and **dw** fields stored within the `gen_fsm`'s state data.

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 the automaton's **q_replies** field within the state data structure with a stream of **W** and **DW** reply messages that matches that expected by the transmitter. For example, if the `put` operation requires **W** and **DW** values of 3 and 2, respectively, then the property's **q_replies** field will be initialised with a list of five corresponding `gen_fsm` event tuples. Depending on the automaton's current state, the functions `waiting_vnode_w` and `dw` are then invoked using events taken from this list as an **Rx** argument.

Generating a matching stream of reply events allows the property to be executed outside of a live Riak system, as it will not require responses from the target system. It also obviates the need to emulate the `gen_fsm`'s blocking nature within the property, as otherwise the property would have to implement a method for harvesting replies. Nevertheless, when executing the runtime monitor on a live system, as is the case in the following examples, the target vnodes will send corresponding reply messages to the system under test. As the runtime monitoring framework makes use of Erlang messages to communicate between the arbiter and the SUT, these events were found to interfere with the monitoring process, particularly when the SUT blocks waiting for **proceed** signals originating from the arbiter. As a workaround, the SUT was set to queue any received non-RV related messages within a buffer when the SUT was expecting a message from the arbiter. Once the expected message is received, the SUT repeats the buffered messages to itself, ensuring that no signals are lost and that the system's execution proceeds normally. Naturally, the blocking nature of the monitoring process may violate any time constraints placed on the system, as would the additional overheads induced by runtime verification (Section 3.1.1). As the property only tests single object insertions at a time, object tracking is disabled, and the arbiter simply binds a monitor to each unique SUT PID.

Object Creation and Insertion, Mismatched State Data Contents

The first test attempts to illustrate the effects of mismatches between the DATE's state data values and the actual system state. These mismatches may arise if the QCFSA's, and subsequently, DATE's internal state data structure is initialised with invalid values, which are then used within the monitor for trace classification.

Figure 6.15 shows the event sequences which are generated and handled when creating a Riak object and inserting it into a database. Although the insertion is performed correctly, event 9 moves the automaton into an idle state. This happens because although the `put` operation was invoked with the default **W** and **DW** values of 2, the QCFSA automaton's initial state data was set with $\mathbf{W} = 3$ and $\mathbf{DW} = 2$. Consequently, the system only generates two **W**-confirmation signals before generating a **waiting_vnode_dw** event, whereas the automaton is still expecting another **W**-confimation to arrive. As **waiting_vnode_dw** events are not handled at state **s_wait_w** in the original QCFSA, the system moves to an idle state.

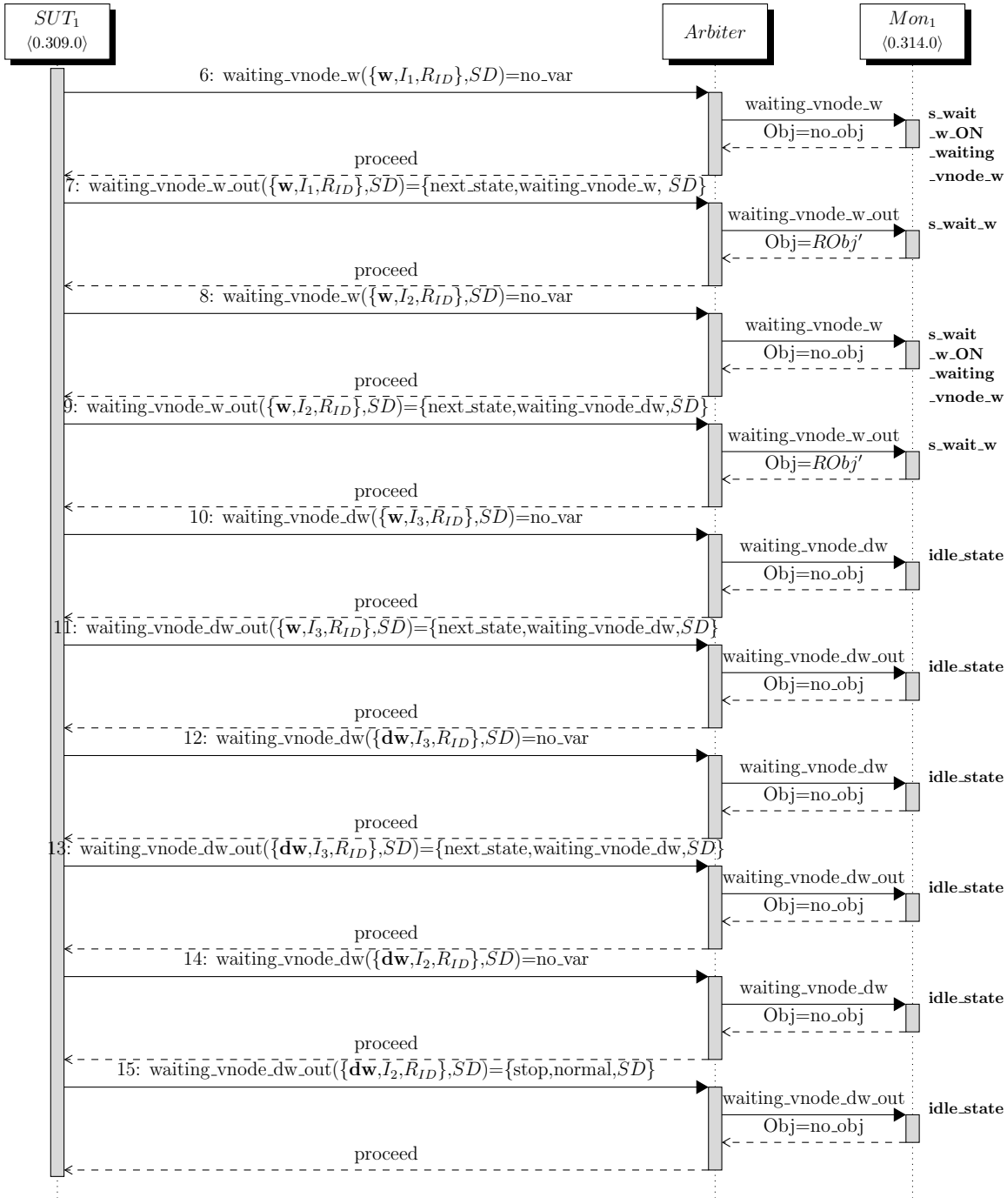

Figure 6.15: Creation and insertion of an object into Riak with incorrect initial state data values. Events 7, 9, 11, 13, 15 return the next state tuples used by the `gen_fsm` (events 1 to 6 are identical to those in Figure 6.16).

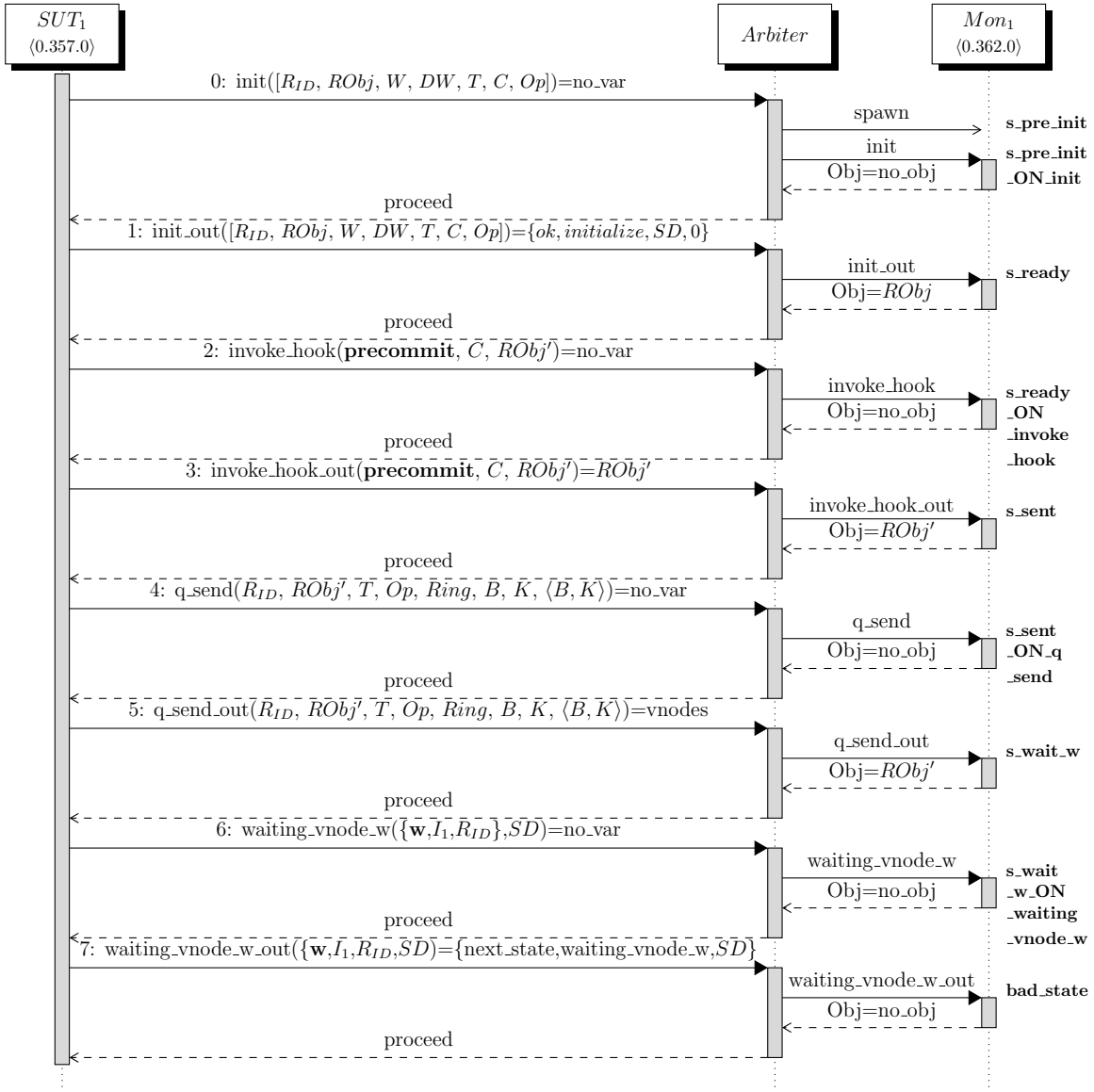

Figure 6.16: Inserting an object into Riak with the postcondition for `waiting_vnode_w` events at **s_wait_w** negated

Mismatches between the state data and live values will result in the monitor generating incorrect verdicts, as in the case illustrated. In general, if a property accesses the automaton's state data from within a transition's conditional or next state data functions, then one must ensure that the internal values and the live system state values are harmonised. One could achieve this by updating the internally-stored values with live values that are determined at runtime. For example, the `next_state_data` function related to the first call to `init` could update the **W** and **DW** values to those being used by the `put` operation, as they are available as arguments to the function. Updating the internal state is not always possible without breaking the property's operation as a QCFSA due to QuickCheck's use of symbolic variables during generation.

Although the arbiter does not make use of object tracking in this experiment, monitors still send messages containing objects as acknowledgement signals to the arbiter. As de-

scribed earlier, these objects are simply the second element of the automaton's state data structure, and in this case are taken as proceed signals whose values bear no significance. Finally, although the monitor moved to the idle state, the insertion operation executed successfully, and a fetch operation using the object's bucket and key values returned the object correctly. The act of moving to an idle state only suspended monitoring and did not directly affect the insertion operation. Nevertheless, the monitor's deficiency led to parts of the operation not being verified.

6.7 Case 4: Translation of Arbitrary Properties

Analyses: application of translation on arbitrary QCFSAs

6.7.1 Property Description

In the previous cases, QCFSA properties were drafted with the express purpose of being translated into DATE properties. Consequently, the QCFSAs were written with considerations for the runtime environment, and certain non-translatable elements of the properties were factored out during their creation. Thus, for example, the QCFSA properties presented did not access state data values which were not bound at runtime.

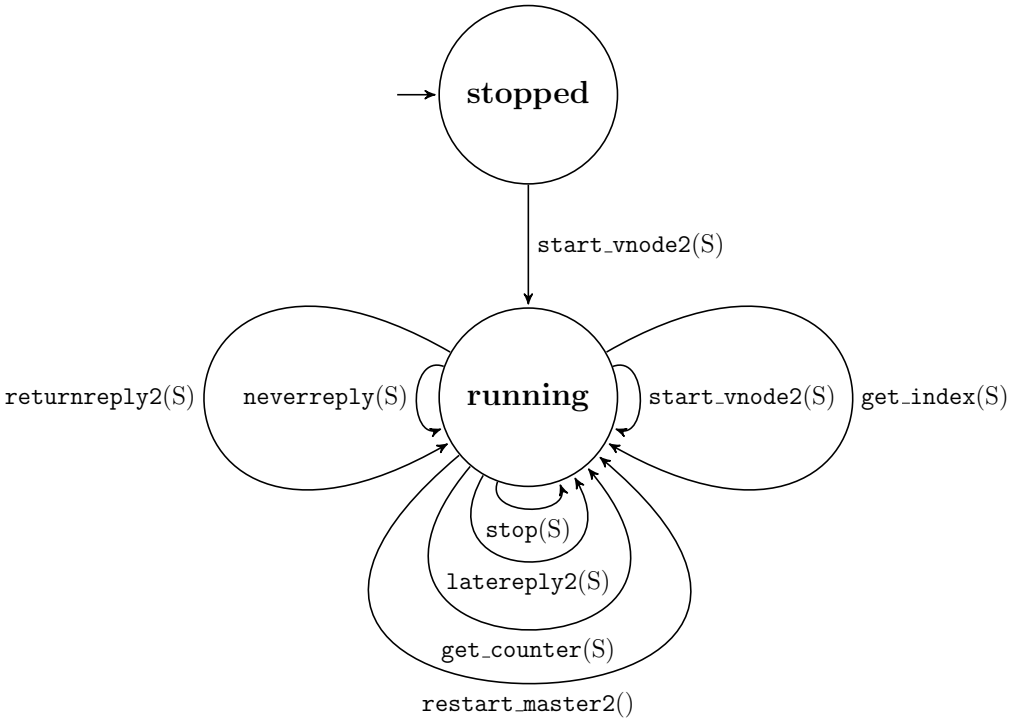

Figure 6.17: A QCFSA packaged with Riak that tests the vnode-master service

The purpose of this test case is to examine the translation of a Riak property that

was written without these considerations, that is, properties that are not biased towards being translatable into DATEs. Riak's source files are packaged with several QCFSAs that were developed for testing. Figure 6.17 is one such automaton, which tests the processes involved in creating and managing a vnode in Riak. This automaton was converted into a DATE monitor without any modifications, other than minor cosmetic changes to the event names which were required due to the underlying functions' existence within different modules.

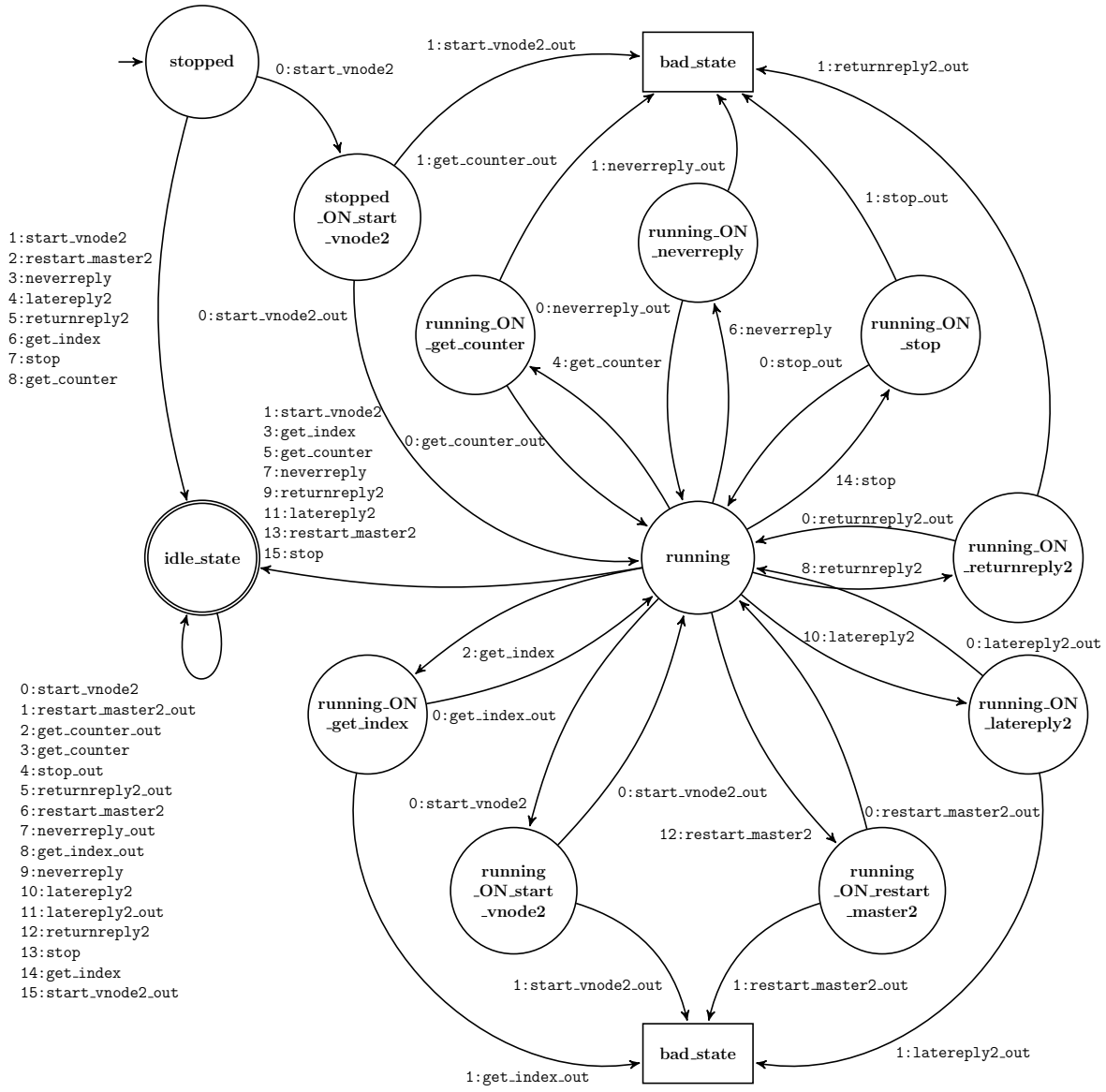

Figure 6.18: DATE for Riak vnode master

6.7.2 Operation and Results

The automaton's state data is set to fixed values on automaton creation, removing any issues related to mismatched live and stored data. The derived DATE monitor is launched

within the live system. The arbiter is set to spawn a new monitor for every unique SUT source PID.

Immediate Monitor Idling

When writing properties, one must keep in mind that traces not within the automaton's set of testable traces will cause monitoring to be suspended. Figure 6.19 illustrates the effects of performing the following command sequence

```
(myclient@127.0.0.1)1> core_vnode_eqc:start_servers().
ok,<0.41.0>
(myclient@127.0.0.1)3> mock_vnode:restart_master2().
ok,<0.50.0>
```

where `start_servers()` is an unmonitored function which initialises the vnodes being tested. The `restart_master2()` command sends a corresponding event to the automaton, which is still in the **stopped** state. As this behaviour is not expected by the QCFSA, the automaton moves to an idle state.

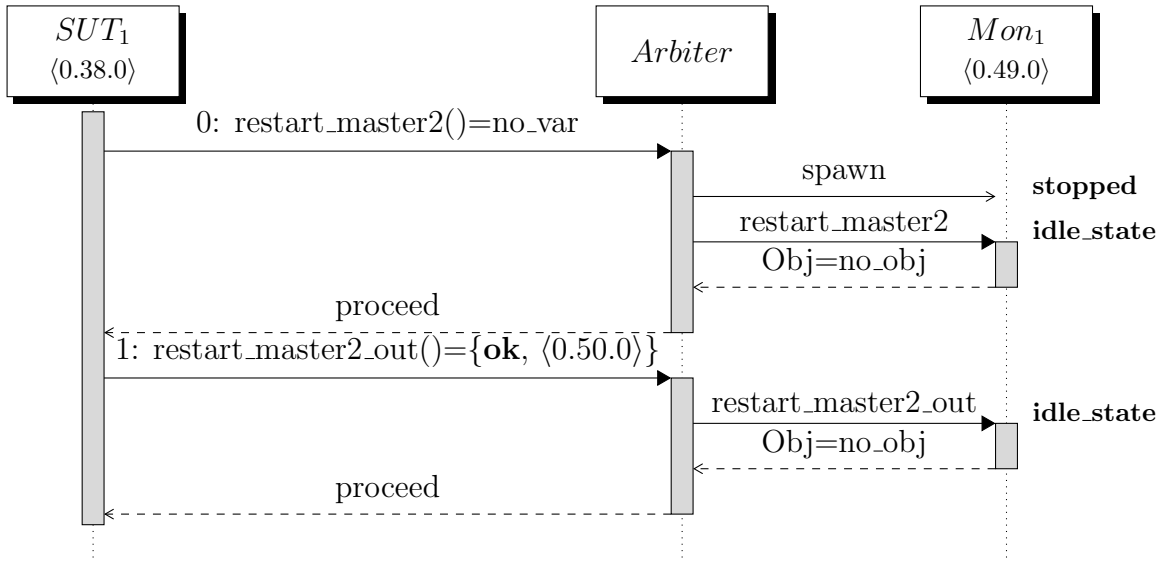

Figure 6.19: The event `restart_master2`, returning a handle to the `gen_server` launched on adding a vnode, is received prior to `start_vnode2`, causing monitoring to cease.

Demonstration of Nested Event Rejection

Figure 6.20 shows the event sequence generated when two vnodes are started. The sequence is a clear example of the intermediate event rejection mechanism at work. Within the `start_vnode2` operation (event 0), the system invokes the `restart_master2` function, which fires corresponding entry and exit events (events 2 and 3). Since these arrive prior

to `start_vnode2`'s matching exit event, they are not not forwarded to the monitor by the arbiter. Consequently, the arbiter only accepts new events once event 1 (the *out* event) arrives and is forwarded. The process is repeated when the second vnode is launched, with matched events 4 and 5 being forwarded and intermediate events 6 and 7 being blocked.

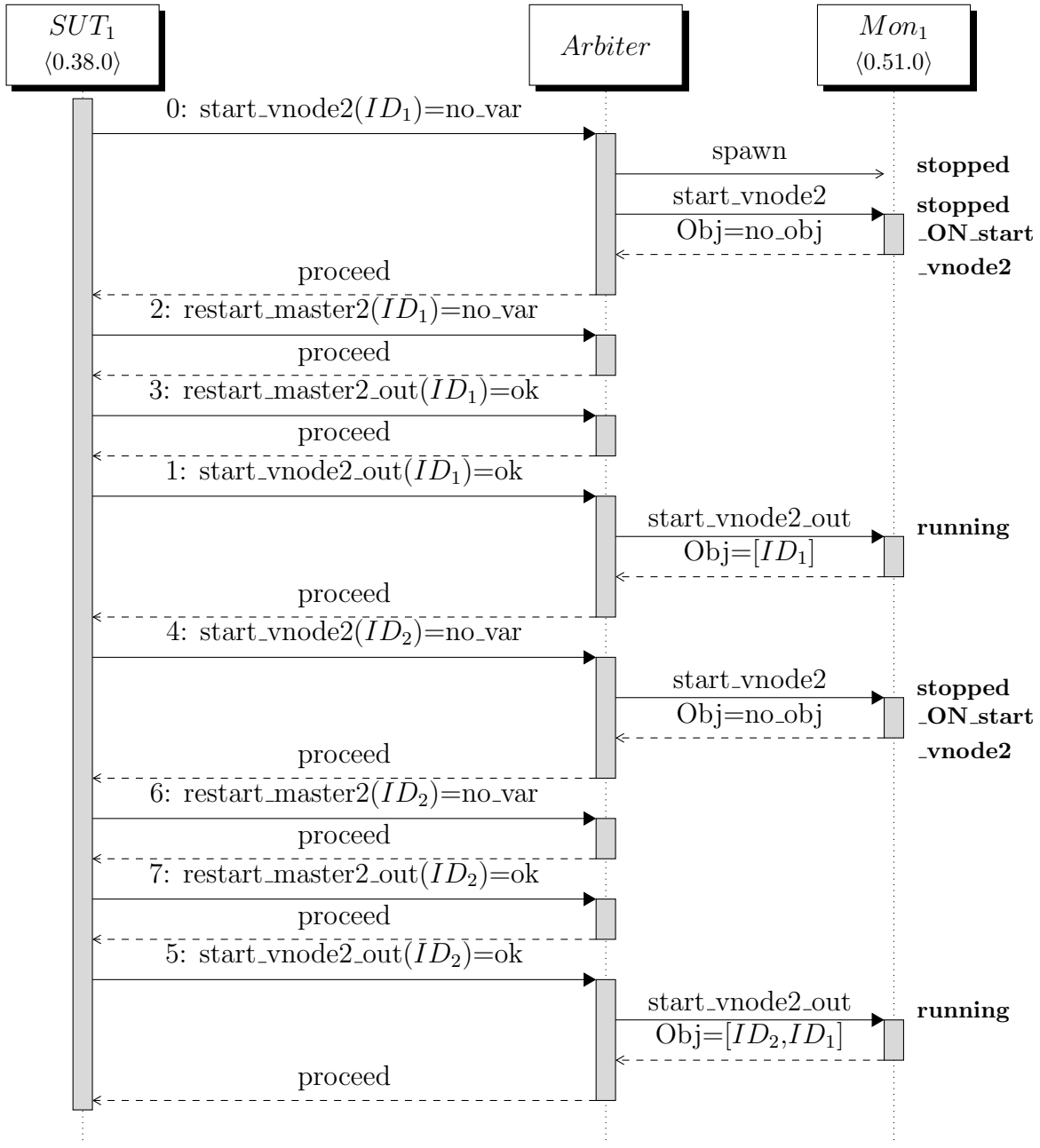

Figure 6.20: Nested event rejection. ID_1 and ID_2 are partitions within the ring.

As always, although the arbiter does not make use of object tracking, the monitor still reports the first element of its state data structure back to the arbiter. In this case, this corresponds to the list of partitions to which the property's initialised vnodes are bound.

6.8 Conclusion

This chapter has focused on the translation and monitoring of four properties using different event forwarding policies. The aim of the case study was to analyse the monitor's performance and behaviour when deployed in different verification scenarios. The next chapter interprets the results, evaluating the approach's general applicability and outlining ways to mitigate any identified shortcomings.

Chapter 7

Evaluation and Comparison with Related Work

7.1 Introduction

The following chapter builds on the results produced by the case study, highlighting the main issues encountered during translation and discussing the method's overall effectiveness, as well as the prime considerations that have to be followed when writing input properties. Other relevant points concerning the translation of models into different representations are also analysed towards the end of the chapter.

7.2 Evaluation of Approach

This section describes the performance of the principal components of the monitoring system and indicates the most relevant details and considerations.

7.2.1 Instrumentation

When instrumenting the system under test to generate events, the solution presented makes use of static code inlining. The solution also limits itself to instrumenting function entry and exit points. The use of static inlining facilitated the evaluation, as the system could be instructed to execute any arbitrary code at the instrumentation points, offering very fine control over the contents of event messages and the event transmission process. This also simplified the implementation of blocking instrumentation points by halting the execution of the system under test until the monitor confirms the event's reception.

A less intrusive method of collecting program executions would be to use Erlang's in-built *tracing* [Eri10d] tools. Tracing allows one to track system-wide events, as well as events occurring within individual processes. Events can consist of inter-process communication and function entry and exit. Tracing can also automatically match function

entry and exit pairs whilst foregoing the logging of intermediate events. Events can be timestamped if necessary. Logged events can be forwarded at runtime to a monitoring process, such as a property arbiter. Such an approach would be ideal for monitoring, but would be more restricted if the system under test is to perform additional tasks at its instrumentation points other than event reporting. In addition, triggers do not cause the system to block, making it harder to synchronise a monitor with the program.

When employing message passing to coordinate instrumentation and event reporting, one must ensure that runtime monitoring messages do not interfere with other messages used within the system under test. One should also appreciate the fact that runtime monitoring, even through Erlang's tracing facilities, can add significant overheads to the system which were not originally present. Delays are further aggravated if the system under test blocks until the monitor sends a reply. Consequently, monitoring could cause certain time-critical processes to fail or time out.

Logging

As described in Section 5.4.1, the implemented solution uses an arbiter process to manage the forwarding of events and acknowledgements between the system under test and the DATE monitors. One advantage of such a system is that the arbiter can analyse the live event streams, which makes logging communications a very simple matter. By analysing logs, one may also extrapolate statistics regarding events and their frequencies. These facilitate the augmentation of the property's QCFSA through the addition of *weights* on its arcs, where one can promote or decrease the frequency with which certain traces are generated by the QuickCheck engine depending on how often they have been verified at runtime.

Granularity of Tests

The granularity at which properties operate depends largely on the framework's instrumentation capabilities. Although the insertion of instrumentation points at arbitrary program locations is technically feasible, especially when employing static inlining, it would complicate the automation process, as the association of events to code locations would no longer be implicit. This would also necessitate changes to the QCFSA model by redefining what constitutes an event. Thus, functions are regarded as the base event type.

When writing properties that concern a sequence of events, one must create an automaton that accepts all the transitions necessary for that trace to behave correctly, notwithstanding the fact that some of these transitions might not require verification. To illustrate, consider a trace concerning three functions $f()$, $g()$, and $h()$, that are called in sequence within the system under test. Although one may only be interested in monitoring the results of $f()$ and $h()$, an arc must be added within the automaton for the intermediate function $g()$, as otherwise the QCFSA would generate traces without $g()$, which might not normally be executed by the system under test.

In general, if the traces require that a specific sequence of functions be executed, then the automaton must enforce the ordering. Also, once a function is instrumented, a transition must be placed at every state at which the event may arrive in order to ensure that the monitor does not move prematurely to an idle state. Thus, properties must be very precise in defining what constitutes a correct program behaviour. The effect of this precision is that the complexity of a property, with regards to control flow logic and transitions, tends to increase as the granularity of the system under test becomes finer. As an example, one may compare the cases analysed in Section 6.4 and 6.6. The former case tests the correctness of an insert operation by analysing the result of a single function call, whereas the latter replicates every step being performed internally by Riak and verifies each individual step. While the latter property can isolate failure points more precisely, it is more complex and harder to follow, and requires detailed knowledge of the system's internals. Coarse-grained properties tend to be straightforward, as they are only concerned with a function's high-level behaviour. Although the pre- and postcondition and next state data computations can be arbitrarily complex, the automaton's control flow can be very simple, as it does not analyse the function's internal state changes and execution path. Finding the right granularity at which to operate is crucial, since very fine-grained monitoring may lead to the system's complete behaviour being replicated within the property, whereas very coarse-grained testing would not allow the monitoring of intermediate steps.

Since instrumentation is not performed at arbitrary program points, the system may only be inspected at a finer granularity by decomposing and chaining functions. When decomposing functions, it is important that the original function's header and ultimate return value remain unmodified. The case presented in Section 6.6 makes extensive use of decomposition in order to access intermediate variables whose values are then checked either as arguments or return values. Moving to coarser granularities would require that the automaton calls the relevant functions in a sequence that constitutes a single grain. One should not create a new function which performs this sequence and call it from within the property, as it would result in the system instrumenting an alien entry point that is never invoked at runtime. Testing may require significant restructuring of the system under test, unless one develops the system and the property simultaneously or opts to follow disciples such as test-driven development (Section 2.6).

7.2.2 Issues Affecting Translation

This section describes the issues identified during the evaluation process which limit the translation's applicability to arbitrary QuickCheck automata, suggesting principles that should be followed when building a property.

Symbolic Variables

Certain properties that would normally be very easy to verify using DATEs can be hard to implement using QCFSAs. One prominent recurring issue encountered throughout the case study was QuickCheck's use of symbolic variables during the trace generation phase.

By replacing function return values with symbolic variables (tokens), the system prevents properties from manipulating or directly inspecting a function's return value, and [Quv10] recommends that this should be avoided, with results being treated as immutable black boxes. Thus, one would have to constrain properties and only examine abstract program behaviours rather than individual low-level operations. Alternatively, one could replicate the relevant state-data changing procedures within the automaton through actions over transitions, removing the property's reliance on externally-generated results. The latter approach violates the principle of never re-implementing a function that is being tested within the test, whereas the former results in clearer properties and cleaner logic at the expense of control.

Pre-generated State Data

As a consequence of its use of symbolic variables, QuickCheck requires that an automaton's initial state data be determined entirely during its initialisation and that it remains the same both during the trace generation and execution phases of the automaton. For example, one cannot initialise a variable to a random value within the initial state data, as this may lead to different paths being taken during the two phases. Instead, one could use a known, fixed seed for the random number generator and update the seed value deterministically, as in the case presented in Section 6.5. Alternatively, one could move the generation code out of the automaton and into the QuickCheck property harness, computing the random values prior to trace generation and fixing the initial state data to these values. This approach leads to the generation of cleaner runtime verification monitors, as the amount of state data that must be maintained over each transition decreases.

As evident in the case presented in Section 6.6, if the initial state data is not set using values that reflect the actual system state, then the use of a QCFSA's state data may mislead verification. For example, one could use the automata's state data structure to store dummy values which the property then uses within its checks. If these dummy values have live counterparts within the system, then the automaton should replace them with the actual readings taken from the runtime environment using a state changing action. An exception to this would be if the state data is being used to restrict the analysis of certain traces (such as ignoring all traces exceeding a given length), in which case the values should remain unaltered. State data values that are set from within the QuickCheck property harness and are vital to verification must be copied into or recalculated within the initial state data function, as otherwise the values would remain undefined.

Concurrency and Object Tracking

QuickCheck automata execute within a precisely-controlled environment. The test harness manages the setup and shutdown procedures prior to every trace execution, which execute as separate processes. This also shields the process from external interactions other than those brought about as a consequence of the functions being tested.

These assumptions do not necessarily hold within a live, multiprocessing environment. As a QCFA manages its own trace generation and threading, it can implicitly identify

which function invocations belong to which trace. Thus, even if one were to test multiple QCFSAs in parallel, the properties will keep track of their traces and progress.

Problems may arise if multiple instances of a monitor for a given property exist simultaneously during verification. The issue arises on account of the scope of events. When running two QCFSAs in parallel, executing a function within one automaton's traversal would not affect the other's monitoring[1]. On the other hand, when an event is fired within a runtime environment, it is forwarded to all the monitors that are listening for that event. This implies that events are global, and that multiple automata will all receive the same interleaved sequence of events. Thus, the concept of object tracking was investigated in an attempt to bind events to specific contexts.

Object tracking centres around the identification of an object which serves as a common link between events, where each event pertaining to a given thread is bound to the same object, with the resultant thread monitored by a dedicated runtime monitor. For example, the case presented in Section 6.5 uses vector clocks as the common factor between related events, with each unique vclock being allocated its own monitor which only considers events that concern that vclock. This allows properties to be written with a narrower scope and without having to consider the entire system as a whole.

Object tracking is limited by three core factors. The first is that it is hard to track and distinguish between objects within the system. Data structures are not automatically assigned unique identifiers, and therefore require an additional mechanism for maintaining mappings between event sources and objects (and consequently, monitor processes). To further complicate matters, events related to the same object may originate from different processes, or conversely, unrelated events originating from different processes may concern objects whose values happen to be the same. The second requirement is that, in the absence of unique identifiers, the system must keep track of objects as they evolve throughout the system. This would demand that all updates to the object are tracked by the system, necessitating a duplication of effort and precise program analysis. Finally, the relation between events may be very contrived as the approach assumes that a property's events revolve around a single, shared and evolving object. Thus, this approach is mostly confined to properties that verify data structures. Unless objects can be assigned unique identifiers, these ambiguities cannot easily be resolved and require that the system's internals be fully understood so as to employ the appropriate mapping policies.

While properties with a global context are simpler to verify, one must still exercise caution when verifying events which originate from different processes. Erlang places no guarantees on the order in which messages sent from different processes are placed onto the shared destination queue. Thus, the order of events in a monitor's incoming verification stream may not correspond to the true order in which the events were generated during the program's execution. This must be catered for either at the property level, with properties written so as to accept different event interleavings, or by serialising events through a single process which re-orders them as necessary.

[1]Unless they share some state across processes.

7.3 Results

Based on the translation and proofs presented in Chapter 4, as well as the results obtained during the evaluation, it can be seen that the translation works, provided that the following conditions hold:

- There cannot be any hidden setup or environment-changing functions. In the translation procedure's formal description, the system state is assumed to be global, and the monitor's execution environment should be the same as that which exists during QCFSA testing and test case generation. Similarly, the automaton's state data should be initialised entirely from within a QCFSA, either using the initialisation function or by binding values when executing actions during transitions. If an automaton's state data is initialised externally prior to executing the automaton, as in the case presented in Section 6.6, then the property must ensure that unbound values are updated correctly during the automaton's traversal prior to their use. Any other relevant initialisation functions would have to be invoked using a transition in the QCFSA.

- The correct monitor spawning and binding policy must be used. Ultimately, the context binding mechanisms all address the same issue, namely that of routing the correct event streams to their associated monitors. The translation and proofs described inherently assume that one event stream is being produced by the system under test. Failing this, parallel event streams should be forwarded to separate monitors, providing the illusion that each monitor exists on its own. If events were to be interleaved, that is, if a monitor had to receive an event generated by a different sequence of operations, then the initial assumption would fail, invalidating the automaton's verdict.

  To illustrate, consider a system making use of a data structure $\mathcal{D}$ which, on being initialised, throws an `init` event and is then operated upon by one or more `exec` events. A monitor $\mathcal{M}$ verifying this structure would thus check event sequences of the form $\texttt{init} \cdot \texttt{exec}^+$. Within a QuickCheck environment, one would invoke and verify calls on $\mathcal{D}$ directly, removing any ambiguity as to which event is being checked. When monitoring at runtime, if only one instance of $\mathcal{D}$ exists at any one time within the system, then it follows that the event sequence is directly related to that data structure. Yet if multiple instances of $\mathcal{D}$ exist, then the event stream will contain an interleaving of events. Thus, a monitor could receive $\texttt{init} \cdot \texttt{init} \cdot \texttt{exec}^+$, which is outside its set of testable traces, causing it to suspend monitoring. Similarly, if a monitor $\mathcal{M}_1$ were to receive an `exec` event that was meant for $\mathcal{M}_2$, then the automaton's verdict is no longer reliable as the stream being monitored does not match the system's true behaviour, even though the resultant trace is still within the automaton's testable trace set.

  In general, interleaving event streams from multiple points of origin will present a monitor with a sequence of events which will cause it to either:

  - **move to an idle state**, as the current transition's associated precondition will fail on account of the trace being outside of the automaton's test set

- **take a transition**, leading to a false negative or positive classification
- **enter an undefined state** on receiving an event which is outside its set of interesting events. Such a trace is trivially outside the monitor's testable trace set, and should cause the monitor to idle. Conversely, it could also be argued that such an occurrence signals an unexpected behaviour and is definitely outside the set of acceptable system traces, implying a bad state. The view adopted by the presented model is to *ignore* unexpected events, as they do not, by definition, form part of an automaton's set of interesting events, which is assumed to be fully specified.

To summarise, a runtime monitor placed within a live environment will produce a correct verdict provided that only a single event stream is produced, with events being received in the same order with which they are produced and with no events being lost. Additionally, any necessary initialisation routines which must be performed should be incorporated into the QCFSA. One problem which may be encountered when initialising an automaton's state using values obtained externally is that the values of interest may not always be passed as arguments to monitored events. This situation would demand a more complex initialisation routine or the decomposition of functions in the system under test into signatures which offer access to intermediate values, as was done in the case study presented in Section 6.6.

As described earlier, the translation adopts the *closed-world* [KMM07] model of events, whereby if a received event does not form part of an automaton's set of interesting events, then it is ignored. While this view is often adopted, differences arise in how the reception of interesting events for which no outgoing arcs from the current state exist is handled, as will be seen in Section 7.4. The model described in this document assumes that the property is fully specified, and that the monitor should move to an idle state if an event is received and the original automaton cannot advance a step. This can be observed in the first scenario of the case study presented in Section 6.7, where a `restart_master` directive is received before a `start_master` event, causing the automaton to move to an idle state. To quantify the property over all event sub-sequences, that is, to generalise the property so that it is verified over all traces containing `start_master` and `restart_master` events in the correct order, one would have to create self-loops on the start state to consume all events in $\overline{\{\texttt{start\_master}\}}$. When adding self-loops, one must be careful not to introduce non-determinism. Other approaches assume that properties are implicitly quantified over the entire trace, with failing preconditions causing no transitions to be taken. The rationale offered is that systems are often partially specified, with verification being performed on a subset of traces, while discarding intermediate and unrelated events. This increases the size of the test set, and restrictions would then have to be added explicitly through events triggered on unwanted behaviours. Conversely, the approach adopted within the translation described in this project is additive, starting with an empty testable trace set which grows as the number of transitions increases.

A final observation is that it is relatively easy to write an invalid QCFSA property which translates into a correct DATE. For example, directly inspecting the return value of a function executed within a QCFSA transition is not allowed within QuickCheck, yet a runtime verification monitor derived from such a property would operate correctly.

7.4 Related Work

This project examined different aspects of verification, including the problem of generating suitable properties to test and verify, the difficulties in translating properties into alternative representations, and the methods by which they can subsequently be verified. The following is a discussion of existing alternative approaches to these tasks. Some of the translations presented make assumptions other than those adopted by this project, including different views on the input property's *completeness* (whether or not they are fully specified), interpretations of *totality*, and handling of *non-determinism*.

7.4.1 Translating and Enriching Models into Properties

The language in which models are expressed often varies across different model-based verification techniques, based on their aims and mode of operation. For instance, while DATEs serve as acceptors describing valid strings of system events, a QCFSA incorporates both a model of a subset of the system's behaviour and a property for classifying traces.

When transforming a model from one notation to another, one must ensure that the original model contains enough information with which to build the target model, or, alternatively, one must seek secondary sources from which the missing required information can be derived. The QCFSA structures used within this project carry at least as much information as the DATEs into which they are translated, and the process requires no further user input aside from directives to the E-LARVA framework for regulating context and object binding. Nevertheless, one does not always have immediate access to a precise specification of a system's behaviour, and weaker models may have to be used as a starting property. In such cases, one could express the initial requirements using a simplified model, which would then be incrementally refined until they are considered complete.

The techniques described by [DLvL06] aim to facilitate the automatic translation from simplified user-specified Message Sequence Charts (MSC) into Labelled Transition Systems (LTS) with the assistance of a user's interaction, which is kept to a minimum. The input MSC defines legal sequences of a system's events, and an LTS is built through grammatical inference (Section 2.3.5) by generating system use-cases which are then classified by the user as positive or negative scenarios. This serves to characterise the automaton's negative trace set from its set of testable traces. Several additional techniques must be applied during the induction stage, as otherwise the number of questions that are posed to the user may grow very large. Consequently, the system applies heuristics to prioritise state merges, and outgoing events from merged states are not added automatically so as to limit the automaton's size. Other extensions aim towards improving the generated model's consistency, which may be jeopardised should induction over-approximate the inferred grammar, and also towards automating steps such as the incorporation of certain safety properties.

7.4.2 From Models to Runtime Monitors

This project has focused on the derivation of runtime monitors from QCFSAs, yet other translations accepting specifications expressed in a different notation can be devised. [KMM07] describes a process of converting an input MSC into a state machine, which is then used for deriving AspectJ runtime monitors[2]. The resulting monitors limit themselves to the observation of communication patterns amongst components rather than their implementation, and classify a system as operating correctly when the messages being transmitted follow the protocol set by the input MSCs. As with the translation described in Chapter 4, generated monitors operate under the *closed-world* assumption, and any events with which the property is not directly concerned will be ignored. This allows MSCs which do not fully describe a system's behaviour to be monitored, that is, it allows the monitoring of MSCs whose set of testable traces is not a superset of the set of all possible interaction patterns.

As when translating QCFSAs, the process assumes that the input specification is deterministic, which simplifies the task of tracking the current state during monitoring. Since the input sequences are finite strings, the output automaton should be regular. The state machines into which MSCs are synthesised have messages as transition labels, and also support guard conditions. Each transition must specify whether a message is incoming or outgoing. Using this automaton, a monitor can detect messages that are repeated or received out-of-order, yet it cannot detect lost messages unless timeouts are used. While message ordering and consequentiality can be verified using QCFSA constructs, timeouts must be implemented at the program level, and a property would have to detect their occurrence indirectly. Thus, a similar effect could be achieved by setting a timer within the system under test prior to communication and instrumenting its associated event handler function. The event would then have a corresponding transition within the QCFSA that leads to a bad state. Alternatively, one could extend the QCFSA model to support guards with clocks, which could then be translated to timed events within DATEs.

7.4.3 Combining Runtime Verification and Testing Automata

While QCFSAs were designed for use within QuickCheck, this project has demonstrated that they can be successfully repurposed for runtime verification. Ideally, properties should be written in a form which can be easily translated into the input language of different verification tools. This promotes modularity by freeing one from the confines of a single verification approach.

[RBJ00] proposes *Input Output Symbolic Transition Systems* (IOSTS), which extend Labelled Transition Systems (LTS) by allowing the use of parameters and variables over transitions, as a base logic for writing properties which could then be translated and fed to different test case generation and verification tools. The extensions allow parameters to be handled symbolically, which can avoid the invalidation of certain testing techniques

[2] AspectJ is also used within the LARVA framework [Col08] to interleave monitoring code with the system under test.

that would otherwise fail due to the state space explosion. Test generators can thus make use of symbolic values combined with techniques such as constraint propagation and static analysis.

An IOSTS consists of an initial location, transitions and input, output and internal actions. Locations can be *pass*, *inconclusive* or *failing*. Transitions between locations are guarded, and consist of an action, a list of data types corresponding to the related function's argument list, and a set of variable assignment operations. A specification is an IOSTS in its initial configuration, with its initial variables bound. While the QCFSA to DATE translation described in this document requires an intermediate step in which the input automaton is rendered total (Section 4.4.2), IOSTS are implicitly complete, and employ a different notion of totality. Actions for which there are no outgoing arcs are consumed without changing the current state, and arcs whose guard condition fails trigger a transition into a *reject* state. The rationale given is that a test purpose should only focus on specifying a correct behaviour, whereas completeness should be handled by the test case generation tool. Thus, when generating test cases, an automaton must be *input complete*, and for every state and value pair, one should be able to find a next state without performing internal actions. Non-determinism is removed heuristically for the common case, or non-deterministic choices are postponed to as late a stage as possible.

A conformant trace is one which produces equivalent output when used to traverse a specification and an implementation. A test case is considered to be correct with respect to a test purpose and an implementation if it is sound, relatively complete, accurate and conclusive. A sound trace conforms to an implementation, and relative completeness assures that non conformance is detectable. An accurate trace is one which succeeds and is within the test purpose, and a conclusive trace is one which, if extended, would return an inconclusive result. Although the generator attempts to minimise test cases using model checking and invariant detection techniques, [RBJ00] states that the reduction phase requires further investigation. The difficulty in writing test purposes is also noted, and that test cases should ideally be generated from program abstractions rather than concrete specifications.

Additional logics may also be eligible for writing properties which are used both as test case generators and runtime monitors. For example, [ABG+05] developed several verification techniques based on the use of the discrete temporal logic Eagle. Eagle is expressive in that it allows other temporal logics, such as LTL, to be embedded within it, and yet it aims towards being simple so as to avoid the high computational cost associated with the verification of more expressive properties. Consequently, a property's computational cost depends on the complexity of the encapsulated logic and the property's size. An input property can 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* model checker, which is extended with symbolic execution capabilities. For runtime verification, Eagle properties are used to derive monitors that examine parametrised events.

The system under test is instrumented using a combination of *code wrapping* and *insertion*. Wrapping involves replacing calls within the SUT with calls to wrapper functions, while insertion is used to directly insert event-generating code into the source or object code. The latter approach is the one that was used by this project, as it only demands

the instrumentation of the functions that must be observed, and no other changes to the program have to be made. The Eagle framework also requires that any values on which a subsequent computation within a trace depends are initialised beforehand. This is also a requirement of QCFSA properties, as the initial state data must be precomputed or set prior to its use.

7.4.4 Alternative Model-Based Test Case Generation Logics

QuickCheck Finite State Automata were chosen as the base property's representation for several reasons which were discussed earlier, yet other model-based test case generation tools exist. Gast [KP03] is a tool similar to QuickCheck, which allows the automatic testing of reactive systems using properties over functions and data types expressed using first-order logic or an LTS. Properties are functions defined in the CLEAN functional programming language, and Gast automatically drives tests using generators for creating input values. A property fails if at least one failing counterexample is found. Support for LTS input properties is obtained by transforming their description into CLEAN, embedding its logic into the property language and executing it. This differs from the approach taken by QuickCheck, which provides support for automata through a separate library that dictates the precise structure and semantics of QCFSAs.

An LTS can be non-deterministic, that is, several transitions may be eligible for traversal from a given state under the same input. Non-determinism is handled during automaton traversal by storing the current state as a list of possible states. While this might suffice when testing generated traces of a short, finite length, it may preclude the logic's translation into a runtime monitor. This is because monitors can be long-lived (such as when verifying server processes), and successive non-deterministic choices could cause them to run out of memory.

Input sequences are generated by traversing the LTS. For deterministic automata, testing prefixes of a path known to be valid will not uncover additional errors. This reduces the number of generated test cases, and encourages the generation of longer and more meaningful input sequences. In addition, the traversal avoids generating paths which have already been produced. Input sequences are terminated and tested if their continuation would require taking a transition too often, otherwise the system attempts to proceed with the generation of longer traces. Testing is black box, and only functions' outputs are observed. The method requires that the system under test is *input-enabled*, that is, its state must be specified for any input in any reachable state, although states can be marked as being invalid.

Gast differs from QuickCheck primarily in its approach to generating values, as it explores the input space systematically and avoids repeating test cases. Given a finite input space and sufficient computational resources, a Gast property can be verified over all possible input values without resorting to full enumeration, and the property will have been proven. Yet for large type domains, Gast will normally resort to a semi-random search, first checking the common border cases and small input values and moving on to larger inputs, until a set number of tests are executed without a counterexample being encountered, as is the case with QuickCheck. The systematic exploration of the state space

may help in refining runtime monitoring, as tested behaviours can be safely excluded from the monitor's test set, reducing unnecessary monitoring.

7.4.5 Automatic Property Generation and Testing in Erlang

While this project has focused primarily on combining testing and runtime monitoring, there are other facets of the testing process which can also be unified. The *ProTest* project [DWA+10] 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. As with Daikon, invariance alone does not imply relevance. 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 [GH01].

7.5 Conclusion

This chapter served to evaluate the procedure adopted by this project by analysing the results obtained when translating concrete examples into runtime monitors. It highlighted the primary issues related to context, event interference and invalid initial state data. It also described the conditions under which the translation will fail, making recommendations which should be adhered to when constructing an input QCFSA property. Other approaches towards the translation of models into different representations were also analysed. The next chapter provides a final summary of the results presented, as well as possible avenues for future research.

Chapter 8

Conclusion

The following chapter concludes the project, describing the core results and possible directions for future research.

8.1 Summary

The project's main aim was to investigate the integration of testing and runtime verification by demonstrating a suitable and effective procedure for translating testing properties into runtime monitors. The source and target logics used were *QuickCheck Finite State Automata* (QCFSAs) and *Dynamic Automata with Timers and Events* (DATEs). These automata-based logics differ in that QCFSAs incorporate both a partial model of the system and a property specifying its correct behaviour, whereas DATEs only serve as classifiers. The translation dissected the QCFSA structure, using the model component to define the set of behaviours over which a verdict should be produced, that is, the *testable trace* set, and the classifying property to characterise the *negative trace* set.

While it was proven that the translation preserved the original property's negative trace set, the process assumed that all and only events relevant to that property are received by the generated monitor, and that the automaton's internal state reflects the live system's state. During the case study, mechanisms for inferring *context* were examined. Through context, the system was able to separate interleaved traces and forward them to their respective monitor, yet the construct was unwieldy and required detailed knowledge of the environment within which the monitors were to operate. Language-level support for context would help solve many of its associated problems, and would avoid the use of ad hoc monitor creation and binding policies. Ensuring consistency between the automaton's internal state and the system's actual state is harder, as an automaton's initial state is typically populated with test values which would then be used by the generated test cases. Thus, for QCFSAs to be translated correctly, they must adhere to a set of recommendations identified within the previous chapter.

In summary, this project has achieved the following aims:

- it provided an overview of testing and several techniques that can be used to automate different testing processes, notably property and test case generation
- it introduced the concept of a property's set of testable and negative traces
- it defined a formal model of QuickCheck automata, as well as a procedure for rendering automata total based on the notion of testable and negative trace sets as applied to QCFSAs
- it described a formal procedure for translating QCFSA's interval-based events into point-based entry and exit events, which was then shown to preserve the input property's negative trace set through the use of unique event identifiers
- it described an Erlang implementation of the property translator and a runtime verification framework for monitoring DATEs, outlining any pertinent implementation issues
- it demonstrated the translation of four properties designed for testing *Riak*, and analysed the deployment of the generated monitors in the environment of the system under test
- it investigated the problem of relating a monitor to a context without explicit property-level constructs or directives.

8.2 Future Work

8.2.1 Event Logging and Statistics

Within the current framework, all events are handled by a single arbiter process, which reroutes messages to the relevant monitors. This arbiter has a global view of events as they occur within the system, and currently maintains an event log. This log could potentially be exploited to derive statistical information, such as the frequency with which an event fires as well as system failure rates. Such statistical information could then be tapped for further modulating runtime verification and testing. In the former case, one could opt to forego the monitoring of certain event sequences which are judged to have proven themselves consistently valid. This could be achieved either by making the preconditions more selective and contracting the automaton's testable trace set, or by simply omitting the relevant transitions, perhaps eventually switching off the monitor entirely. Similarly, one could use frequency analysis during test case generation to direct testing by associating weights with transitions, discouraging the traversal of certain automaton branches.

8.2.2 From Runtime Verification to Testing Automata

This project has focused primarily on the translation of testing automata into runtime verification monitors, yet the reverse operation is also of interest. By automating the

translation of runtime verification properties into testing automata, one could move between representations given a property written in either notation. In conjunction with statistical analysis of logged events, the framework could automatically fine-tune the verification process by updating properties on-the-fly based on the derived information.

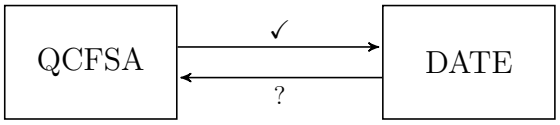

Figure 8.1: From QuickCheck Automata to DATEs ... and back?

The process may be hampered by the difficulty in moving from point-based events to intervals. Since not every transition in a user-supplied DATE will necessarily form a matching pair of entry and exit events, problems will arise when trying to discriminate between recursive calls and top-level consecutive calls to the same function. Consequently, the translation process may also require analysis of the control flow of the system under test.

8.2.3 Channel Communication Analysis

Events are currently assumed to correspond directly to function calls, yet the system may be extended to monitor channel communication between processes. While this could be implemented within the runtime verification monitor by adding a *communication event* type, special constructs may have to be added to the QCFSA notation, which currently assumes that transitions correspond solely to function calls. To leave the current framework unmodified, one could wrap channel communication operations within functions. Special considerations may also have to be made due to the possible introduction of *side-effects*, which could also affect monitoring.

8.2.4 Temporal Properties

DATEs support temporal properties through the use of real-valued clocks and timer operations. As QuickCheck's temporal property verification capabilities are broadened, the translation procedure may be augmented to support their verification.

8.3 Concluding Note

Verification is a field that has much to gain from increased automation, which can decrease human involvement and improve interoperability between verification tools, leading to more thorough and consistent results. Identifying additional translations focuses testing further. By devising translations to refashion models into inputs for different verification tools, one can improve synergism between the various test processes employed, thus amplifying monitoring reliability.

Appendix A

Erlang

A.1 Introduction

The following is an overview of *Erlang*, a multi-paradigm programming language with a strong focus on concurrency and distributed programming. The primary constructs of the language and its approach to concurrency, especially *interprocess communication*, are illustrated, ending with a description of *behaviours* and *generic server* processes.

A.2 Overview

Erlang[1] is a declarative programming language with functional programming concepts which was designed for the creation of highly-concurrent applications. It facilitates the programming of distributed systems in a quick and efficient manner. It also contains mechanisms for implementing *fault tolerance*, increasing applications' robustness. Threads are managed by an underlying virtual machine. Memory allocation is automatic, with variable deallocation being handled by a garbage collector. [CT09]

Erlang is based on the use of concurrent *lightweight threads*, which communicate exclusively through *asynchronous message passing* and *signals*, the latter being an event-based variation of the former. These alleviate the need for shared memory and locks. Erlang was originally developed for applications within telecommunications, yet nowadays it is being used in other problem domains. [CT09] claims that this is primarily due to its concurrency model as well as its fault tolerance capabilities, which result in code that is often both clearer and more compact than the equivalent program implemented in another high-level language such as C.

[1] Project website: `http://www.erlang.org` (last accessed July 2011)

A.3 Basic Concepts

The following section introduces the fundamental language constructs used in Erlang, which are used throughout examples presented within this document.

A.3.1 Variables and Atoms

Erlang employs a *dynamic* type system [CT09], where variable data types are determined at runtime. Assignment, comparison and retrieval operations on variables are performed using pattern matching. Variables are initially *unbound*, and can only be assigned a value once. The interpreter will attempt to match subsequent assignment values with the existing bound values, generating an exception on failure.

Listing A.1: Assignment operation

```
1  A = 12,     % First assignment succeeds and returns 12
2  A = 12,     % Succeeds (value matches first assignment)
3  A = 24.     % Fails and generates an exception
```

Sequential composition of statements is denoted by the `,` operator, while `.` signifies the end of an execution block. Variable names must begin with an uppercase letter. Variables can be bound to data of any type, including integers, lists, tuples, atoms and functions.

The *atom* data type [Eri10e] is used extensively in Erlang when representing named entities. Atoms serve as constant literals and perform a role similar to that of *enumerations* in other programming languages. Atoms must be declared with a leading lowercase letter, and can be encapsulated within single quotes. Atoms only support one operation, namely *comparisons* with other atoms. While the role of atoms could in theory be fulfilled using integers or some other enumeration, they help in making code more legible and are also handled more efficiently than the former.

A.3.2 Tuples, Lists and List Comprehensions

Tuples

Tuples [Eri10c] are a composite data type with a fixed number of elements of a possibly varying type, delimited by { }. Different forms of data are often represented using tuples with an atom as their first element, which serves to describe that tuple's data type. Such atoms are known as *tags* [CT09]. Instead of using tags, one could declare a data type using *records* [Eri10e]. Record structures are declared using the `-record` compiler directive, and are populated using `#recordname`, as shown in Listing A.2.

Listing A.2: Defining and populating records

```
1  % Declaring the 'person' record structure
2  -record(person, {name, surname}).
3
4  ...
5  % Assignment
6  Guy1 = #person{name="Joe", surname="Random"},
7
8  % Same as 1 (order not important)
9  Guy2 = #person{surname="Random", name="Joe"},
10
11 % Equivalent using tuples
12 Guy3 = {person, "Joe", "Random"}.
```

Lists

Lists are Erlang's primary collection type. Lists are sequences of objects, which need not necessarily be of the same type. Lists are delimited by `[ ]`, with individual elements in a list separated by commas. Non-empty lists can be divided into a *head* and a *tail*, head being the first element and tail being the remainder of the list. The split between a list's head and tail is delimited by the *constructor* operator |. The tail can itself be a list. Lists with an empty list as a last tail element are said to be *well-formed* [CT09].

Listing A.3: Examples of equivalent lists of atoms

```
1  A = [a|[b]],
2  A = [a,b],
3  A = [a,b|[]].
```

Several standard list operations are pre-defined within the `lists` module (modules are described in Section A.3.4. [Eri10f] enumerates and describes each function within this module, which includes

`append(`**L1**, **L2**`)`	Returns a list **L1** ++ **L2**
`reverse(`**L**`)`	Returns a list with all the elements of **L** in reverse order
`foreach(`**Fun**, **L**`)`	Applies function **Fun** on each element in **L**
`sort(`**L**`)`	Returns **L** with its elements sorted

List Comprehensions

List comprehensions allow one to map functions onto each element of a list and to filter elements based on some specified criteria. The general form of list comprehensions is given as

[**Expression** || **Generators** and/or **Guards**]

Generators are of the form **Pattern** ← **List**. Each element in the list is bound to a specified pattern, which can then be used in other list comprehension components.

Guards are boolean expression which determine whether an element should be included within the result list, typically based on the value of the patterns bound by generators.

Expression is the element which will be placed into the resultant list.

Listing A.4: List comprehension returning all prime numbers up to N

```
1 Primes = range(2,N) -- [X*Y || X <-range(2,N), Y <-range(2,N)].
```

Listing A.4 is an example using list comprehensions which produces a list of primes up to a given number **N**. The function `range(N,M)` produces a list containing all the integers between **N** and **M** inclusive, and is defined in Section A.3.4. Alternatively, one could use the `seq()` function which is provided with the standard library under the *lists* module [Eri10f]. The list comprehension produces the products of all possible pair combinations of integers between 2 and **N**. The resultant list is then removed from the list of all numbers from 2 to **N**, leaving only the primes[2].

A.3.3 Preprocessor

Erlang allows the creation of header files and *macros*. Macros allow text to be substituted at the pre-compilation stage. Macros are defined using the `-define` directive, as in

`-define(`**ORIGINAL**, **replacement**`)`

For replacement to occour, tokens in the source file must be preceded by a '`?`' character. Thus, using the previous declaration, code containing **?ORIGINAL** would be replaced with **replacement**. There are also several standard macros, such as **?MODULE** (translates to the name of the module within which the token exists), which are pre-defined in Erlang. [Eri10e].

A.3.4 Program Structure and Control Flow

Functions

Programs in Erlang consist of *functions*, with related functions typically grouped into single *modules*. Functions that are to be exposed to external modules should be marked

[2] 1 is not considered prime.

by listing them within the `-export()` compiler directive. Modules can be replaced and loaded at runtime. Function names are considered as atoms. [CT09]

When invoking a function, Erlang uses pattern matching to compare the supplied function arguments with the parameters defined in the function's signature. Consider the definition of the function `invert()` in Listing A.3.4:

```
1 invert(true)  % Input == true
2   -> false;     % return false
3 invert(_)     % Input == anything else
4   -> true.      % return true
```

The function is composed of two *clauses* which are checked sequentially, starting from the top definition. A function returns the value of the last expression executed within its body. When passed the atom **true**, the function returns **false**. If this is not the case, the second function is considered. The parameter _ is a *wildcard* and matches any input, discarding its value. Thus, the function will always return **true** should the first comparison fail. Functions are terminated with a period. If none of the clauses were to match, the system would generate a `function_clause` runtime error [Eri10c]. Functions are called *by value*, meaning that arguments given to a function are evaluated before the function is invoked, and the function will operate on local copies of the arguments. Local variables in a function are unique to each invocation, and their bindings are not maintained between calls [CT09].

Erlang supports *higher-order functions*, where functions can be defined, passed as parameters, assigned to variables and subsequently executed. Anonymous functions are defined using the `fun()` keyword [Eri10e].

Listing A.5: Example using Higher-Order Functions

```
1 map(_, []) ->
2   [];
3 map(Func, [H|Tail]) ->
4   [Func(H) | map(Func, Tail)].
5
6 test() ->
7   L = map(fun(X) -> X * X end, [1,2,3,4,5]).
```

Listing A.5 is an example showing an application of higher-order functions. It defines the function `map(`**Func**, **List**`)`, which applies function **Func** on each element in **List** and places the return values in a result list (an implementation of `map()` can be found in the *lists* module [Eri10f]). The `map()` function makes use of *recursion*. An anonymous function that squares a given value is declared and passed as a parameter to `map()`, along with a list on which the function will be applied. Each element of the list is passed as an argument to the function, which then maps the given value with the placeholder variable used when defining the function.

Recursion

Recursion is a technique for diving a large problem into smaller and simpler tasks. A function is recursive if it invokes itself at some point in its execution. Recursion is often used as a technique for iterating through lists, performing an operation on each element with every iteration and calling the same function with the remaining elements as an argument. When creating recursive functions, one often defines a *trivial* or *base* case which stops the function from recurring. Without a base case, a function would keep recurring, which can sometimes be the desired behaviour as in the case of server processes.

[CT09] states that when implementing *direct recursion*, a function could be implemented in such a way that each intermediate call's stack frame would have to be saved until the recursion has reached its base case. For example, implementing a function which generates a list of all numbers within a range as

```
1 range(N,M) when N > M -> % Reverse values
2   range(M,N);
3 range(N,N) -> [N];
4 range(N,M) -> [N] ++ range(N+1,M).
```

would require that `range()` be unravelled to its full depth, after which the generated lists are appended.

Another alternative described is to use *tail recursion* [CT09]. This makes use of an *accumulating parameter*, whereby instead of expressing the result in terms of an operation between the current stage and the subsequent recursive call's return value, the local result is calculated and stored in an accumulator and the recursive call is invoked at the end of the function clause. Thus, a tail-recursive version of `range()` would be as follows

```
1 range1(From, From, Res) ->
2   lists:reverse([From | Res]);
3 range1(From, To, Res) ->
4   range1(From + 1, To, [From | Res]).
5
6 range(From, To) when From > To ->
7   range(To, From);
8 range(From, To) ->
9   range1(From, To, []).
```

The `range()` function initialises the accumulator as the empty list and invokes a helper function `range1()`. This then iterates recursively, adding the current value to the accumulator at each stage and invoking the function anew. Although the order of elements in the resultant list was not specified as a formal requirement, the final call to the terminating clause returns a reversed version of the accumulator, since the values are appended to the front of the accumulator during the iterations and are thus in reverse order.

While tail recursion should theoretically be more efficient than its direct-recursion counterpart (as it avoids having to allocate a new stack frame with each call), recent

releases of Erlang include optimisations which often make it no longer the case. When opting for high performance, [CT09] suggests that one tries the different options and chooses the adequate one after measuring their characteristics.

A.4 Concurrency

Erlang has a strong focus on concurrency, allowing massively parallel systems to execute efficiently. Erlang threads are actually lightweight processes, as they do not share memory directly. Data exchange and synchronisation between threads is performed through *message passing*, eliminating the need for explicit locks or memory barriers. This makes Erlang a good candidate for developing software on *multi-core* platforms, as parallelism is inherent in the paradigm and is automatically catered for. [CT09]

A.4.1 Threads

Threads are created using the `spawn()` function, which takes a function name and a list of arguments as a parameter and returns a *Process Identifier* (PID) [Eri10d]. This creates a process that executes the specified function, with the original caller's thread continuing with its execution from after the `spawn` command. A PID can also be mapped to an arbitrary atom using the `register()` function [Eri10d]. Once an atom is registered, it can be used to refer to the process endpoint in the same manner as a PID would, with some caveats, as will be described in Section A.4.2.

The two basic communication operations are *sending* and *receiving* messages, which operate as follows.

Sending is an *asynchronous* operation [CT09], where the sending process transmits a packet of information such as an atom, tuple or any valid variable, and continues with its execution. This data is then placed on the destination process' receive queue. The notation for sending atom **a** to process **P** is

$$\mathbf{P} \mathbin{!} \mathrm{a}$$

, where **P** is a PID or a registered process name. A process can retrieve its own PID using the `self()` function. Sending messages to a specified PID never fails, even if the process does not exist. Conversely, if a message is sent to an unregistered process, an exception will be thrown. [Eri10e]

Receiving causes a listening process to block until at least one element is placed on its receive queue. The process then handles the incoming messages selectively based on the conditions specified by its `receive` clause, which defines a series of guarded conditions to which the input elements are compared. If a condition is matched, the related conditional branch is pursued. The general form for the `receive` construct is

```
receive
  atom →
    % Matches value 'atom'
  ...
  Any →
    % Binds input value to 'Any' end
```

Figure A.1 illustrates an example of two sending processes communicating with a single server process. In this example, the server process waits for either an **a** or a **b** atom and outputs the respective received value to standard output. The server process will execute indefinitely, as once it receives either element, it invokes the function once again. Otherwise, the process would terminate once execution fell through. If either process were to send an element other than one that matches the guards (such as the atom **c**), the process would simply move to the next item in its receive queue [CT09]. If there are no remaining unprocessed elements on the receive queue, then the process will block once again.

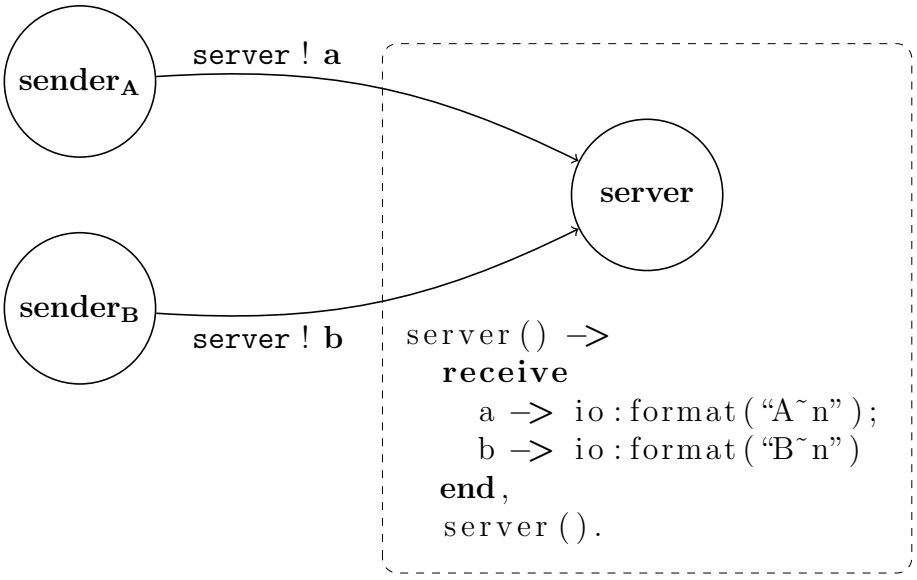

Figure A.1: Two sender processes transmitting to a single server process

A `receive` block can also specify a *timeout* in milliseconds using the `after` guard clause. This will cause the process to resume and execute the associated code block should the process block for the specified amount of time. Timeouts are often used when waiting on external events so as to avoid blocking indefinitely. [Eri10e]

Signals

Apart from simple interprocess communication via channels, Erlang also supports *signals*. Processes can be linked to other existing processes using the `link()` function. On termination, a process sends a signal to every process to which it is linked. Processes can

terminate normally once their associated function returns. Alternatively, a process can force termination using the `exit()` [Eri10e] function, which also takes a parameter which specifies the reason for exiting. Linked processes can then perform an action based on the terminating process' exit status.

Signals are often employed for monitoring purposes, with arbiter processes (typically the parent thread which launched the child processes) listening on other processes' termination codes and ensuring that faults and abnormal terminations are handled correctly. One could also create a hierarchy of such supervising processes.

A.4.2 Distributed Systems

Syntactically, extending a concurrent application to allow distribution across a network of nodes requires a few minor modifications. These are described by [Eri10e] as follows:

- Each node on the network must be assigned a unique name which will be used when specifying a channel endpoint.
- When sending a message to a process registered on another node, the destination is changed from a single identifier (as described in Section A.4.1) into a tuple of the form

  {**registered_process_name**, **node_name**} ! data

  If the process' PID is known, then a message can be sent without specifying the node name, as the PID data structure also contains information related to the process' location.
- Nodes that would like to communicate with each other must all share an agreed-upon password, which is then stored in a file on each node within a *magic cookie*.

A node can obtain its own node name by using the `node()` function. Additionally, processes within a distributed system can choose to spawn new threads remotely on another node within the network by using a variant of the `spawn()` [CT09] function. This is invoked in the same manner as when programming for single-node systems, except that the target node name must also be supplied. The spawned process' standard input and output are automatically rerouted to the parent node.

A.4.3 Generic Servers

Certain applications and their implementations follow similar patterns, and in some cases, these patterns may be generalised into common *behaviours*. Identifying general behaviours simplifies the programming effort by allowing the common elements of programs to be reused, thus offering an increase in productivity and robustness.

One behaviour implemented by the Erlang OTP library [Eri10d] is that of a *generic server* (`gen_server`). Many server processes follow the same pattern, listening on an incoming connection and performing an action based on the contents of the received message and the current server state. Thus, given a description of how to handle incoming messages, Erlang can generate a server which will process these messages and react to them in the specified manner. One need only supply the automaton's transition functions and actions as *callback functions*, and Erlang will manage the creation of all the boilerplate code related to communications, event monitoring and message management.

Function	**Operation**
`start_link`	launches the server process, returning a handle to the server
`call`	sends a synchronous event (request) from a client to a specific `gen_server`
`cast`	similar to `call`, but asynchronous
`abcast`	sends a multicast to one or more registered `gen_servers`

Table A.1: Core generic server functions

Table A.1 lists the essential functions that are exposed by a module implementing the generic server behaviour. Calling any of these functions will result in a message being sent to the `gen_server` in question. This message is interpreted by the server, and the relevant implemented callback function is invoked. The core functions that a callback module must implement are:

1. `init`, which initialises the process' state and is invoked through `start_link`
2. `handle_call`, which defines how the server should react to a given event generated through `call` and how the internal state should be updated
3. `handle_cast`, which fulfils the role of `handle_call` for asynchronous messages sent through casts

A.4.4 Generic Finite State Machines

Generic Finite State Machines, or `gen_fsms`, are similar in some ways to `gen_servers`, in that they are based on the use of a callback module which is invoked from a standardised and automatically-generated engine. Generic FSMs are used to implement finite state automata within Erlang. Automata are created using `start_link`, which spawns a new `gen_fsm` process initialised using the state data specified. An event **E** can be sent asynchronously to automaton **A** by calling `send_event`(**A**, **E**) function. If **A** is implemented within module **M**, then the automaton will invoke function **M**:**StateName** with **E** as an argument, where **StateName** is the name of the automaton's current state (it follows that a function must be defined for every state within the automaton). Each state function must return a tuple which specifies the automaton's next state. An automaton also opt to stop, in which case the process is terminated. On entering the new state, the `gen_fsm` blocks until the next event is received. Generic servers, FSMs and other behaviours are described in greater detail in [Eri10f].

A.5 Conclusion

This chapter has provided an overview of the fundamentals of Erlang. It is by no means comprehensive, limiting itself to those language features that directly concern this document. For a more thorough description of the language, one may refer to [CT09], or the online language reference [Eri10c] and manual [Eri10d].

Bibliography

[AB05] Cyrille Artho and Armin Biere. Combined static and dynamic analysis. In *Proceedings of the AIOOL 05, ENTCS*, pages 98–115. Elsevier Science, 2005.

[ABG+05] Cyrille Artho, Howard Barringer, Allen Goldberg, Klaus Havelund, Sarfraz Khurshid, Mike Lowry, Corina Pasareanu, Grigore Rosu, Koushik Sen, Willem Visser, and Rich Washington. Combining test case generation and runtime verification. *Theoretical Computer Science*, 336:209–234, May 2005.

[ACH08] Thomas Arts, Laura M. Castro, and John Hughes. Testing erlang data types with quviq quickcheck. In *ERLANG '08: Proceedings of the 7th ACM SIGPLAN workshop on ERLANG*, pages 1–8, New York, NY, USA, 2008. ACM.

[AHJW06] Thomas Arts, John Hughes, Joakim Johansson, and Ulf Wiger. Testing telecoms software with quviq quickcheck. In *ERLANG '06: Proceedings of the 2006 ACM SIGPLAN workshop on Erlang*, pages 2–10, New York, NY, USA, 2006. ACM.

[AN08] Irem Aktug and Katsiaryna Naliuka. Conspec – a formal language for policy specification. *Electronic Notes in Theoretical Computer Science*, 197(1):45–58, 2008.

[AO08] Paul Ammann and Jeff Offutt. *Introduction to software testing*. Cambridge University Press, Cambridge, UK, 2008.

[AS85] Bowen Alpern and Fred B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, October 1985.

[AT10] Thomas Arts and Simon Thompson. From test cases to fsms: augmented test-driven development and property inference. In *Proceedings of the 9th ACM SIGPLAN workshop on Erlang*, Erlang '10, pages 1–12, New York, NY, USA, 2010. ACM.

[Bas11] Basho. The riak wiki. `http://wiki.basho.com/` (last accessed 19 March 2011), March 2011.

[BBCF08] F. Benigni, A. Brogi, S. Corfini, and T. Fuentes. Service contracts in a secure middleware for embedded peer-to-peer systems. In *Proceedings of the 2nd Workshop on Formal Languages and Analysis of Contract-Oriented Software (FLACOS)*, 2008.

[Ber07] 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.

[BHJP05] Johan Blom, Anders Hessel, Bengt Jonsson, and Paul Pettersson. Specifying and generating test cases using observer automata. In *Proc. 4 th International Workshop on Formal Approaches to Testing of Software 2004 (FATES04), volume 3395 of Lecture Notes in Computer Science*, pages 125–139. SpringerVerlag, 2005.

[BHL+07] Eric Bodden, Laurie J. Hendren, Patrick Lam, Ondrej Lhoták, and Nomair A. Naeem. Collaborative runtime verification with tracematches. In *Workshop on Runtime Verification*, pages 22–37, 2007.

[BJ03] Johan Blom and Bengt Jonsson. Automated test generation for industrial erlang applications. In *In ERLANG 03: Proceedings of the 2003 ACM SIGPLAN workshop on Erlang*, pages 8–14. ACM Press, 2003.

[BJPW99] Antoine Beugnard, Jean-Marc Jézéquel, Noël Plouzeau, and Damien Watkins. Making components contract aware. *Computer*, 32(7):38–45, 1999.

[BLS07] Andreas Bauer, Martin Leucker, and Christian Schallhart. Runtime verification for LTL and TLTL. *ACM Transactions on Software Engineering and Methodology*, 2007.

[Car10] Neal Carothers. The babylonian method: Examples. `http://personal.bgsu.edu/~carother/babylon/Examples.html` (last accessed 25 February 2010), 2010.

[CDH+00] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Pasareanu, and Hongjun Zheng. Bandera: Extracting finite-state models from java source code. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 439–448. ACM Press, 2000.

[CM05] Séverine Colin and Leonardo Mariani. *Model-Based Testing of Reactive Systems*, chapter 18 Run-Time Verification, pages 525–555. Springer, 2005.

[CM08] Jun Chen and Steve MacDonald. Towards a better collaboration of static and dynamic analyses for testing concurrent programs. In *PADTAD '08: Proceedings of the 6th workshop on Parallel and distributed systems*, pages 1–9, New York, NY, USA, 2008. ACM.

[Col08] Christian Colombo. Practical runtime monitoring with impact guarantees of java programs with real-time constraints. Master's thesis, University Of Malta, 2008.

[CPS08] Christian Colombo, Gordon J. Pace, and Gerardo Schneider. Dynamic event-based runtime monitoring of real-time and contextual properties. In *Formal Methods for Industrial Critical Systems (FMICS)*, volume 5596 of *Lecture Notes in Computer Science*, pages 135–149, L'Aquila, Italy, 2008.

[CR99] S.J. Cunning and J.W. Rozenblit. Automatic test case generation from requirements specifications for real-time embedded systems. In *1999 IEEE International Conference on Systems, Man, and Cybernetics*, volume 5, pages 784–789, Piscataway, N.J., 1999. IEEE Press.

[CR09] Richard Carlsson and Mickael Remond. Eunit - a lightweight unit testing framework for erlang. `http://svn.process-one.net/contribs/trunk/eunit/doc/overview-summary.html` (last accessed 26 May 2010), May 2009.

[CSH10] Koen Claessen, Nicholas Smallbone, and John Hughes. Quickspec: guessing formal specifications using testing. In *Proceedings of the 4th international conference on Tests and proofs*, TAP'10, pages 6–21, Berlin, Heidelberg, 2010. Springer-Verlag.

[CT09] Francesco Cesarini and Simon Thompson. *Erlang Programming: A Concurrent Approach to Software Development*. O'Reilly Media, 1005 Gravenstein Highway North, Sebastopol, CA 95472, first edition, 2009.

[DLvL06] Christophe Damas, Bernard Lambeau, and Axel van Lamsweerde. Scenarios, goals, and state machines: a win-win partnership for model synthesis. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, SIGSOFT '06/FSE-14, pages 197–207, New York, NY, USA, 2006. ACM.

[DWA+10] John Derrick, Neil Walkinshaw, Thomas Arts, Clara Benac Earle, Francesco Cesarini, Lars-Åke Fredlund, Victor Gulias, John Hughes, and Simon Thompson. Property-based testing: the protest project. In *Proceedings of the 8th international conference on Formal methods for components and objects*, FMCO'09, pages 250–271, Berlin, Heidelberg, 2010. Springer-Verlag.

[EFD05] Clara Benac Earle, Lars-Åke Fredlund, and John Derrick. Verifying fault-tolerant erlang programs. In *Proceedings of the 2005 ACM SIGPLAN workshop on Erlang*, ERLANG '05, pages 26–34, New York, NY, USA, 2005. ACM.

[Eri10a] Ericsson. Academic and historical questions. `http://www.erlang.org/faq/academic.html` (last accessed 14 March 2010), February 2010.

[Eri10b] Ericsson. Common test reference manual version 1.4.7. `http://www.erlang.org/doc/apps/common_test/` (last accessed 22 March 2010), February 2010.

[Eri10c] Ericsson. Erlang reference manual user's guide version 5.7.5. `http://www.erlang.org/doc/reference_manual/users_guide.html` (last accessed 14 March 2010), February 2010.

[Eri10d] Ericsson. Erlang run-time system application (erts) reference manual version 5.7.5. `http://www.erlang.org/doc/apps/erts/index.html` (last accessed 14 March 2010), February 2010.

[Eri10e] Ericsson. Getting started with erlang user's guide version 5.7.5. `http://www.erlang.org/doc/getting_started/users_guide.html` (last accessed 14 March 2010), February 2010.

[Eri10f] Ericsson. Stdlib reference manual version 1.16.5. `http://www.erlang.org/doc/man/STDLIB_app.html` (last accessed 14 March 2010), February 2010.

[FMPW04] Steve Freeman, Tim Mackinnon, Nat Pryce, and Joe Walnes. Mock roles, not objects. In *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 236–246, New York, NY, USA, 2004. ACM Press.

[GH01] Dimitra Giannakopoulou and Klaus Havelund. Automata-based verification of temporal properties on running programs. In *Proceedings of the 16th IEEE international conference on Automated software engineering*, ASE '01, pages 412–, Washington, DC, USA, 2001. IEEE Computer Society.

[GJ08] Alex Groce and Rajeev Joshi. Extending model checking with dynamic analysis. In *VMCAI'08: Proceedings of the 9th international conference on Verification, model checking, and abstract interpretation*, pages 142–156, Berlin, Heidelberg, 2008. Springer-Verlag.

[GKS05] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: directed automated random testing. *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, 40(6):213–223, June 2005.

[HBAA10] Hadi Hemmati, Lionel Briand, Andrea Arcuri, and Shaukat Ali. An enhanced test case selection approach for model-based testing: an industrial case study. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, FSE '10, pages 267–276, New York, NY, USA, 2010. ACM.

[HJK+11] Andreas Holzer, Visar Januzaj, Stefan Kugele, Boris Langer, Christian Schallhart, Michael Tautschnig, and Helmut Veith. Seamless testing for models and code. In *Proceedings of 14th International Conference on Fundamental Approaches to Software Engineering (FASE 2011)*, volume 6603 of *Lecture Notes in Computer Science*, pages 278–293. Springer, April 2011.

[HSRT08] A. Hoole, I. Simplot-Ryl, and I. Traore. Integrating contract-based security monitors in the software development life cycle. In *Proceedings of the 2nd Workshop on Formal Languages and Analysis of Contract-Oriented Software (FLACOS)*, 2008.

[HW05] Geng-Dian Huang and Farn Wang. Automatic test case generation with region-related coverage annotations for real-time systems. In Doron Peled and Yih-Kuen Tsay, editors, *Automated Technology for Verification and Analysis*, volume 3707 of *Lecture Notes in Computer Science*, pages 144–158. Springer Berlin / Heidelberg, 2005.

[JM97] Jean-Marc Jézéquel and Bertrand Meyer. Design by contract: The lessons of ariane. *Computer*, 30(1):129–130, 1997.

[JuRJBP07] Muhammad Jaffar-ur Rehman, Fakhra Jabeen, Antonia Bertolino, and Andrea Polini. Testing software components for integration: a survey of issues and techniques. *Software Testing, Verification and Reliability*, 17(2):95–133, 2007.

[JVSJ06] Pankaj Jalote, Vipindeep Vangala, Taranbir Singh, and Prateek Jain. Program partitioning: a framework for combining static and dynamic analysis. In *WODA '06: Proceedings of the 2006 international workshop on Dynamic systems analysis*, pages 11–16, New York, NY, USA, 2006. ACM.

[KMM07] Ingolf H. Krüger, Michael Meisinger, and Massimiliano Menarini. Runtime verification of interactions: from mscs to aspects. In *Proceedings of the 7th international conference on Runtime verification*, RV'07, pages 63–74, Berlin, Heidelberg, 2007. Springer-Verlag.

[KP03] Pieter Koopman and Rinus Plasmeijer. Testing reactive systems with gast. In Stephen Gilmore, editor, *Trends in Functional Programming 4*, pages 111–129. Intellect Books, September 2003.

[LPSS09] A. Lomuscio, W. Penczek, M. Solanki, and M. Szreter. Runtime monitoring of contract regulated web services. In *Proceedings of the 12th International Workshop on Concurrency, Specification and Programming (CS&P09)*, 2009.

[LQS08] Alessio Lomuscio, Hongyang Qu, and Monika Solanki. Towards verifying contract regulated service composition. In *ICWS '08: Proceedings of the 2008 IEEE International Conference on Web Services*, pages 254–261, Washington, DC, USA, 2008. IEEE Computer Society.

[LS09] Martin Leucker and Christian Schallhart. A brief account of runtime verification. *Journal of Logic and Algebraic Programming*, 78(5):293–303, May 2009.

[Mey92] Bertrand Meyer. Applying "design by contract". *Computer*, 25(10):40–51, 1992.

[MFC01] Tim Mackinnon, Steve Freeman, and Philip Craig. *Endo-testing: unit testing with mock objects*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[MH03] Vincent Massol and Ted Husted. *JUnit in Action*. Manning Publications Co., Greenwich, CT, USA, 2003.

[NE02] Jeremy W. Nimmer and Michael D. Ernst. Automatic generation of program specifications. In *ISSTA '02: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, pages 229–239, New York, NY, USA, 2002. ACM.

[PA09] Javier Paris and Thomas Arts. Automatic testing of tcp/ip implementations using quickcheck. In *ERLANG '09: Proceedings of the 8th ACM SIGPLAN workshop on ERLANG*, pages 83–92, New York, NY, USA, 2009. ACM.

[PE05] Carlos Pacheco and Michael D. Ernst. Eclat: Automatic generation and classification of test inputs. In *19th European Conference on Object-Oriented Programming*, pages 504–527, 2005.

[PMBF05] Patrizio Pelliccione, Henry Muccini, Antonio Bucchiarone, and Fabrizio Facchini. TeStor: Deriving test sequences from model-based specifications. In George Heineman, Ivica Crnkovic, Heinz Schmidt, Judith Stafford, Clemens Szyperski, and Kurt Wallnau, editors, *Component-Based Software Engineering*, volume 3489 of *Lecture Notes in Computer Science*, pages 267–282. Springer Berlin / Heidelberg, 2005.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *SFCS '77: Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, pages 46–57, Washington, DC, USA, 1977. IEEE Computer Society.

[PPW+05] A. Pretschner, W. Prenninger, S. Wagner, C. Kühnel, M. Baumgartner, B. Sostawa, R. Zölch, and T. Stauner. One evaluation of model-based testing and its automation. In *Proceedings of the 27th international conference on Software engineering*, ICSE '05, pages 392–401, New York, NY, USA, 2005. ACM.

[Quv10] Quviq AB. *QuickCheck Documentation Version 1.191*, March 2010.

[RBJ00] Vlad Rusu, Lydie Du Bousquet, and Thierry Jeron. An approach to symbolic test generation. In *Proceedings of Integrated Formal Methods*, pages 338–357. Springer Verlag, 2000.

[SC07] Yannis Smaragdakis and Christoph Csallner. Combining static and dynamic reasoning for bug detection. In *Proceedings of the 1st International Conference on Tests And Proofs (TAP)*, pages 1–16. Springer, 2007.

[vRDGT08] Robbert van Renesse, Dan Dumitriu, Valient Gough, and Chris Thomas. Efficient reconciliation and flow control for anti-entropy protocols. In *Proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware*, LADIS '08, pages 6:1–6:7, New York, NY, USA, 2008. ACM.

[ZH02] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, 2002.