polyLARVA plugin for Erlang

Ivan Galea

Supervisor: Mr. Christian Colombo
Co-supervisor: Ms. Ruth Mizzi

Faculty of ICT

University of Malta

16 April 2013

Submitted in partial fulfillment of the requirements for the degree of B.Sc. I.C.T. (Hons.)

Faculty of ICT

Declaration

I, the undersigned, declare that the dissertation entitled:

polyLarva plugin for Erlang

submitted is my work, except where acknowledged and referenced.

Ivan Galea

17 April 2013

Acknowledgements

I would like to thank my supervisor, Christian Colombo, and co-supervisor, Ruth Mizzi, for the continuous support and patience throughout the development of this thesis.

Abstract

Runtime verification is a technique used to verify the correctness of software behaviour at run-time as described in a formal specification. In general this requires the concept of system events, which means that a system running under runtime verification must fire system events during execution which exist for the purpose of making the system behaviour detectable. Additionally a runtime verification tool may also execute actions on a monitored system normally upon the detection of incorrect behaviour. These requirements suggest that a runtime verification tool cannot be entirely independent of the programming language used to implement a system to be monitored.

polyLARVA is a runtime verification framework which is designed to support multiple programming languages by separating generic functionality from the functionality that is specific to a particular language. This is achieved by creating a remote-side monitor and a system-side monitor where the remote-side is completely independent of the programming language of the monitored system while the system-side monitor is specific to a particular language.

In this work, polyLARVA is extended to support the Erlang language. This involved defining an appropriate specification to define behaviour of Erlang programs and using this specification to generate a system-side monitor for Erlang which is incorporated into the system using aspect-oriented programming techniques. The generated monitor detects Erlang events and communicates with a remote-side monitor to check for correctness accordingly.

The resulting artifact is used to monitor Riak, an open-source database written in Erlang, such as to determine the effectiveness of the solution on a real-world example. Riak's implementation is investigated in depth to define aspects of its behaviour and monitor them to determine to the usefulness of the artifact.

Contents

1. Introduction **1**
1.1 Report Structure 4

2. Background **5**
2.1 Erlang 5
2.1.1 Actor model 5
2.1.2 Basic Types 6
2.1.3 Variables 7
2.1.4 Erlang tracing 7
2.1.5 Parse transformation 8
2.2 Aspect-oriented programming 9
2.3 Runtime Verification 10
2.3.1 Intrusive vs non-intrusive monitoring 11
2.3.2 Synchronous and Asynchronous monitoring 11
2.4 polyLarva 12
2.4.1 Monitoring Boundaries 13
2.4.2 Specification 14
2.5 Elarva 21
2.6 Conclusion 21

3. Problem Definition **22**
3.1 Aim and Objectives 23

4. Design **26**
4.1 Monitoring control flow 26
4.2 Approaches to elicit events 27
4.2.1 Using Erlang tracing 27
4.2.2 Using AOP 28
4.2.3 Conclusion 29
4.3 Erlang events 29
4.4 Specification script 32
4.4.1 System-side monitoring state 32
4.4.2 Events block 36
4.5 Conclusion 39

5. Implementation **40**
5.1 AOP Library . . . 40
5.1.1 Combining the two implementations . . . 43
5.2 Erlang plugin . . . 43
5.2.1 Generating AOP Code . . . 44
5.2.2 Generating Monitor context code . . . 46
5.2.3 Monitor communication . . . 48
5.3 Logging . . . 51
5.4 Suggestions for polyLarva . . . 51
5.5 Conclusion . . . 52

6. Evaluation **53**
6.1 Introduction to Riak . . . 53
6.1.1 Basic Functionality . . . 54
6.1.2 Riak key-value stack . . . 55
6.1.3 Topology . . . 56
6.2 Properties about Riak . . . 57
6.2.1 Property 1: High-level functionality . . . 59
6.2.2 Property 2: Vector Clocks . . . 66
6.2.3 Property 3: Riak Distribution . . . 69
6.3 Conclusion . . . 74

7. Related Work **76**
7.1 polyLarva plugins . . . 76
7.2 Elarva . . . 77

8. Conclusions **79**
8.1 Summary . . . 79
8.2 Future Work . . . 80
8.2.1 Overhead analysis . . . 81
8.2.2 Configuring synchronous and asynchronous monitoring . . . 81
8.2.3 Distributed central monitor . . . 82
8.3 Concluding Remarks . . . 82

References **83**

List of Figures

1.1 Runtime verification of a system by eliciting events 2

2.1 Actor communication 6
2.2 Monitoring using Erlang tracing 8
2.3 Compilation with parse transformation 9
2.4 Monitoring architecture in polyLarva [15] 12
2.5 Monitor interaction 13

6.1 Riak key-value stack 55
6.2 GET request hashing on three node cluster 57
6.3 Riak cluster communication to the central monitor 58

List of Tables

6.1 Used event constructs in Specification 6.1 62
6.2 Used system-side monitoring constructs in Specification 6.1 62
6.3 Used event constructs in Specification 6.3 65
6.4 Used system-side monitoring constructs in Specification 6.3 66
6.5 Used event constructs in Specification 6.4 68
6.6 Used system-side monitoring constructs in Specification 6.4 69
6.7 Used event constructs in Specification 6.5 73
6.8 Used system-side monitoring constructs in Specification 6.5 73
6.9 Usable event constructs in total (for the properties identified) . . . 74
6.10 Used system-side monitoring constructs in total 75

1. Introduction

Runtime verification [5, 14, 11] is a technique used to verify software correctness at runtime where correct behaviour means that the software behaves as it was intended to behave with respect to a specification. Runtime verification uses a formal specification to define a system's correct behaviour as a collection of properties, it is often much easier to define properties about a program's correct behaviour than it is to develop a program that behaves correctly. For example, there exist multiple sorting algorithms to efficiently sort a list of elements but it is much simpler to check if a list of elements is sorted from a programmer's point of view.

The traditional approach to identify software faults is to test the system. The system is put under test through a number of test-cases for a tractable number of cases and while this works reasonably well to detect most software faults it can never guarantee the absence of software faults. Another two techniques related to software correctness are formal proving and model checking. Formal proving uses formal methods in mathematics to formally prove or disprove correctness of algorithms used by a system however this approach usually requires manual work which requires expert knowledge and is often expensive and time-consuming. Whereas model checking seeks to exhaustively verify correctness over all execution paths in a system which is not a viable option for most industry-sized systems because verifying all execution paths in a system quickly becomes intractable. In contrast to these verification techniques, runtime verification is only concerned with the

correctness of a system's execution path that is chosen dynamically at runtime which can prove that a running system is respecting a given number of properties, assuming that the runtime monitor itself behaves correctly. The different approach used by runtime verification makes it a lightweight complementary technique to the mentioned traditional techniques; it is not intended to replace other techniques but used in order to give an additional protection and visibility of any incorrect behaviour.

The nature of runtime verification to verify correctness at runtime enables a runtime monitor to act upon the monitored system upon the detection of incorrect behaviour which is advantageous because it can take steps to rectify any resulting problems. For example, a payment transaction can be canceled if a problem is found.

Runtime verification needs to detect system behaviour to monitor its correctness. This may be done by instrumenting the source code of the monitored system to send system events at points in the program interesting to the runtime monitor. Figure 1.1 illustrates a simplified view of runtime verification through the use instrumentation.

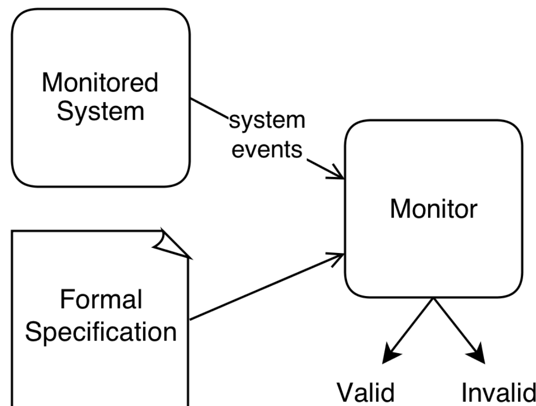

Figure 1.1: Runtime verification of a system by eliciting events

polyLarva [15] is an extensible runtime verification framework which can be extended to support multiple languages. The framework itself is technology-agnostic

while each language plugin provides the specific monitor generation capability for a particular language. The framework separates monitoring into a system-side monitor/s and a remote-side monitor where the system-side monitors detect system behaviour by eliciting events from the system and the remote-side monitor acts as a central monitor for all the system-side monitors. Monitoring can be executed on either monitor. Multiple system-side monitors may be used to provide monitoring across multiple components. The requirements for a system-side monitor to elicit events and possibly act on the monitored system requires polyLarva to provide language-specific support.

In this work, polyLarva is extended to support the Erlang[1] language. Erlang has an in-built support for concurrency using message-passing communication and it is a functional language; the language is used to build highly concurrent, distributed, scalable, fault-tolerant and soft real-time software systems.[4] Supporting the Erlang language provides interesting challenges because it embodies a completely different programming paradigm than the programming paradigms of currently supported languages (Java and C). The question of whether Erlang is supportable by polyLarva and how seamless is that integration together with the currently supported languages is answered in this work through the development of the Erlang plugin.

The Erlang plugin developed in this project is evaluated through a case-study, Riak. Riak[2] is an open-source distributed NoSQL database. It is an appropriate case-study in this context because it is used in the industry so we can evaluate the artifact on an industry-sized system and because Riak's distributed and fault-tolerant design also makes Erlang an ideal language for its implementation language. Riak's design and implementation are analyzed to define properties about it using an appropriation specification for polyLarva to define Erlang behaviour. Finally, Riak is monitored against these properties.

[1]Erlang website: http://www.erlang.org/ (last accessed May 2013)
[2]Riak website: http://docs.basho.com/riak/latest/ (last accessed May 2013)

1.1 Report Structure

The rest of this report is divided into the following chapters.

Chapter 2: Background Background information is written about a range of topics relevant to this work. These include Erlang, Aspect-oriented programming, Runtime verification, polyLarva and Elarva.

Chapter 3: Problem definition The aim and objectives of this project are described clearly and at length along with the descriptions of the challenges stemming from each objective.

Chapter 4: Design This includes the Design decisions involved in created the Erlang plugin.

Chapter 5: Implementation The implementation work involved to implement the Erlang plugin is described at length.

Chapter 6: Evaluation Riak is introduced and properties used to monitor Riak are described in detail.

Chapter 7: Related Work A discussion of work related to this project including the currently supported languages and Elarva, a runtime verification tool similar to the artifact created in this work.

Chapter 8: Conclusion The results of this work are summarized and possible future work.

2. Background

This chapter provides background information about the Erlang language; Aspect-oriented programming, to elicit events from monitoring; Runtime verification; poly-Larva and Elarva, a runtime verification tool for Erlang for asynchronous monitoring.

2.1 Erlang

Erlang [4] is a general-purpose garbage-collected language built for concurrency based on the actor model. The sequential aspect of the language is a dynamically-typed functional language. Erlang was built by Ericsson (hence the name Er-lang) to implement highly scalable concurrent telecoms systems.

2.1.1 Actor model

The actor model[1] is a concurrent model of computation. Actors can compute independently of other actors and do so concurrently. Actors can communicate with each other using message passing. Each actor has its own local memory and memory cannot be shared between processes. Communication must occur by sending data from one actor to another. Each actor has a mailbox where messages are received. Actors can use the mailbox to choose which message to read and process, and which to ignore if necessary. Since there is no shared memory, memory syn-

chronization problems to prevent race conditions are avoided. Figure 2.1 illustrates communication between two actors (or processes).

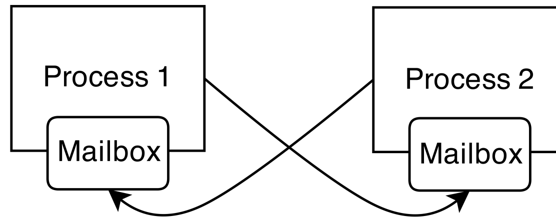

Figure 2.1: Actor communication

In Erlang, creating a new process is referred to as process **spawning**. When a process is spawned a Pid is returned. A Pid is used as an identifier when **sending** a message to a process. Messages **received** by a process are anonymous unless the sending process leaves its Pid in the message. A process may be assigned a name that may be used interchangeably with its Pid; this is called **registering**. **Process linking** sets up a two-way link between two process such that when a process dies, the other process receives a message for which the default operation is to kill the process. The Erlang operations for sending and receiving messages are primitive operators because Erlang was designed to be highly concurrent.

2.1.2 Basic Types

A brief description of types in Erlang is necessary to understand the main differences between Erlang and Java in discussions further in this work. The basic types in Erlang are:

Atoms An atom is simply a name representing a unique value. Similar in concept to *enums* in Java.

Integers and Float-point numbers These are normal numeral types except that they are unlimited in size; there is no maximum or minimum number.

Tuples A tuple holds fixed-size multiple values such as a pair of values or a record in a table.

Lists Variable-size multiple-valued type. A list can be divided in its first element (head) and the remainder of the list (tail).

Bit-string Useful for efficiently processing binary data.

Functions In functional languages, functions are a data type like any other. That means it may be used as input or output of another function.

Unique Identifiers This includes process Ids (Pid), socket ports and other referential values.

2.1.3 Variables

Variables in Erlang, as well as in any functional language, cannot update their value. A variable is either bound to value or unbound. Variables do not have to be declared as is required in other imperative-style languages. A variable's type is determined only once it binds to a value. Since Erlang is dynamically-typed, type errors are not caught at compile-time. This means that type errors within code from a specification script won't be be detected until the synthesized monitor executes that code.

2.1.4 Erlang tracing

Since Erlang runs on a virtual machine, the virtual machine can detect any events occurring at runtime; in Erlang this is called *tracing*. Tracing works by registering a process as a tracer process; a tracer process receives the events occurring in the system as messages. When event messages are received, the parts of the system causing the events continue execution; in other words the event messages are received asynchronously by the tracer process.

The tracer process can listen to events occurring in any process except for itself. It may also choose the kind of events that it wants to receive. The messages for *send*, *receive* and *call* events contain the Pid of the process which caused the event and any other data specific to the event. Send event messages detail the message being sent and to whom; receive messages detail the message received and call messages detail the module name, function name and arguments to the called function. Also, tracing can be switched on and off at runtime.

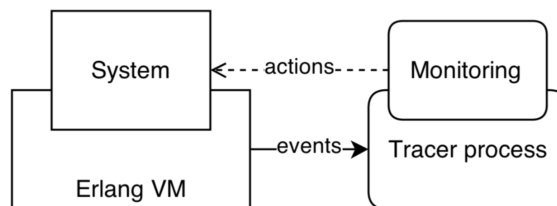

Figure 2.2: Monitoring using Erlang tracing

This tracing mechanism can be taken advantage of to extract monitoring events for asynchronous runtime monitoring in Erlang. Figure 2.2 illustrates a tracing process monitoring a system asynchronously.

2.1.5 Parse transformation

When compiling Erlang code, the parser will translate the source code to an intermediate form known as the *abstract format*. The compile function in Erlang allows the option of taking a function, called a *parse transform*, which takes an abstract form and returns a new abstract form. The parse transform is used to change the abstract form of a program under compilation which essentially changes the program.

Parse transformations are ideal for implementing AOP libraries in Erlang and all known current AOP libraries for Erlang use this technique.

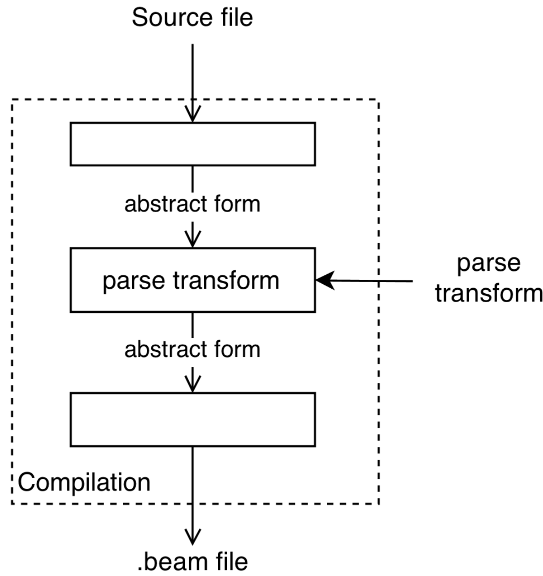

Figure 2.3: Compilation with parse transformation

2.2 Aspect-oriented programming

Aspect-oriented programming (AOP)[13] is a programming paradigm which aims to be complementary to other programming techniques. AOP is an attempt to modularize crosscutting concerns; for instance error handling can be seen as a crosscutting concern as it does not implement any code that is necessary for the primary function of a program and yet error handling code is usually scattered across several parts of a program. Separating concerns would further modularize code and remove dependencies which conforms with software engineering principles.

A concern can be seen as any aspect of program behaviour and cross-cutting concerns are concerns which interact with several other concerns. In other words, aspect-oriented programming can modularize cross-cutting concerns; in AOP, these modules are called *aspects*.

AOP works by defining identifiable points in program code (such as the start of a function call) and injecting code related to the cross-cutting concern which is called *advice* in AOP. These identifiable points in program code are known as join points or *pointcuts*. An aspect is therefore a number of advices which may be

injected at a number of pointcuts.

These aspects are injected in the program code by an aspect weaver which modifies the code according to the given aspects.

A frequent application of AOP is for the extraction of monitoring events for runtime verification. Pointcuts can serve to identify a monitorable event and the monitoring functionality can be injected as advices. Example 2.1 shows an example of an aspect, composed of a pointcut and advice, to monitor events just before the function call *user:login/2* is called.

Example 2.1: Aspect for monitoring a user logging into a system

```
Pointcut: before user:login/2
Advice: advice(Args) ->
          trigger_event(login, Args).
```

Code 2.2 shows the relevant source code of the *user* module with an implementation of the *login/2* function. Code 2.3 shows the code as it would be after the advice in Example 2.1 is applied.

Code 2.2: Source code before weaving aspect

```
-module(user).
-export([login/2]).
login(Username, Password) -> <login implementation>.
```

Code 2.3: Source code after weaving aspect

```
-module(user).
-export([login/2]).
login(Username, Password) ->
   trigger_event(login, [Username, Password]),
   <login implementation>.
```

2.3 Runtime Verification

Runtime verification takes a lightweight approach to verify software correctness by attempting to verify a system's correctness at runtime, and essentially only checking

correctness on the current execution path of a running system. The behaviour of a system is checked against a behaviour specification. This specification is used to produce a runtime monitor that checks if current system behaviour violates the correctness properties in the specification.

2.3.1 Intrusive vs non-intrusive monitoring

A runtime verification monitor needs to extract system events in order to check for system correctness. Extracting system events may be intrusive or non-intrusive. A non-intrusive way (or minimally intrusive) to extract events would be to listen on a communication channel between system components. Events would be based on messages communicated between components.

Intrusive monitoring extracts system events at the source code level of a program. Events would therefore be based on function calls and possibly other operations and data internal to the system becomes accessible to the monitor. This level of intrusion allows the runtime monitor to monitor the internal operation of software components which is not possible using non-intrusive monitoring. Intrusive monitoring can also force a system to perform actions such as restarting a system.

2.3.2 Synchronous and Asynchronous monitoring

Runtime monitors can either be synchronous or asynchronous. In synchronous monitoring, a process in the system that triggers an event has to stop execution while the monitor checks for correctness. This impacts system performance negatively but actions on the system will occur at the exact point in execution that caused a property violation. In asynchronous monitoring, the system continues execution while the monitor checks for correctness but it may be the case that upon a property violation an action taken on the system occurs too late such as stopping a payment transaction from completion.

2.4 polyLarva

polyLarva [15] is an extensible and configurable runtime verification framework. This framework is a novel approach to runtime verification that can support intrusive runtime verification across multiple languages. The framework is designed to be extensible to add support to unsupported languages. Correctness properties can define behaviour spanning across multiple components in component-based systems, possibly written using different languages.

polyLarva uses a specification script to define system behaviour. This specification script is used to synthesize system-side monitors and a central monitor[1] separately using separate polyLarva compilers as illustrated in Figure 2.4. The central monitor is technology-agnostic, it is synthesized in Java and is responsible for executing the main monitoring logic. It may run on a separate computing entity from the system being monitored, freeing computational resources which may be utilized by the system.

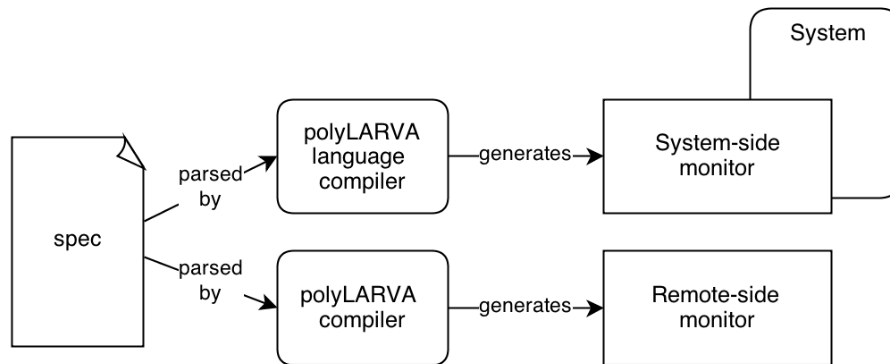

Figure 2.4: Monitoring architecture in polyLarva [15]

The system-side monitor is instrumented within the system and is responsible for eliciting system behaviour events and communicate them to the central monitor. Additionally, the system-monitor may verify correctness properties locally upon request from the central monitor. Multiple system-side monitors may be instrumented within the system to monitor separate system components.

[1]The terms *central monitor* and *remote-side monitor* will be used interchangeably for the remainder of this document depending on the context in which the terms are used.

Communication between the system-side monitor and the central monitor is synchronous so monitor actions on the system may occur immediately.

2.4.1 Monitoring Boundaries

polyLarva separates monitoring into system-side monitors and a central monitor, it is then left to the user writing the specification to configure where the verification of properties will occur. That is, either locally on the system side, on the same computing entity which is executing the system being monitored, or at the remote side which is the central monitor which may be running on a separate computing entity. An example of configuring monitoring boundaries may be found following this section in the Specification section.

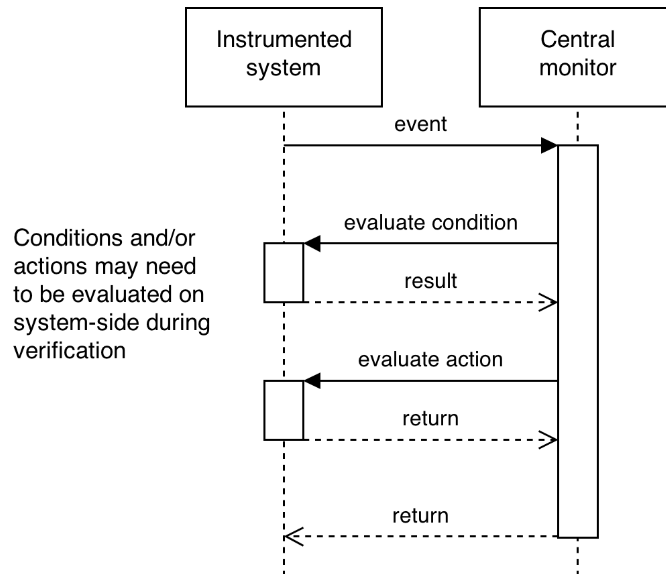

Figure 2.5: Monitor interaction

Verifying properties on the remote side monitor can move the monitoring overhead on a separate machine but it may also allow the definition of properties across multiple components by extracting events from multiple components and defining behaviour over the components as a whole.

However monitor verification at the remote-side has its limitations stemming from the intrusive monitoring approach that is normally used in polyLarva. Veri-

fication properties may need access to data in the system whose type is specific to a particular language or a system implementation. In case a type does not exist in Java, the language used for the central monitor, values are communicated as a unique id to the central monitor; therefore the central monitor would only be able to perform equality checks on such data.

polyLarva deals with these problems by allowing the user to configure monitoring boundaries through the specification script. By making correct use of these configurable monitoring boundaries, the user is provided with the necessary tools to express properties about the system over data types that are specific to the system.

2.4.2 Specification

The polyLarva specification is a script which defines system behaviour. Since the specification script is used to generate the central monitor, it is crucial for polyLarva that a common format is used to generate any system-side monitors even though they are language-specific.

polyLarva specification properties are defined using a list of guarded rules. Rules trigger actions if an event is triggered and a rule condition evaluates to true.

$$\mathit{event} \,|\, \mathit{condition} \mapsto \mathit{action}$$

Whenever an event is triggered by the instrumented system, the list of guarded rules is traversed to find a rule with a matching event. Subsequently an action may be executed depending on the result of the condition.

This guarded-rule approach is the basis of how properties are defined. The definition of events, conditions and actions are defined separately from the rules in order to keep the specification more organized and because these constructs are inevitably specific to a particular language. Therefore the rules are actually referring to names of events, conditions and actions specified elsewhere. The condition

defined in a rule may be a boolean expression of condition names where the boolean expressions allowed are: *not* (!), *and* (&&) and *or* (||). All these separate blocks of specification have to be present in a global context or a nested context. Therefore the following is a basic template of a specification script.

Specification 2.4: Basic polyLARVA specification template

```
global {
    events {
        event1 = { ... }
        event2 = { ... }
    }
    conditions {
        condition1 = { ... }
        condition2 = { ... }
    }
    actions {
        action1 = { ... }
        action2 = { ... }
    }
    rules {
        rule1 = event1 \\ condition1 -> action1;
        rule2 = event2 \\ condition2 -> action2;
    }
}
```

Three remaining blocks are still unmentioned; the *imports* block, the *timers* block and the *states* block. The imports block serves to import any classes or modules depending on the language. The timers block is used to declare timers which are useful to define properties with temporal constraints and will be explored later. The states block is where variables may be declared for the purpose of maintaining a local monitor state. Maintaining a state is necessary to define properties which need a to keep a state to express the property.

Example 2.4.1. A simple example that may warrant the use of a state would be in the scenario of a log-in system to unlock a mobile device where a property about this system is that a user cannot attempt to input a password after three consecutive failed tries. This will require the use of a counter to remember the

number of consecutive failed tries.

Specification 2.5: Basic login protection

```
global {
  states {
    int count;
  }
  events {
    systemStartup = { ... }
    inputPasswordAttempt(boolean succeeded) = { ... }
  }
  conditions {
    valid = { count <= 3; }
    success = { succeeded; }
  }
  actions {
    resetCount = { count=0; }
    incCount = { count++; }
    issueError = { ... }
  }
  rules {
    initialization = systemStartup -> resetCount;
    ifFailedIncrement = inputPasswordAttempt(succeeded) \\ !success && valid
      -> incCount;
    unlockAttempt = inputPasswordAttempt(succeeded) \\ success && valid
      -> resetCount;
    unlockAttempt = inputPasswordAttempt(succeeded) \\ !valid -> issueError;
  }
}
```

Events

Event declarations are defined as method names or function names depending on the language. It is up to the language compiler's plugin designer to decide what these method names mean. In the Java plugin the method name refers to a method call of methods of the same name as follows.

```
event = { target.methodName() }
```

Events may also specify variables and the polyLARVA compiler parses for them and their types. If the types are not specified they are ignored by the remote-

side monitor's compiler and simply used as wildcards for the associated parameter. Event variables are accessible by the conditions and actions specifications. Event variables are made accessible by listing them in parenthesis after the event name.

A event variation may be specified which varies where the event actually occurs. For instance the default variation is that the event is triggered upon the function call however the $uponReturning(returnedVariable)$ variation changes the event to trigger when the function call returns and also makes accessible the returned variable.

Finally, events may have a where clause where assignment operations may be specified which are executed when the event is triggered. Specification 2.6 uses the where clause to access the name of a user object instance. This can be useful when event variables are used from the central monitor as in this case the type User is system-specific so the name cannot be accessed from the central monitor.

Specification 2.6: Retrieving a user's name as the user logs in

```
event(String name) = { loggedUsers.add(User user) }
   where { name = user.getName(); }
```

Configuring Monitoring Boundaries

As explained above, polyLARVA requires the user writing the specification script to specify monitoring boundaries. The blocks which have to specify the monitoring boundary are the states block, the conditions and actions blocks; as can be observed in Specification 2.5, these blocks need to contain code and the language used to write that code changes according to the monitoring boundary. Monitor-side code is always written in Java because the central monitor is synthesized in Java code. System-side code is written in the language used in the part of the system that is to be monitored. There is no default monitoring boundary when writing the specification, the monitoring boundaries have to be configured explicitly; therefore note that the states, conditions and actions blocks in Specification 2.5 are incomplete.

The monitorSide and systemSide blocks are used to configure the monitoring

boundary. These blocks go within the states, conditions and actions blocks and relevant specifications go inside the monitorSide or systemSide blocks as exemplified in Specification 2.7.

Specification 2.7: Configuring monitoring boundaries

```
...
states {
   monitorSide{
      int var;
      ...
   }
}
conditions {
   monitorSide { ... }
   systemSide { ... }
}
...
```

Multiple component support

The system to be monitored may have several components and it may be necessary to separate the specification script for each component. This may be because that they are written in different languages or simply because some specification applies only to a particular component.

Separating component specification is achieved by labeling event names and systemSide blocks using the @ symbol as exemplified in Specification 2.8.

Specification 2.8: Separation component specification using labels

```
...
events {
   event@erlangSide = { ... }
}
actions {
   monitorSide { ... }
   systemSide@javaSide { ... }
   systemSide@erlangSide { ... }
}
...
```

Nesting Monitor Contexts

In Specification 2.5, the property was defined in a global context. Nesting contexts within the global context can be used to create separate monitor instances with a more specific concern. It is possible to write a rule in the rules block that triggers new instances of a monitor context within that rule. A nested monitor context would have its own properties to monitor and a separate scope for variables.

This construct is useful to write properties about a particular concept according to the language being monitored. For example, in Java, you could write a rule to dynamically create a new monitor upon the creation of a new object and the new monitor would be able to monitor properties about that monitor. In Erlang, this construct can be used to monitor properties about particular processes instead of objects.

As monitor instances are replicated for each of such concepts, each monitor instance has a context variable to define which instance it is representing. In addition, event declarations within nested contexts should specify which instance events are bound to in the where clause of the event declaration [15].

In the specification, nested monitors are specified with the *upon/load* construct as exemplified in Specification 2.9. Monitoring contexts can be nested within each other indefinitely.

Specification 2.9: Nesting monitors

```
...
rules {
  ...
  upon {
    rule(User user) = ...
  } load (user) {
    events {
      event(User u) = { ... } where { user = u; }
      ...
    }
    ...
  }
}
```

```
...
```

Monitoring directives

Monitoring directives are a set of predefined commands that may be issued to the central monitor. There commands may include dynamically switching rules on and off, timer-related directives and triggering internal events. Monitoring directives start with *larva* : such as larva:switchOff (to dynamically set whether a specified rule will be used or not) and may be used as actions in rule definitions.

Specification 2.10 uses the switchOff directive to exclude a rule depending on which rule is triggered first.

Specification 2.10: Monitoring Directives

```
...
ruleA = { event1() -> larva:switchOff(ruleB), action1 }
ruleB = { event2() -> larva:switchOff(ruleA), action2 }
...
```

Temporal properties and Internal Monitor Events

A timers block exists to declare timers which are useful to define temporal constraints which cannot be defined otherwise. Timer events are an example of an event that occurs internally within the monitor without it being triggered from system monitored. Internal events can also be triggered through an action. Internal events can be used to trigger events across nested monitors and optionally communicate variables across. Internal events are defined by a question mark (?) followed by an event name.

Specification 2.11: Internal events

```
...
event = { ?internalEventName() }
...
rule = { someEvent() -> larva:fire(internalEventName()) }
...
```

2.5 Elarva

Elarva is an asynchronous runtime verification tool for Erlang. Elarva elicits events from the system by the Erlang tracing mechanism of the Erlang VM. As explained in the Erlang section, Erlang tracing occurs asynchronously and so Elarva can only provide asynchronous runtime monitoring.

The specification properties are written using Dynamic Automata with Timers and Events (DATEs). Unlike polyLarva the specification is written specifically for Erlang and besides function calls has concepts for process spawning, linking and registering as well as concepts for inter-process communication, sending and receiving data. These concepts can be represented just as well using function calls, such as *erlang:spawn/1*, except for the primitive operators for sending and receiving data.

2.6 Conclusion

The subjects described above provide the required information to understand the remainder of this document. The next chapter defines explicitly the problem of this thesis.

3. Problem Definition

An absolute requirement for runtime verification is that the entity supervising a system with the task of verifying behaviour is to be aware of that system's behaviour. In polyLarva, that awareness is solicited from the system by instrumenting the system at compile-time. That is, the process of intrusively injecting actions within the system at the source-code level such that the system's actions are dynamically made available to an external monitor. These injections are responsible for triggering the events that reveal the system's behaviour. As the instrumentation occurs at the source-code level, the events generated reveal language-specific constructs such as function calls and data types.

Meanwhile, a central monitor synthesized in Java must be able to monitor the system's behaviour but the system's language-specific constructs may be completely foreign to Java. Since the central monitor is synthesized in a different language from that of the system being monitored, the central monitor is not able to process data with types which do not exist in Java. Thus, in polyLarva such processing must be processed on a system-side monitor. For these reasons, polyLarva requires a language specific plugin for any language to be supported. This plugin will be responsible for synthesizing the system-side monitor.

This problem will be explored from Erlang's perspective. Being actor-based and a functional language, Erlang uses a different programming paradigm than C or Java, which are the only languages currently supported by polyLarva. Extending

polyLARVA for Erlang will therefore introduce fresh challenges which are discussed below.

A support for Erlang will enable synchronous or asynchronous runtime verification for Erlang through polyLARVA. Also, due to its technology-agnostic nature, polyLARVA is able to monitor systems developed in multiple languages. Supporting Erlang will enable the monitoring of correctness properties across components of which only a subset are written in Erlang and the remaining components may be written in other languages.

3.1 Aim and Objectives

The aim of this work is to extend polyLARVA to support the monitoring of systems or part thereof written in Erlang. This will result in a plugin which will allow polyLARVA to synthesize a system-side monitor in Erlang which mainly must be able to extract system events, communicate them to a remote monitor and execute any local monitoring code[9].

Eliciting events This requirement has two challenges: defining Erlang events in the specification script and extracting them. Since a requirement of polyLARVA is that multiple language-specific components have to be generated from a single specification script, the limitations of the specification script allowed by polyLARVA needs to be explored in order to understand how flexibly the events can be defined in order to design how events will be specified for Erlang in a user-friendly manner. For instance, besides monitoring function calls it would be preferable to also monitor some primitive operators related to inter-process communication in Erlang. Also, it would be interesting to know if it is possible to define event variables without declaring their type.

The second challenge relates to how the events will be extracted from the program. The currently implemented language-specific plugins use aspect-oriented programming (AOP) to extract events. The options available are to

either use an AOP library in Erlang or to use Erlang's tracing mechanism. The downside of using Erlang's tracing mechanism is that the monitoring will be asynchronous because polyLarva is intended to provide synchronous monitoring and Elarva already provides asynchronous monitoring for Erlang. Therefore, it is preferred to use AOP although known AOP libraries for Erlang are not particularly mature in functionality.

Communication polyLarva monitors communicate through standard TCP-based sockets. This places the requirement that TCP sockets are implemented for a language to be supported by polyLarva. Two-way TCP communication has to be supported because the central monitor will send replies to event messages through the same socket. Communication may need to occur concurrently for parallel systems. The plugin will need to communicate events through an agreed format along with any event variables necessary and wait for a response before proceeding; it may need to execute system-side conditions or actions depending on the messages received by the remote monitor.

System-side monitoring The system-side monitor will need to be able to maintain a state to store monitor variables on the system side. It must also be able to execute system-side conditions and actions upon request from the central monitor. The state, conditions and actions required will be specified in the specification script and the plugin will use that information to synthesize the system-side monitor.

In polyLarva's specification script, the general work-flow when using state variables is to declare the type variables in the states block and then the variables' values are used or updated in other sections of the script. This approach is natural for languages like C and Java but it is strange in the functional paradigm. In Erlang, variables are never declared and variables do not get updated once they are they are bound to a value.

The implemented plugin for Java maintains the state using global variables

and the conditions and actions are implemented as methods in a class for each monitoring context. However, Erlang does not have the concept of global variables because it does not match the functional paradigm so a different technique is needed to implement the local monitoring.

4. Design

This chapter will start with an overview of the control flow of the monitoring process mainly from the perspective of the system-side monitor. The key decisions in the design stage involved defining what Erlang events need to be monitored and how the specification is written in a way that is appropriate to the Erlang language within the restrictions that are imposed by the polyLARVA parser. Two approaches are identified to elicit events from an Erlang program which will be explored in detail.

4.1 Monitoring control flow

The Erlang system-side monitor forms part of the monitoring system or the parts of the system written Erlang. An initial message is sent by the system-side monitor to the central monitor to start monitoring. All communication between the monitors is synchronous, that is whenever the system-side monitor sends a message to the central monitor, a reply is always expected by the central monitor. After the initial message, the monitoring control flow will typically follow the following the steps.

1. Monitored system enters a part of the system that is being monitored
2. Monitored system starts executing monitoring code
3. A message is sent to the central monitor containing information about the

event triggered and relevant data about the event

4. Central monitor receives event message and starts monitoring logic

5. System-side monitor receives a reply from the central monitor either requesting local monitoring where a reply is sent back the central monitor to continue monitoring logic or a message indicating that the central monitor finished checking for correctness.

Internal events from the central monitor such as from a timer event may cause the central monitor to initiate communication with the system-side monitor to execute local monitoring code either computing the result of a system-side condition or executing a system-side action. The system-side monitor must be listening for messages from the central monitor while the monitored system is executing normally. The system-side monitor must also provide the functionality to log its actions in a file documenting any triggered events, any executed system-side conditions or actions, and any messages sent between the monitors.

4.2 Approaches to elicit events

In order to monitor behaviour within a system, it is not enough to monitor the external functionality of a system because we may be interested in the internal behaviour of a software component. A system's source code reveals the required information about a system's internal functionality and so monitoring events will be based on a system's source code.

In Erlang, events may be elicited using Erlang's tracing mechanism or more generically we can use the aspect-oriented programming (AOP) approach.

4.2.1 Using Erlang tracing

Erlang's tracing mechanism involves setting up a tracer process running independently that listens to events asynchronously from the Erlang virtual machine (VM)

about a monitored system. The Erlang VM would send events whenever the monitored system performs an action from a set of actions selected at startup, containing all the required information about the event. The tracer would be responsible for communicating the events to a central monitor and performing monitoring locally. Since the events are sent asynchronously the monitored system and the monitoring would be running in parallel.

Erlang tracing has the advantage that it is already built-in in the Erlang VM and asynchronous monitoring does not force the system to wait while the monitor is checking for correctness. However this means that upon detecting incorrect behaviour the monitor would not be able to act upon the system at the exact point that in the system's execution that incorrect behaviour was detected.

4.2.2 Using AOP

A generic approach across monitoring tools to elicit monitoring events is to use AOP. This involves taking the original source code and modifying such key points in the program it would do monitoring computation with data taken from the program essentially creating events to the system-side monitor. AOP uses pointcuts to identify the points in the program where advices can be injected. In this context, the pointcuts identify points in the program which the system-side monitor needs to be notified of, that is the system events. The advices will be Erlang source code that will be responsible for the system-side monitoring. This involves sending a message to the central monitor, waiting for a reply and acting accordingly.

By forcing a program to compute advices, it is possible to force the program to do anything. This means that unlike Erlang tracing, AOP can be used to force the monitored system, or at least the process that calls the advice, to stop execution while the system-side monitor communicates with the central monitor and performs other local monitoring.

4.2.3 Conclusion

polyLARVA was developed to provide synchronous monitoring; the currently implemented language-dependent plugins for Java and C offer synchronous monitoring. The central monitor always sends a reply to the system-side monitor to instruct it to allow the system to continue execution. Despite this, it is perfectly possible to implement asynchronous monitoring for the Erlang plugin by using a process running separately from the monitored system to wait for replies from the central monitor so both monitoring techniques were considered.

There are advantages and disadvantages to both techniques. Asynchronous monitoring does not interrupt the monitored system or at least the interruption is minimal to communicate the events however asynchronous monitoring is not able to act on the system immediately upon detection of incorrect behaviour. The argument for synchronous monitoring is vice-versa. Synchronous monitoring was the preferred technique for the Erlang plugin because polyLARVA was intended to be synchronous, and there already exists an asynchronous monitoring tool for Erlang called ELARVA.

4.3 Erlang events

Events are elicited by forcing a monitored system to run monitoring code at key points in a program such as just before a function call; events should also communicate information from the system related to the event such as the arguments to a function call. When describing system behaviour, the property writer needs to have a solid understanding of the source code to be monitored because correctness properties depend on events and events will depend on the source code. The user who writes correctness properties needs to identify points in the source code from which some meaning of the system's behavior can be derived upon their execution. It is interesting to identify which events are most descriptive of the behaviour of an Erlang program. A requirement in general would be to monitor function calls

by identifying the function by its name. In an object-oriented language it may be interesting to know which object called a method or when an object is instantiated however Erlang does not have objects. The actor model in Erlang encourages processes to perform separate roles and it may be interesting to allow monitoring of events related to processes and inter-process communication to define program behaviour related to processes. The following are Erlang actions related to processes that we would like to monitor.

spawn Process spawning creates new processes. This requirement may be useful to write properties about a single process, a monitor may be started upon detecting a process spawn.

link/unlink Process linking is the use of two-way links to trigger messages upon process termination.

register/unregister Register names to processes such that a process name may be used interchangeably with its Pid.

send/receive These are primitive operators in Erlang for message passing among processes.

With the exception of send and receive operators, this functionality is implemented as functions in a standard library and so, from the context of monitoring, that functionality can be monitored in the same manner as other function calls. The send and receive operators however require special consideration in order to be defined in a specification script.

Therefore, the identified events to describe Erlang program behaviour can be categorized into monitoring functions and process communication. These two categories can be expanded as follows.

- functions
    - function execution

  - function call

- process communication
  - sending messages
  - receiving messages

Since the chosen approach to elicit events is using AOP, a distinction must be made between function execution and a function call. Function execution is when a function actually executes while a function call is when the function is called. In source code, the code actually implementing the function is executed during function execution while a function call is the code calling the function. Since AOP works by modifying the source code and assuming we do not have access to the standard library function implementations, then we can only alter the source code of the given program to be monitored. Therefore, the function implementation of standard libraries cannot be altered to send events however it is possible to alter the source code where the program calls a function. An alternative will be provided for the user to create events at the function's implementation whenever that source code is available.

For each of the above listed actions, there are at least two events that may be monitored, before and after the action. Function events just before the function is called or executed, would contain the arguments given to that function while an event after the function is called/executed contains the arguments and the value returned. Sending an event would contain the message sent and the Pid of the process receiving the message. Receiving a message has three events that may be monitored. Before it starts waiting for a message, upon the reception of a message before it processes the message containing the message received, and after the message is processed.

4.4 Specification script

The design of the specification required to describe Erlang behaviour is described in this section.

The structure of the polyLARVA specification script consists of some blocks that are language dependent which affect the synthesis of the system-side monitor while other blocks are mainly generic regardless of the language of the monitored system and serve to synthesis the central monitor. Language dependent blocks in the specification include the events block and any block which separates its contents within system-side and monitor-side monitor-side blocks where the system-side blocks are only for the system-side monitor whereas the monitor-side blocks are only used by the monitor-side monitor. In this section, the focus will be on the language dependent parts of the specification script with respect to Erlang.

4.4.1 System-side monitoring state

The specification script defines a number of context monitors where each monitor contains its own ruled-based properties. Each monitor may contain its own separate monitoring state where variables may have the same name as long as they belong to different contexts. The monitoring state of each monitor is normally defined by declaring variables within the states block or the events block, such that they may be used and/or updated within the conditions and actions block.

Example 4.4.1. A simple example would be to maintain a counter such that it is checked and updated every time a particular event is triggered.

Specification 4.1: Using a counter using an imperative language

```
...
events { event1 = ... }
states {
  systemSide {
    int counter {
      restoreWith { counter = 0; }
```

```
    }
  }
}
conditions {
  systemSide {
    cond1 = { counter < 5; }
  }
}
actions {
  systemSide {
    action1 = { counter++; }
  }
}
...
```

Unlike Erlang variables, the variables in Specification 4.1 are mutable. Variables in Erlang do not change value, they are bound to a value and stay bound. Therefore, an operation like `counter++` or `counter=counter+1` would return an error. Additionally, a variable in Erlang does not need to be declared.

To solve the mutability problem, variable names in the specification script will actually be keys in a key-value map so that values may be updated. Variable names should therefore be used as atoms by the user. A pair of functions will therefore be necessary to access and update the values associated with the variable names, which will be referred to as *var:get* and *var:put* respectively. Variables can be initialized in the restoreWith block using the *var:put* function. Declaring a variable type in the states block does not make sense here since Erlang does not type check at compile-time so type is not declared but the word *var* is used instead. The polyLarva language parser requires a type to be specified before the variable name so the word *var* is used as a placeholder.

Specification 4.2 shows the specification for the above example using the approach just described.

Specification 4.2: Using a counter with a mapping approach

```
...
events { event1 = ... }
states {
```

```
    systemSide {
        var counter {
            restoreWith { var:put(counter, 0) }
        }
    }
}
conditions {
    systemSide {
        cond1 = { var:get(counter) < 5; }
    }
}
actions {
    systemSide {
        action1 = { var:put(counter, var:get(counter)+1) }
    }
}
...
```

The specification may define nested monitors within the global context monitor and it is a requirement that the scope of the variable names is limited to its context. In order to satisfy the mapping functions can take an additional integer to denote the context Id; this additional value will be added automatically by parsing the Erlang code to add the additional argument. It would be ideal if only the variables names, or keys in the key-value map, declared in the states block or events block could be used in the remainder of a context and so the the code will be parsed by the Erlang plugin to check for this as well.

Alternative approach to updating variables

An alternative approach is identified to explore whether the monitoring state can be maintained using a cleaner, simpler approach.

The use of the *var:put* function can be completely hidden from the user using an alternative approach to updating the variables. All expressions in Erlang return a value, for instance the assignment operation returns the value assigned and a sequence of expressions returns the value of the last expression in the sequence. The result of the Erlang expression written in the sections of the specification that

can update the monitoring state would be taken to modify the state. In order to update the state, the result of an action would have to be in either of the following types (If the result does not match the following types it may simply be ignored):

- A two-valued tuple where the first element is the variable name (atom) and the second element is the updated value
- A list of the above

Since the restoreWith block are already associated with a variable; the result of the expression can be taken as the initialized value of the variable instead of expecting a tuple. Specification 4.3 implements the specification for the example shown above.

Specification 4.3: Alternative approach to update state

```
...
events { event1 = ... }
states {
   systemSide {
      var counter {
         restoreWith { 0 }
      }
   }
}
conditions {
   systemSide {
      cond1 = { var:get(counter) < 5; }
   }
}
actions {
   systemSide {
      action1 = { {counter, var:get(counter)+1} }
   }
}
...
```

While this approach arguably looks cleaner, it does not give any advantages over simply using the *var:put* function and moreover using *var:put* would arguably be easier to learn because it is more consistent how the monitoring state is updated

in other polyLarva plugins, which is the imperative approach. Additionally, implementing this approach would require yet additional implementation time which cannot be afforded. Therefore, since this alternative approach is entirely for cosmetic purposes, it is abandoned in favor of the simpler approach using the *var:put* function.

4.4.2 Events block

The events blocks is where monitoring events are specified in the script such that the specified events are elicited from an Erlang program. An event specifies the pointcut within the program where the event occurs, this may include multiple points within the program, and it may also declare variables to extract information from the program, such as function arguments. Variables declared within the events block form part of the local monitoring state and may be sent over to the central monitor such that they may form part of the monitoring state of the central monitor.

Event variables

The polyLarva language parser accepts event variables which are declared with or without a type; the difference is that variables declared without a type are ignored by the central monitor. In the Java and C plugins, a variable in an argument list of a function event simply serves as a placeholder for the function argument at its position. A variable declared with a type can be used from the monitor-side conditions and actions depending on the type. If the type is a primitive Java type it can be used as a variable of that type in the monitor-side conditions and actions; otherwise the central monitor uses it as a String variable holding a unique identifier which can be used only for equality comparisons. If an event variable does not declare a type, it cannot be used in the monitor-side conditions and actions as it is ignored. These observations influenced how event variables are specified for Erlang.

Since the Erlang variables names do not need to declare a type, declaring event

variables without a type is sufficient if the event variables do not need to be used from the monitor-side. If the variable needs to be used on the monitor-side a type is declared to make the central monitor compiler create a central monitor that can use the event variable. If the type of the expected value of the event variable has a Java primitive type equivalent, the Java type is used otherwise the word *var* can be as a placeholder so that the event variable is stored as a String variable holding a unique identifier of the Erlang value.

Event variables used from the system-side are used normally as other variables declared in the states block. That is, as atoms associated to values in a key-value map using the *var:put* and *var:get* functions.

Event definition

The specification needs to be able to specify the events identified above, that is function events and process communication events. One way to specify these events would be to use the following format for functions.

```
<module>:<function>(<argument list>)
```

where `<module>` and `<function>` are replaced by the module and function name respectively of the function to be monitored; `<argument list>` is a comma-separated list of variable names, the asterisk (*) may be used as a last argument to denote zero or more arguments. The communication events could be specified as follows.

send `pid ! message`

receive `receive`

The *send* event is denoted by the exclamation mark and *pid* and *message* are variables. The receive operator is denoted by the *receive* word, the default receive pointcut can be considered as the pointcut where the program starts waiting to receive a message so there are no variables here as the message has not yet been received. Send and receive operators are fairly generic operators so it could be

useful to restrict these events to operations within a given function which could be specified as follows.

send `pid ! message within <module>:<function>/<arity>`

receive `receive within <module>:<function>/<arity>`

Defining events this way would be the most intuitive but unfortunately the polyLARVA parser only accepts event definitions of the following format.

```
<target>.<method>(<argument list>)
```

where `<target>` is a variable, `<method>` is a method name and `<argument list>` is list of arguments of arguments in the same way as explained above. Therefore, the events definitions have to be written differently to conform to the parser's restrictions.

function call `<module>.<function>(<argument list>)`

send `Message.send(pid, message, <module>, <function>, <arity>)`

receive `Message.receive(<module>, <function>, <arity>)`

The function definition above refers by default to a function call; to specify an event for when the function executes, the word `execution` is written at the beginning of the definition before the module name. The last three arguments in the send and receive event definitions denote the function which contains the send or receive operation respectively. The default pointcut of the event definitions is just before the event occurs, to define an event just after the event occurs, the `uponReturning(<return variable>)` event variation can be used where `<return variable>` is a variable containing the result of the function. The uponThrowing(error) variation handles cases where a function throws an error. The use of these event variations for function events is standard for all other plugins. The uponHandling(message) event variation specifies the event where the receive operation receives a message just before the message is processed by the monitored system and the message variable contains the message received.

Where clause

The where clause contains code that is processed after an event is triggered. The polyLARVA parser requires the specification within the where clause to be an assignment operation in the following format.

```
variable = <erlang expression>;
```

Multiple such operations may be written in sequence within the where clause.

4.5 Conclusion

This chapter identifies aspect-oriented programming as the method used to elicit events and describes the alternatives considered. The Erlang events necessary to detect Erlang behaviour and the specification used to describe Erlang behaviour were identified. The next chapter describes the implementation of the Erlang plugin according to the design described in this chapter.

5. Implementation

This chapter includes implementation details of the Erlang plugin to support the requirements discussed in the Design chapter. The AOP library required for this work needed to be implemented using other existing implementations which only partially supported the necessary requirements to elicit events for the events identified in the Design chapter.

5.1 AOP Library

As discussed in the Design chapter, the chosen method for eliciting events is aspect-oriented programming (AOP). An AOP library implementation written by Alexei Krasnopolski[1] was discovered which can inject advices on the following pointcuts.

- before a function executes
- after a function executes
- when function execution throws an error

Note that since AOP works by changing the source code, the pointcuts described work only if the source code of the function is provided. Krasnopolski's plugin works by renaming the function which fits a given pointcut and uses the function

[1]Website for Krasnopolski's AOP library: http://sourceforge.net/projects/erlaop/ (last accessed May 2013)

original name to create a new function which calls the original renamed function and adds advices before and/or after the original function is called; essentially it replaces an implemented function with a function that includes advices.

This plugin does not have pointcuts for function calls, so it cannot be used to monitor functions for which the source code is not provided such as functions in the standard library such as the function used to spawn new processes. Also, it does not have pointcuts for the *send* and *receive* operators used for process communication. An AOP plugin was implemented by Clifford Galea Valletta[2] around the same time of this thesis. His plugin supports exactly the pointcuts which are missing from Krasnopolski's plugin, that is function calls and the send and receive operators however it does not support the pointcuts in Krasnopolski's plugin.

The input required to provide the aspects for the two implementations are completely different so the two plugins were combined to create a new plugin that supports all the required aspects using a common way to specify them. The method used to specify the aspects for both plugins will be briefly explained in order to explain the work involved to combine the two.

Krasnopolski's library The advices are implemented in an Erlang source file/s and the aspects are specified in a separate file/s. The aspect file contains a list of aspects where each aspect was defined as an advice related to a list of pointcuts. An advice is specified by module and function, and includes whether the pointcut is before or after; the advices are expected to have a specific arity to take the required parameters including function arguments for instance. A pointcut is specified using regular expressions for the module and function, and by arity and scope. Function scope includes public, local (private) or both.

Def. 5.1: Definition file example for Krasnopolski's plugin

```
[Aspect(
    Advice(before, advices_module, before_advice),
```

[2] I helped Clifford by testing his plugin, discovered and fixed some bugs.

```
    [ Pointcut("module", "function", "*", public) ]
),
Aspect(
    Advice(after_return, advices_module, after_advice),
    [ Pointcut("module2", "function2", "1", public) ]
)].
```

Clifford Galea's library Advices are also implemented in a separate Erlang source file however the module and functions have to be called a specific name to be recognized by the implementation. The arity of the advices is also different from that of the other implementation. Another file/s is used to specify the pointcuts; the advice functions are not specified since they are assumed to be implemented in predefined function names. Pointcuts specify a module, function and arity and whether is a function call, send or receive operation. If it is a function call, the module, function and arity specify the function call; otherwise they specify the function containing the send or receive operation.

Def. 5.2: Definiton file example for Clifford Valletta's plugin

```
[Pointcut(call, "_", "spawn","*"),
 Pointcut('receive', "module", "receiving_function","*")].
```

Def. 5.2 shows an example of a definition file containing two pointcuts. The first pointcut is specific to function calls to functions named *spawn* with any arity or module implementing the function. The second pointcut is specific to the receive operator used within any function called `receiving_function` inside a module called `module`.

Pointcuts automatically include both before and after and it is up to the advice function implemented by the user to discriminate between the two. For instance, if the pointcut of a function is only required before the function, the advice function would check a parameter to the function and implement the advice for when it is *before* and do nothing when it is *after*.

5.1.1 Combining the two implementations

The combined AOP library uses the same method as the latter described implementation to specify aspects. The difference is that a new pointcut type is added to specify function pointcuts when the function executes. The code used to read the aspect files in Krasnopolski's implementation was re-written to read pointcuts from the new format and look for the new pointcut type while Clifford Galea's implementation had to exclude the new type.

Since the expected advices for Krasnopolski's implementation are different from the advices actually given, new functions are injected into the source files implementing the advices expected by the implementation which calls the actual advices. When reading the aspect files, the advices for the execution pointcuts point to the injected functions. After the aspect files are read as the expected internal format, Krasnopolski's parse transform is used to modify the abstract form of the source file and then the parse transform of the other implementation is used on the transformed abstract form.

5.2 Erlang plugin

polyLARVA is implemented in Java and the language compiler API provides an abstract class called *MonitorFileWriter*[15]; any language plugin needs to extend this class and implement all of its abstract methods. A simple addition is added to the class LarvaCompilerFactory to include code to create an object of Erlang's extension of the *MonitorFileWriter* class whenever the language flag *erlang* is passed to the language compiler's JAR file.

The aim of the plugin is to generate the source code of the system-side monitor and to generate the AOP code to elicit events from the system. Some of the generates code depends on the given specification script while other code is the same regardless of the specification script which exists to provide the necessary functionality for the system-side monitor to work. The AOP code depends entirely on the

events blocks in the specification script. Parsing of the specification script is carried out by the polyLarva parser and the intermediate object-oriented representation is used by the Erlang plugin.

The requirements to implement the *MonitorFileWriter* class, as identified from polyLarva's source code, include the following.

- create folders which will contain the generated files.
- create configuration files for communication (port numbers, etc) and logging (logging levels, etc) details which can be edited by the user.
- create communication code including starting new connections with the central monitor, sending and waiting for messages, processing messages, and starting a process to listen for unexpected messages from the central monitor.
- create source files for each monitoring context in the specification script implementing code related to local variables, conditions and actions.
- create the necessary AOP code to elicit events from the monitored system.
- create an implementation to generate unique IDs and any other language specific code necessary for the system-side monitor such as the mapping functions *var:get* and *var:put* to map variable names with values discussed in the Design chapter.

5.2.1 Generating AOP Code

The AOP library requires one or more files specifying the pointcuts and one Erlang module called *advices* that implements the advices. The task of the advices is to send a message to the central monitor specifying the event triggered accompanied by any variables related to the event, and wait for a reply from the central monitor essentially shifting control to the monitoring code until the monitoring is complete.

The event message to the central monitor is a String containing IDs to indicate that it is an event message, and the event ID of the triggered event. Additionally it requires a list of variables associated with the event which include the context variable (a variable associated with a nested context monitor), event parameters and where clause variables. Any variables that are not declared with a type are not sent since they are not expected by the central monitor. Variables declared with a Java primitive type cause the Erlang value to be converted to a value which is readable for the Java central monitor before it is sent. Any variables declared with a type that is not a Java primitive type, causes the value to be converted to a unique Id, which is a unique identifier for any Erlang value, before it is sent.

Specification 5.3 shows a simple event definition for a function computing the length of a list. This should generate one advices source file for all the events in the script and one aspect file for the global context which contains only one event.

Specification 5.3: Event definition

```
global {
  events {
    testEvent(list, return) = { execution testModule.length(list)
      uponReturning(return) }
  }
}
```

The advice generated falls under the *after_advice* function because the event definition specified the event variation *uponReturning* which causes the event to be triggered after the event is executed. A *case...of* expression is generated for each event definition because a pointcut may have multiple advices in the case that more than one event clause defines an event which includes the same pointcut. The variables *V1list* and *V1result* are named so because Erlang variables have to start with an uppercase letter (or an underscore)[4]; the number denotes the event Id to prevent an Erlang compiler error which complains about the same variable name being used more than once (in the case that two variables from different events use the same variable name) even though their use in separate *case...of*

expressions wouldn't technically interfere with each other. The empty list passed to *tcp:connectSend/3* is a list of values associated to event variables but since the central monitor ignores variables without a type declaration the values are not sent.

Code 5.4: Generated advice

```
after_advice(Type, _, Module, Function, Args, R) ->
    Tuple = {Type, Module, Function, Args, R},
    case Tuple of
        {exec, 'testModule', 'length', [V1list], V1return} ->
            var:put(0, 'list', V1list),
            var:put(0, 'return', V1return),
            log4erl:info("Sending event notification: _testEvent0"),
            tcp:connectSend('ASPECT_EVENT', 1, []);
        _ -> ok
    end,
    ok.
```

The generated pointcut is of type *exec* because the event definition uses the *execution* keyword. This pointcut definition technically injects advices both before and after but the *before_advice* function will ignore this pointcut and do nothing while the after_advice is implemented to trigger the event.

Code 5.5: Generated aspect file

```
[Pointcut(exec, "testModule", "length", "1")].
```

5.2.2 Generating Monitor context code

An Erlang source file is generated for each monitoring context in the specification script. The generated code is created from the system-side blocks within the imports block (which may contain module attributes such as an import attribute to import source files), the states block, the conditions block and actions block.

The code generator needs to check that any variable name used with the mapping functions *var:get* and *var:put* are declared either as a context parameter, or an event parameter or within the states block. The variable names used are extracted

from the conditions and actions blocks by using regular expressions to match calls to *var:get* and *var:put* and removing the code before and after the variable name.

The code generator has to create functions to initialize variables declared in the states block with a restoreWith block; to implement the conditions and the actions. This involves dumping the code in the specification script within the function as well as some logging calls for the conditions and actions. However, before dumping the code found in the script, the calls to the *var:get* and *var:put* have to be modified to add the context Id as an extra parameter such that variables in separate monitoring contexts have different scopes. Additionally, the variable name is put within single inverted commas to make sure the variable names are treated as atoms[3]. This is achieved with the help of some complicated regular expressions.

Code 5.6: conditionToCode implementation

```
public String conditionToCode(Condition c, Context cxt) {
    StringBuilder sbr = new StringBuilder("");
    sbr.append(c.getName() + "() -> \r\n");
    sbr.append("\tlog4erl:info(\"Evaluating condition {" + c.getName()
          + "}\"),\r\n");
    String text = addContextToVarCalls(cxt.getID(),
                  atomizeVarCallVariables(c.getText())).trim();
    sbr.append("\t" + text + ".\r\n"); // a system side condition
    return sbr.toString();
}
```

To demonstrate the result with an example, Specification 5.7 continues on the example used above. A function is created within context source file to implement the specified condition.

Specification 5.7: Event and condition definitions for checking a function's output

```
global {
    events  { testEvent(list , return) = { execution testModule.length(list)
       uponReturning(return) } }
    conditions  {
       systemSide {
```

[3]Literals enclosed within single inverted commas are atom types[4]

```
            lengthMatches = {
                erlang:length(var:get(list)) == var:get(return)
            }
        }
    }
}
```

Code 5.8 shows clearly the addition of a logging call, that logs a message whenever the function executes, and the modifications to the *var:get* calls discussed above.

Code 5.8: Generated condition function

```
lengthMatches() ->
    log4erl:info("Evaluating condition {lengthMatches}"),
    erlang:length(var:get(0, 'list')) == var:get(0, 'return').
```

5.2.3 Monitor communication

The communication architecture between the system-side monitor and the remote-side monitor is largely predetermined by polyLARVA because the polyLARVA language plugins are irrelevant to how the polyLARVA compiler creates the remote-side monitor and the remote-side monitor expects the system-side monitor to communicate with it in a particular manner.

In order to support the monitoring of parallel systems, a new socket connection has to be established every time the system-side or remote-side monitor needs to start communication to the other monitor. The same socket connection is used for synchronous communication until the monitoring for the triggered event is complete. Both the system-side and remote-side monitors need to have a process (or thread) listening on a port for new socket connections; the central monitor could trigger internal events from timer events which may require the evaluation or execution of system-side conditions or actions respectively.

The messages are sent over a TCP/IP protocol using host names and port numbers read from a configuration file. This file can be edited by the user to

change host names and port numbers in case the default port is already in use. The configuration file is generated automatically, the format of the file is written by writing two-valued tuples delimited by a period (or full stop). For instance the default host name and port number of the remote-side monitor are generated as follows.

Config. 5.9: Configuration for TCP configuration

```
{monitor.port, 4444}.
{monitor.host, localhost}.
```

Message Types and Formats[15]

Conceptually the message types to communication through the TCP protocol are the following:

STARTUP A message sent at the beginning when the monitored system starts up. This allows the central monitor to initialize states and start any timers.

ASPECT_EVENT The is a message sent by the system-side when the monitored system triggers an event.

SYSTEM_CONDITION_EVAL This message is sent by the remote-side monitor when it needs the system-side monitor execute a condition or action. The reply to the central monitor uses the same message type.

RETURN_CONTROL This message is sent by the remote-side monitor when the monitoring for a particular event is complete and control is returned to the monitored system. In the case, the message is sent is "Done.".

SHUTDOWN The message is sent by the system-side monitor when the system shuts down.

There are only three message type identifiers, excluding the RETURN_CONTROL message type. The STARTUP message identifier is the same as that of the ASPECT_EVENT type but uses specific event identifier which always means the

STARTUP message. The STARTUP message is sent when the user runs monitor:start/0 which performs some initialization and sends the STARTUP message to the central monitor.

The actual message is always a String which consists of comma-separated values in the following order.

message type identifier This is an integer identifying the message type

message identifer This can refer to an event identifier of a condition or action identifier depending on the message type

list of parameters These are String representations of values of primitive Java types or of unique identifiers for system specific types.

Communication control flow[15]

The flow of control is described for when an event is triggered from the monitored system.

1. System-side monitor requests a connection with the central monitor and acquires a socket
2. An ASPECT_EVENT message is sent through the acquired socket
3. Remote-side monitor will evaluate rules related with the event and send a SYSTEM_CONDITION_EVAL message if a system-side condition or action needs to be evaluated
4. System-side monitor executes condition or action and replies the result back to the remote-side monitor.
5. Remote-side monitor sends a RETURN_CONTROL message
6. System-side monitor closes connection and returns control to the monitored system

5.3 Logging

The system-side monitor can log the actions occurring within the system-side monitor to a file during execution. Separate logging levels are assigned to different types of logging messages. A configuration is generated by default which can be edited by the user. Functionality exists to add loggers such as adding logging to more than one file at different logging levels or adding logging to the terminal. The logging is implemented with the use of the log4erl[4] library, the log4erl manuals should be referenced for more information on modifying the configuration file.

Logging levels have the following order: *debug*, *info*, *error* and *none* where the first level logs all messages and the last level logs nothing. Logging messages that fall under these logging levels are as follows.

error Logs which include any unexpected errors which occur during execution.

info Logs at this level include logging the evaluation of conditions and execution of actions, whenever an event is triggered and when the system is shut down.

debug Logs at this level include logging every message sent and received to and from the central monitor.

5.4 Suggestions for polyLarva

One problem in implementing this plugin relates to how variables are communicated between the system-side and remote-side monitor. Whenever an event message is triggered and an event message is sent to the remote-side monitor, a list of variables related to the event are sent to the remote-side monitor.

Since the central monitor reads these variables in a particular order, the system-side monitor must send these variables in that exact order. Unfortunately, this order is not particularly well-documented and coupled with the further problem that the intermediate object-oriented representation of these variables had a lot of

[4] log4erl: https://code.google.com/p/log4erl/

cases which needed to be treated differently, created a considerable and unnecessary hurdle. A suggestion to fix this problem would be for future implementations of polyLARVA to provide a generic method to obtain all the required variables that need to be sent to the central monitor for a particular event in the required order. This would hide details about the polyLARVA parser which are not particularly necessary for developers implementing system-side monitors for polyLARVA.

5.5 Conclusion

Once a specification script is written, monitoring is achieved by using the polyLARVA language compiler to compile the script with the "erlang" language name (not case-sensitive). The following template shows how the Erlang plugin is used.

```
java -jar LarvaLangCompiler.jar -d <target_dir> -s <path_to_script_file> -l erlang
```

where <target_dir> is the directory in which the monitoring files are generated and <path_to_script_file> is a pathname to a specification script from which the monitors are generated. After generating the files, the source files are compiled and the monitored system is compiled with the generated aspects using the implemented AOP implementation. The final result is that, when run, the augmented monitored system will itself check verify its correctness as it executes through execution path, executing actions on the system and writing monitoring actions to a log file as necessary.

The next chapter evaluates the implemented plugin using Riak, an open-source database, as a case-study.

6. Evaluation

The aim of this work is to explore the extensibility of polyLarva to define and monitor Erlang behaviour. Support for Erlang must be seamlessly integrated with the rest of the polyLarva framework. The specification constructs identified to monitor Erlang behaviour and other design decisions in the Design chapter will be made evaluated here.

The implemented Erlang plugin has been used to monitor Riak, an open-source Erlang project. This chapter will start with an introduction to Riak and then three properties describing behaviour in Riak will be discussed in detail. The usability of the developed plugin is discussed with each monitored property.

6.1 Introduction to Riak

Riak[1] [3] is a large-scale open-source project written mainly in Erlang. It is a distributed NoSQL database system or more specifically, a distributed key-value store.

Riak has been created by Basho Technologies[2] and its developers were mainly driven by the need to have a database that has data available at low latency at a large scale. Two other influences in the development of Riak are the CAP theorem

[1]Riak website: http://docs.basho.com/riak/latest/ (last accessed May 2013)
[2]Basho website: http://basho.com/ (last accessed May 2013)

(defined below) and Amazon's paper[10] on Dynamo, a key-value store used by Amazon (an online shop). Like Dynamo, Riak is completely decentralized; every node in Riak is identical such that there is no single point of failure. This makes Riak fault-tolerant because it can keep the data available even if several nodes go down. Also, like Dynamo, Riak replicates the values stored to a given number of nodes.

Riak is also eventually consistent because as stated in the CAP theorem it has to make a trade-off between availability and consistency. The CAP theorem states that, at most, a database can only satisfy two of the following desirable guarantees:

Consistency Nodes see the same data at the same time.

Availability Every request gets a response.

Partition tolerance Tolerance to message losses or failure in the system. In the real world, messages will be lost and failures will occur so Riak had to satisfy this property leaving the main trade-off between consistency and availability.

Therefore, by favoring availability Riak cannot be consistent but it is eventually consistent which means the nodes will eventually see the same data. Object version control is used to keep track of the latest version of the values stored in the database which works by storing a timestamp with each copy of the value. These timestamps are actually vector clocks which represent a logical time. Since Riak is eventually consistent, vector clocks are essential to determine which Riak node has the latest version of an object.

6.1.1 Basic Functionality

Riak is essentially a key-value store (or a map data structure), which associates keys to values. Every stored value (or data item) is associated to a unique key and given a key, Riak searches and returns a value associated with that key if it exists. Riak also has buckets which are namespaces for keys to allow a key name to be re-used across multiple buckets; buckets have configuration options such as replication

factor (a number of physical nodes on which a stored value is to be replicated) and pre/post-commit hooks. Data values are therefore actually associated to <Bucket, Key> pairs. This mapping is stored within a Riak data structure called *Riak object* which also contains other data such as a vector clock.

The basic operations that can be accessed from a Riak client connected to a Riak node are the following.

- Create a riak object given a <Bucket, Key> pair and a value to store
- Get value from a riak object
- Store a riak object
- Delete/Get a riak object given a <Bucket, Key> pair

When storing a value, a *W value* may be given to specify how many replicas need to be written before a store operation succeeds. The default W value is 2 and must not be greater than the replication factor n_val. When retrieving a stored value, an *R value* may be given to specify how many replicas need to agree before returning successfully.

6.1.2 Riak key-value stack

The internal workings of Riak can be viewed as a stack of layers which is illustrated in Figure 6.1. The Riak stack shows simplicity at the edges whereas the center deals with distribution and deals with the complexities that come with it. The numbers at some of the layers indicate the properties that are monitored on Riak (defined further below) where they are numbered from 1 to 3 based on the order in which they are presented. Each property defines behaviour at a

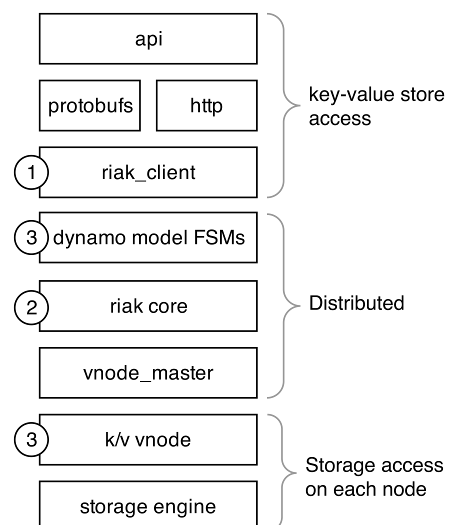

Figure 6.1: Riak key-value stack

particular layer/s in the Riak stack as illustrated in Figure 6.1 and as explained later.

Riak clients are at the API level. They provide programmatic access to a Riak database and are implemented in numerous languages including Erlang and Java. Riak clients use the *riak_ client* as an interface to the Riak database and can communicate through protocol buffers (*protobufs*) or HTTP which for the purpose of this work may be simply regarded as different protocols of communication. The *riak_ client* module uses finite-state machines (FSMs, inspired by Amazon's Dynamo) to issue requests (such as put or get requests) across multiple nodes. *Riak core* implements the fundamental distribution functionality such as vector clocks. The *vnode master* is a server implementation listening for operation requests and is the entry point to a node's storage backend, a physical node has multiple vnodes (this is described in detail further below).

There are multiple implementations for a *storage engine* that may be used by Riak, the *k/v node* module is an abstraction layer over the multiple storage engines.

6.1.3 Topology

Riak is intended to operate over multiple physical servers which are used to store data among them. This distribution is hidden from systems using Riak as explained above. When Riak stores data, it must decide on which nodes the data will be stored such that the data is evenly distributed among the nodes. The stored data is replicated over a given number of the Erlang nodes such that in the event that a node fails and becomes unavailable, other nodes will contain data to maintain availability and whenever the failed node comes back online, it is gradually updated.

Riak uses a partitioning space to distribute data across a number of equal-sized partitions of a 160-bit integer space. An equal amount of these partitions are owned by a node, and act as a virtual node or *vnode*. Riak uses *consistent hashing* to determine which nodes are used to store or retrieve data. In Riak, the partitioning space is known as a ring, with equal-sized evenly distributed partitions

belonging to all the nodes running Riak. Operations in Riak hash a given <Bucket, Key> pair to a value within the partitioning integer space to identify a number of adjacent vnodes (depending on R or W value) to perform the operation. If a node to which a vnode belongs to is not online, subsequent vnodes in the ring are chosen to perform the operation.

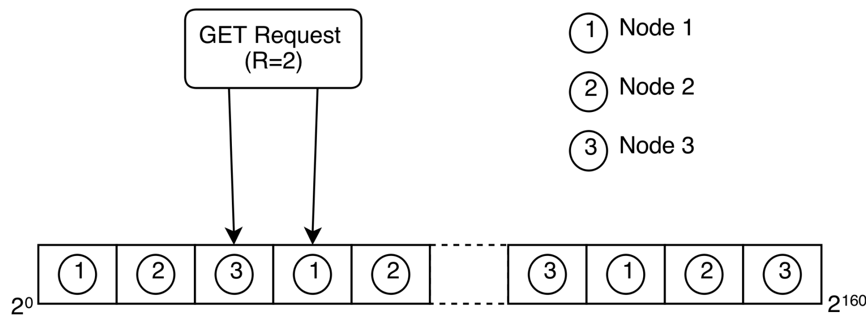

Figure 6.2: GET request hashing on three node cluster

Figure 6.2 shows the ring of a three node cluster where the integer-space is distributed evenly for the three nodes. A GET request hashes to two nodes, node 2 and 3. If Node 2 is not working, Node 1 is chosen because it is the next free node on the ring after Node 3.

Ring state and bucket properties are shared among nodes using a *gossiping protocol*. When a node changes its claim on the ring, the change is announced via the protocol. Riak nodes randomly share their ring state to a random peer at random times to make sure the nodes did not miss any updates.

6.2 Properties about Riak

The following Riak properties are a description of the behaviour of specific aspects about the system. Properties are defined using the specification developed in this work to define Erlang behaviour which now makes part of polyLARVA's rule-based specification language. Riak's source code had to be referenced to find meaningful points within the source code on which to specify events in order to specify some

meaningful behaviour which Riak is supposed to respect. The identified properties are written in separate specification files although all the properties may be put into one specification script to generate one system-side monitor to monitor all the identified properties; Erlang code may be written in separate Erlang source files if necessary. The term property is used loosely here, it will usually refer to the correctness checking performed in one specification script although that correctness checking may be described as multiple properties.

The identified properties are monitored on a three-node cluster. Three identical instances of Riak operate, on separate Erlang nodes, as a single key-value store while stored data is distributed among them. A system-side monitor in attached to each instance of Riak which communicate to a central monitor such that monitoring across multiple components is achieved where necessary.

Figure 6.3 illustrates communicates between the system-side monitors of a Riak cluster and the central monitor. Each system-side monitor creates a new TCP socket connection each time it needs to communicate to the central monitor. Hence the separate system-side monitors are indistinguishable to the central monitor unless a Pid variable referring to Pid of the process triggering the events is used specifically in the specification to identify between the processes and by consequence between the system-side monitors.

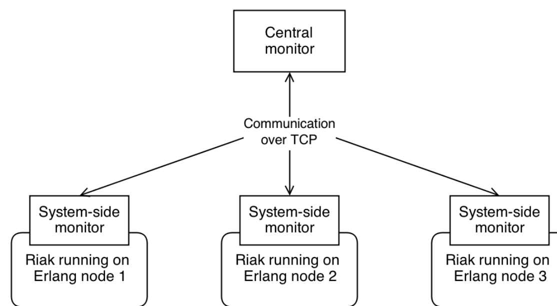

Figure 6.3: Riak cluster communication to the central monitor

6.2.1 Property 1: High-level functionality

The first property to be observed about Riak is a basic course-grained property to monitor Riak's operations at a high-level. The property monitors Riak's behavior as a key-value store.

The property monitors Riak's ability to store and retrieve data by checking that when data is inserted using <Bucket, Key> pair, that same data is returned given the same <Bucket, Key> pair unless the data was deleted from the system. This is the most high-level property about Riak as it only monitors the external behaviour of the system without the requirement to understand the internal workings of Riak. Figure 6.1 illustrates this property as monitoring behaviour at the *riak_ client* layer.

Monitoring property on system-side

When communicating to a Riak cluster of nodes, operations on the database are performed through one of the nodes. Each Riak node has a *riak_ client* module which acts as an interface for Riak by implementing the basic operations to the database, that is, the whole cluster. The events extracted from Riak to monitor this property are taken from the *riak_ client* module. Therefore when monitoring on the system-side, monitoring is performed separately on each of the nodes. The property being monitored here is high-level and treats a cluster of nodes as one data store so if the property uses a state, that state is not specific to one node but to the whole cluster. Therefore if the property maintains a state on the system-side it becomes a necessity to perform all the operations on Riak through the same node in order for the correctness property to be valid.

Specification 6.1 implements the specification for the described property by performing all the monitor checking on the system-side. Whenever a riak object is retrieved from Riak, the system-side monitor checks that the value of the retrieved object is the same as the value inserted into Riak with the <Bucket, Key> pair. Monitoring this on the system-side requires that all operations on the database occur through the same node as the monitor maintaining the state has to maintain

the state across the whole cluster and the system-side monitors cannot share data.

The implemented events specify the points in Riak where a riak object is *put* into the database; when Riak *get*s a riak object and when it *delete*s an object given a <Bucket, Key> pair. The given R or W values (or other values) are ignored here since we will only monitor the values being retrieved or inserted given the associated <bucket, key> pair.

The monitoring state is maintained using an ETS table (ETS is an Erlang module which implements a built-in Erlang storage[4]) which maps tuples of <Bucket, Key> pairs to Erlang terms; the variable *objects* just stores the table ID. Whenever a put or delete operation is performed on Riak, the system-side monitor does the same operation on its local ETS table and when a get event is triggered, the system-side monitor checks if the value of the returned riak object corresponds to the value stored on the monitor ETS table. This property is only interested in the correctness of Riak when the operations return successfully so any unsuccessful operations are ignored. The delete event is not vital for defining correctness here but it is used to remove values whenever it is no longer necessary to store them.

Specification 6.1: High-level monitoring on system-side

```
global {
   events  {
      put(obj, return) = { execution riak_client.put(obj, _opts)
         uponReturning(return) }
      get(bucket, key, return) = { execution riak_client.get(bucket, key, _opts)
         uponReturning(return) }
      delete(bucket, key, return) =
         { execution riak_client.delete(bucket, key, _, _) uponReturning(return) }
   }

   states  {
      systemSide {
         var objects {
            restoreWith { var:put(objects, ets:new(objects, [set])) }
         }
      }
   }
```

```
  conditions {
    systemSide {
      succeeded = { ... }

      matchesState = {
        TableID = var:get(objects),
        Bucket = var:get(bucket),
        Key = var:get(key),
        {Flag, RObj} = var:get(return),
        case ets:lookup(TableID, {Bucket, Key}) of
          [{_, Res}] -> (Flag==ok) andalso (Res==riak_object:get_value(RObj));
          [] -> (Flag == error) and (RObj == notfound)
        end
      }
    }
  }

  actions {
    systemSide {
      removeObject = {
        ets:delete(var:get(objects), {var:get(bucket), var:get(key)})
      }
      addObject = {
        TableID = var:get(objects),
        Obj = var:get(obj),
        Bucket = riak_object:bucket(Obj),
        Key = riak_object:key(Obj),
        Value = riak_object:get_value(Obj),
        ets:insert(TableID, {{Bucket, Key}, Value})
      }
    }
    monitorSide {
      putGetError = {System.out.println("Error: incoherent object retrieval");}
    }
  }

  rules {
    addObjectRule = put(obj, return) \ succeeded -> addObject;
    removeObjectRule = delete(bucket, key, return) \ succeeded -> removeObject;
    matchStateRule = get(bucket, key, return) \ (succeeded && !matchesState)
        -> putGetError;
  }
}
```

Table 6.1 shows the event specification constructs used in Specification 6.1. Only function execution is used to detect Erlang behaviour although function calls could be used just as easily instead of function execution. The only event variation used is *uponReturning* such that the result of the function is obtained in the specification to check if the function calls were successful depending on the output and to verify the correctness of retrieved values.

Erlang events		Event variations	
Used	Events	Used	Variations
Yes	function execution	No	default (immediately before event)
applicable	function call	Yes	uponReturning
No	send message	No	uponHandling
No	receive message		

Table 6.1: Used event constructs in Specification 6.1

Table 6.2 shows the system monitoring constructs used in Specification 6.1, that is the system-side state, conditions and actions. All three constructs were needed to specify the property since the monitoring is specified on the system-side and the state is required to mimic the key-value store by maintaining a replica of it on the monitor.

Used	System-side construct
Yes	State
Yes	Conditions
Yes	Actions

Table 6.2: Used system-side monitoring constructs in Specification 6.1

System-side monitor logging

The system-side generated from Specification 6.1 was tested and Riak was shown to respect the property as expected. A look at the logging file reveals the monitoring actions that were carried out. Log 6.2 shows the logs from when monitor was started and after one put operation. The message "1,0,null" is the STARTUP message to

notify the central monitor that the Riak node started. The put operation triggered the event twice. There are multiple put functions which all ultimately call the put function which takes two arguments. The reason that the event was triggered twice is that there are two function clauses for the put function depending on whether the second argument is a list. The put functions call the put function where the second argument is not a list while the latter calls the put operation with two arguments with the second argument as a list. Essentially, the put function of arity two which the specification defines calls itself once causing the event to be triggered twice. The same thing happens for the get operation. This does not cause problems with correctness although it causes the associated rules to be triggered twice. The extra event can be ignored by adding an extra condition with checks that the second argument is a list, the events will still be triggered twice however the associated actions will only execute once.

Log 6.2: System-side logging for high-level property

```
[debug] Sending message: "1,0,null\n"
[debug] Message received from monitor: "Done."
[info] Sending event notification: _put0
[debug] Sending message: "1,1,null\n"
[debug] Message received from monitor: "2,1,␣"
[info] Evaluating action {updateObjectsAction}
[debug] Sending message: "2,1,true,\n"
[debug] Message received from monitor: "Done."
[info] Sending event notification: _put0
[debug] Sending message: "1,1,null\n"
[debug] Message received from monitor: "2,1,␣"
[info] Evaluating action {updateObjectsAction}
[debug] Sending message: "2,1,true,\n"
[debug] Message received from monitor: "Done."
```

Monitoring property on remote-side

Maintaining the state on the remote-side does not require all Riak operations to be accessed from one node. For demonstration purposes, Specification 6.3 shows the specification for the same property but with the monitoring state being maintained

on a remote monitor to showcase the differences involved.

A limitation of monitoring on the central monitor is that it cannot interpret Erlang terms except for equality comparisons. The where clause is used to extract the value from a riak object such that the value is sent to the central monitor instead of the riak object. This value is extracted on the system side such that it does not need to be done on the remote-side monitor. Also of note is that the event variables used on the remote-side monitor are declared with the *var* keyword such that the polyLARVA compiler recognizes them as variables and uses as variables of type String. The user must aware of the type of these variables on the central monitor as s/he may have to utilize them as a String as shown in the monitor-side conditions and actions where bucket and key variables are concatenated to make a key for the Hashmap used to maintain monitoring state on the central monitor.

Specification 6.3: High-level monitoring on remote-side

```
imports{ monitorSide { import java.util.HashMap; } }
global {
  events {
    put(var value, return) = { execution riak_client.put(obj, _opts)
      uponReturning(return) }
      where { value = {Flag, Obj} = var:get(return), riak_object:get_value(Obj)}
    get(var bucket, var key, var value, return) = {
      execution riak_client.get(bucket, key, _opts) uponReturning(return) }
      where { value = {Flag, Obj} = var:get(return), riak_object:get_value(Obj)}
    delete(var bucket, var key, return) = {
      execution riak_client.delete(bucket, key, _, _) uponReturning(return) }
  }

  states {
    monitorSide {
      HashMap objects {
        restoreWith { objects = new HashMap(); }
      }
    }
  }

  conditions {
    systemSide { ... }
    monitorSide {
```

```
            matchesState = {
               String k = bucket+","+key;
               if(valueTable.containsKey(k)) {
                  return value.equals((String) objects.get(k));
               } else {
                  return false;
               }
            }
         }
      }

      actions {
         monitorSide {
            putGetError = {System.out.println("Error: incoherent object retrieval");}
            addValue = {
               String k = bucket+","+key;
               objects.put(key, value);
            }
            removeValue = {
               String k = bucket+","+key;
               objects.remove(k);
            }
         }
      }
      rules { ... }
}
```

Table 6.3 shows the event specification constructs used in Specification 6.3 were the results obtained are the same as the results obtained for Specification 6.1. The only difference for the event definitions is the use of the *var* word as a type placeholder and the use of the where clause to access system-specific data which cannot be accessed from the remote-side monitor.

Erlang events		Event variations	
Used	Events	Used	Variations
Yes	function execution	No	default (immediately before event)
applicable	function call	Yes	uponReturning
No	send message	No	uponHandling
No	receive message		

Table 6.3: Used event constructs in Specification 6.3

Table 6.4 shows the system monitoring constructs used in Specification 6.3 were only conditions are used in this approach. The use of the monitoring state and actions were moved to the remote-side monitor. In the case that the monitor would need to perform an action on the monitored system, system-side actions would be required.

Used	System-side construct
No	State
Yes	Conditions
No	Actions

Table 6.4: Used system-side monitoring constructs in Specification 6.3

In either case, both the system-side and remote-side specifications should be expected to slow down the system as every operation needs to be replicated on the monitor. Additionally, the either monitoring approach (the system-side approach needs to maintain all the state on a single monitor) creates a single of point of failure which is a liability for Riak especially since Riak was designed with availability in mind so single of failure is unacceptable in a real-world environment. If the monitor were to fail it would fall out of sync with Riak, which means that monitoring for this property would have to be discontinued once the monitor fails. For these reasons, this property only serves as a gentle monitoring example to introduce monitoring on Riak.

6.2.2 Property 2: Vector Clocks

Vector clocks are a logical timestamp used to synchronize a Riak cluster to the latest riak objects inserted into it. Vector clocks are used instead actual exact timestamps because it is difficult to synchronize computer clocks perfectly together which may subject Riak to error. Vector clocks do not have an absolute time but rather have time in the sense that when vector clocks are compared with each other, an order from earliest to the latest vector clock can be obtained although it is possible that it cannot be determined which of two vector clocks is the latest (for instance when

two nodes update the same object independently). If a vector clock descends from another vector clock, it means that the vector clock is more recent. Whenever a node updates a riak object, the vector clock is incremented. When two or more vector clocks do not descend from each other, they can be merged to create a vector clock that descends from all the vector clocks used to merge it.

This property [12] about vector clocks checks that when a vector clock is created from another vector clock, it is also its descendant. More specifically Specification 6.4 checks for the following.

- Incrementing a vector clock v results in a vector clock v' such that v' is a descendent of v
- Merging a list of vector clocks l results in a vector clock v such that v is a descendant of each vector clock in l

Specification 6.4 does not need to maintain a monitoring state, it just checks that the result of the *increment* and *merge* functions is a descendant of the vector clock/s in the parameters. Checking for descent is done using the function *vclock:descent/2* which is also a function from a Riak module. It returns true if the first parameter is a descendant of the second parameter.

Upon detection of an error, the specified action is to print an error at the central monitor. It would be important to print the exact parameters used when error occurred such that it may provide some insight for Riak developers who would later examine the error; this wasn't written in the specification to keep the script simple to understand. Additionally, it may be a good idea to restart the faulty riak node in case the incorrect result occurred because the node entered a state under which it operates incorrectly and by restarting the node it may start from a new state without going back to a faulty state.

Specification 6.4: Monitoring vector clock descent

```
global {
  events {
    inc(vclock, return) = { execution vclock.increment(_, _, vclock)
```

```
        uponReturning(return) }
      merge(clock_list, return) = { execution vclock.merge(clock_list)
        uponReturning(return) }
  }

  conditions {
    systemSide {
      incDescends = {
        vclock:descends(var:get(return), var:get(vclock))
      }
      mergeDescends = {
        List = var:get(clock_list),
        Return = var:get(return),
        lists:all(lists:map(fun(X) -> vclock:descends(Return, X) end, List))
      }
    }
  }

  actions {
    monitorSide { incError ... mergeError ... }
  }

  rules {
    incRule = inc(vclock, return) \ !incDescends -> incError;
    mergeRule = merge(clock_list, return) \ !mergeDescends -> mergeError;
  }
}
```

Table 6.5 shows the event specification constructs used in Specification 6.4. Only the returning variation of function execution constructs were necessary to define the Erlang events although the function call is again applicable instead function execution.

Erlang events		Event variations	
Used	Events	Used	Variations
Yes	function execution	No	default (immediately before event)
applicable	function call	Yes	uponReturning
No	send message	No	uponHandling
No	receive message		

Table 6.5: Used event constructs in Specification 6.4

Table 6.6 shows the system monitoring constructs used in Specification 6.4 were again only conditions are used. This property checks for verification correctness immediately as the event is triggered by verifying the output of a function against its input variables.

Used	System-side construct
No	State
Yes	Conditions
No	Actions

Table 6.6: Used system-side monitoring constructs in Specification 6.4

Unlike the first property, this property does not create a single point of failure because the property monitors behaviour on a single node rather than behaviour on an entire cluster of nodes. Therefore, this category of properties translate well for monitoring Riak while it used for long term in real-world environment.

6.2.3 Property 3: Riak Distribution

This property is concerned with object replication across multiple nodes. Riak replicates inserted riak objects to a given number of nodes (default is three nodes[2]) and uses W/R values to fine-tune consistency as explained above. The property checks for the following.

- the number of successful put or get operations on each node is greater or equal to the given W or R values respectively
- the values written to each node are the same value as the value used in the put operation that caused the per-node operations. (the same may be done for the get operation but was not implemented to maintain simplicity as it is fairly redundant in terms of how it is written.

Specification 6.5 monitors behaviour across multiple nodes and requires feedback from each node to check the property. Therefore the monitoring must be performed on the remote-side such that it may receive events from multiple nodes.

Specification 6.5 checks for corresponding values separately from the checking of the corresponding number of per-node put/get operations. Each put/get operation is given a request ID in Riak. A put/get request with the request ID is then sent to each of the nodes on the ring which the hash function selects (this process is explained in detail above in Riak's introduction). The specification uses the request ID to relate the per-node operations to the high-level operation. For instance if a put event with W=2 is triggered, the monitor will check that by the time the put function returns two other events will be triggered to perform the per-node operation with the same request ID and yet other events are triggered for each of the per-node operations to check that the value inserted is the same as the value given to the high-level operation.

The events used to generate the put/get events are triggered from the *riak_kv_put_fsm* or *riak_kv_get_fsm* modules. These modules among others implement the dynamo model FSMs mentioned in Riak's introduction. At this point in the put/get operations the request ID is exposed in the parameters. The FSMs then send the necessary requests to multiple nodes to perform the put/get operation locally.

The events used to generate the events for the put/get operations on each local node are triggered from the module *riak_kv_vnode* which is the k/v vnode layer mentioned in the Riak stack in Riak's introduction. This point in the program abstracts over the storage engine used.

Correctness is checked when the put/get operations return for the node operation counting part while the value correspondence is checked every time the put/get operation is performed per-node locally.

Specification 6.5: Monitoring distibution in put/get operations

```
imports{ ... }
global {
  events {
    begin_put(int reqid, var value) = {
      execution riak_kv_put_fsm.start_link(from, obj, _) }
      where { reqid = element(2, var:get(from));
```

```
            value = riak_object:get_value(var:get(obj)); }
    nodePutVal(int reqid, var value) = {
        execution riak_kv_vnode.do_put(_, _, obj, reqid, _, _, _) }
        where { value = riak_object:get_value(var:get(obj)); }

    put(int reqid, int wOrR, return) = {
        execution riak_kv_put_fsm.start_link(from, obj, opts) uponReturning(return) }
        where { wOrR = proplists:get_value(w, var:get(opts), 2);
                reqid = element(2, var:get(from)); }
    get(int reqid, int wOrR, return) = {
        execution riak_kv_get_fsm.start_link(from, _, _, opts) uponReturning(return) }
        where { wOrR = proplists:get_value(r, var:get(opts), 2);
                reqid = element(2, var:get(from)); }

    nodePut(int reqid) = { execution riak_kv_vnode.do_put(_, _, _, reqid, _, _, _) }
    nodeGet(int reqid) = { execution riak_kv_vnode.do_get(_, _, reqid, _) }
}

states {
    monitorSide {
        HashMap countTable {
            restoreWith { countTable = new HashMap(); }
        }
        HashMap valueTable {
            restoreWith { valueTable = new HashMap(); }
        }
    }
}

conditions {
    systemSide { ... }
    monitorSide {
        satisfiesCount = {
            Integer key = new Integer(reqid);
            return countTable.containsKey(key)
                   && wOrR <= (Integer) countTable.get(key);
        }

        matchesValue = {
            Integer key = new Integer(reqid);
            return valueTable.containsKey(key)
                   && value.equals((String) valueTable.get(key));
        }
    }
```

```
  }

  actions {
    monitorSide {
      addValue = {
        Integer key = new Integer(reqid);
        valueTable.put(key, value);
      }
      addCount = {
        Integer key = new Integer(reqid);
        if(countTable.containsKey(key)){
          countTable.put(key, ((Integer) countTable.get(key)) + 1);
        } else {
          countTable.put(key, new Integer(1));
        }
      }
      valueError ... putError ... getError ...
    }
  }

  rules {
    addValueRule = begin_put(int reqid, var value) -> addValue;
    checkValueRule = nodePutVal(int reqid, var value) \ !matchesValue -> valueError;

    putAddCountRule = nodePut(int reqid) -> addCount;
    getAddCountRule = nodeGet(int reqid) -> addCount;
    putCheckCount = put(int reqid, int wOrR, return)
      \ (succeeded && !satisfiesCount) -> putError;
    getCheckCount = get(int reqid, int wOrR, return)
      \ (succeeded && !satisfiesCount) -> getError;
  }
}
```

Table 6.7 shows that Specification 6.5 also uses the function execution construct to define behaviour however both the variations, before and after the function execution, were required to defining the necessary Erlang behaviour.

Table 6.8 shows the system monitoring constructs used in Specification 6.5 were yet again only conditions are used to check if function calls were successful. Since the monitoring is required to be specified on the remote-side, there is no use the system-side monitoring state and system-side actions.

Erlang events		Event variations	
Used	Events	Used	Variations
Yes	function execution	Yes	default (immediately before event)
applicable	function call	Yes	uponReturning
No	send message	No	uponHandling
No	receive message		

Table 6.7: Used event constructs in Specification 6.5

Used	System-side construct
No	State
Yes	Conditions
No	Actions

Table 6.8: Used system-side monitoring constructs in Specification 6.5

The central monitor in this property is the single of failure but this vulnerability only effects the monitoring as in the case that the central monitor were to fail, the augmented Riak system would fail to connect to the central monitor and continue to function as normal. This difference from the first property is that the monitoring state for this property is only required to maintain state related to the current put/get operations occurring on Riak. Once those operations are completed the state related to their request IDs is no longer required. This means that the central monitor may be restarted while Riak continues operation. In contrast, the monitoring state for the first property is required to be synchronized with all the operations on Riak from when it starts executing so once the monitor fails it is no longer relevant.

Therefore even though this property monitors behaviour over an entire cluster, it can perform monitoring while keeping Riak available as it was meant to be since a failed monitor can be recovered without restarting the Riak cluster.

6.3 Conclusion

The monitored specifications above show that the Erlang plugin for polyLarva implemented in this work can be used effectively to monitor behaviour in Riak. The first property is simplistic as it does not require knowledge of Riak's internal working but it was shown impractical for actual use. The last two properties monitor the internal behaviour in Riak and were shown to be acceptable properties to be monitored in Riak in a real-world setting.

The specification constructs used to specify the properties identified for Riak were identified throughout the properties section of this chapter. Tables 6.9 and 6.10 together show the specification constructs used or applicable for the identified properties. They were no identified properties in Riak that required the monitoring the primitive operators for sending and receiving messages. In Riak, these operations are usually encapsulated within functions implementing behaviors such as a generic server. That is, even though messages are being communicated between processes, such as in the third property where operation requests are sent to multiple Riak nodes, this functionality is encapsulated within generic implementations which send and receive messages using function calls (or executions). The unused *uponHandling* event variation is related to the Erlang event where a message is received by the process upon the event where a message is read by the system. Therefore, the used constructs for defining events were not required although they might still prove useful for other systems or properties.

Erlang events		Event variations	
Used	Events	Used	Variations
Yes	function execution	Yes	default (immediately before event)
Yes	function call	Yes	uponReturning
No	send message	No	uponHandling
No	receive message		

Table 6.9: Usable event constructs in total (for the properties identified)

Other features in the Erlang plugin used to define the identified properties in-

Used	System-side construct
Yes	State
Yes	Conditions
Yes	Actions

Table 6.10: Used system-side monitoring constructs in total

clude the use of specifying Java types or the *var* placeholder to send event variables to the remote-side monitor which was especially made use in the third property. The where clause was mainly used to access data from the event variables before it is sent to the remote-side monitor.

Communication through a TCP protocol was necessary for system-side monitors to communicate to the central monitor where the monitors are running on different computing entities but are connected to a common network. System-side actions may be executed on Riak such as restarting the Riak node however there was no property identified that absolutely requires an immediate action on Riak upon detection of incorrect behaviour. This means that the identified properties about Riak did not require the Erlang plugin to implement synchronous monitoring.

The following chapter describes work related to the Erlang plugin for polyLarva.

7. Related Work

This chapter describes the similarities and dissimilarities of work related to this project including ELARVA and the polyLARVA plugins for Java and C.

7.1 polyLARVA plugins

The languages supported for polyLARVA before the beginning of this project were Java and C. These two plugins[15] are very similar in terms of the issues regarding their use and implementation. Like the Erlang plugin developed in this work, events for these plugins are elicited using AOP libraries; AspectJ for Java and ACC (AspeCt-oriented C compiler) for C. The events supported for the two plugins include method/function call and execution pointcuts. Unlike Erlang, variables for both languages are declared with a type and are mutable which makes the definition and translation of the specification much more straight forward.

Event definitions for Java take the following form:

```
ObjectType object.methodName(Type arg1, Type arg2, ...)
```

whereas an event definition for C is defined in the following form:

```
moduleName.functionName(Type arg1, Type arg2, ...)
```

Note that the use of types here is entirely straight-forward and seamless with the languages used. Like Erlang the list of argument is finite but a last argument using

asterisk (*) would signify, zero or more arguments in addition to the number of arguments specified before the asterisk.

In Erlang, variables types are not declared so event variables do not specify a type in the event definition except if the variable is used on the remote-side monitor. This became complicated because the remote-side monitor also makes use of the event variables but being written in Java it requires type declarations for event variables with primitive Java types. So variables for Erlang event definitions have to create a balance between being variable names in a functional language and variables for the statically-typed Java.

Updating variables is also straight-forward for the Java and C plugins as variables in these languages are mutable while the Erlang plugin had to make use of mapping functions to update variable state.

7.2 Elarva

Elarva is a runtime monitoring tool for Erlang. There are three main differences between Elarva and the Erlang plugin developed in this work. The first is that Elarva uses asynchronous monitoring whereas the polyLarva plugin uses synchronous monitoring. The second difference is that specification script for the two tools are significantly different; Elarva uses DATEs to define properties whereas the polyLarva plugin uses rule-based properties. The third difference is that the polyLarva plugin, by being part of the polyLarva framework, can define properties across several software components whereas Elarva can only define properties over single Erlang components. Besides defining properties across software components written in the different languages polyLarva can define properties across multiple software components running independently. The first and third Riak properties defined in the Evaluation chapter are good examples of this. Elarva would only be able to monitor the equivalent of the first version of the first Riak property which maintains the monitoring state on the system-side. As for the third

Riak property, there is no equivalent specification for ELARVA as it requires the use of a central monitor to maintain a monitoring state from multiple Riak nodes.

8. Conclusions

This chapter finalizes the report with a summary of this work and provides possible directions for future work.

8.1 Summary

The aim of this work was to explore the extensibility of polyLarva to support Erlang for a seamless integration with the original polyLarva framework. The development of the Erlang plugin may be divided into three parts: eliciting events, communication with the polyLarva central monitor and system-side monitoring. A specification to describe Erlang behaviour was defined within the language-dependent parts of the polyLarva specification to define Erlang events and system-side monitoring.

Eliciting events The Erlang plugin can extract system events from the system's source code for the following constructs: function calls, function execution, message sending and message reception. The actual event may be defined as *before* or *after* the execution of a given construct; the receive construct has three event variations: before it starts waiting for a message, upon the reception of a message and just after a message is processed.

Aspect-oriented programming (AOP) was used to select pointcuts in a system to inject monitoring code for the events defined in the specification. An AOP

library for Erlang was developed from existing AOP libraries to create a more complete library to satisfy the needs of the Erlang plugin, that is to support the required pointcuts.

Communication The Erlang system-side monitor generated from the Erlang plugin communicates with a remote-side monitor through a TCP/IP protocol. Therefore the system-side monitor can reside on a separate computing entity from that of the remote-side monitor. The system-side monitor communicates every system event to the remote-side monitor through a new socket connection and waits for instructions from the remote-side monitor. The system-side monitor also listens for unexpected events from the remote-side monitor. Communication is synchronous which allows the system-side monitor to act upon the system at the immediate point in the program where incorrect behaviour may be detected.

System-side monitoring The Erlang system-side monitor can maintain a local state, evaluate conditions and execute actions locally. The mapping functions *var:get* and *var:put* are used to associate variable names to an Erlang terms (or values) and by doing so a "variable" can be easily updated in the specification. Conditions and actions can be executed at the system-side to enable access to system-specific data.

Finally, three properties about Riak were drawn up to evaluate the developed Erlang plugin by using the plugin to monitor Riak and explore instances where monitoring is useful and other instances in which it's not.

8.2 Future Work

This section describes ideas for future work related to this work. That is analyzing performance overheads of the Erlang plugin, configuring synchronous and asynchronous monitoring by the user and a distributed central monitor.

8.2.1 Overhead analysis

Runtime verification, although it is a lightweight approach to verify correctness, incurs a performance overhead on the monitored system. Since the Erlang plugin is synchronous, the Erlang process which triggers an event has to incur the overhead of waiting for the monitoring to complete before it can continue operation. A highly parallel system would potentially incur a lower performance overhead since processes waiting for monitoring to complete would be replaced by other processes.

The configurable boundaries in polyLARVA can also be used to fine-tune the performance overhead of runtime verification using polyLARVA [8]. Analyzing the performance overhead of runtime verification is required to learn how the performance can be minimized which is necessary for wide option of runtime verification in the software industry.

8.2.2 Configuring synchronous and asynchronous monitoring

In the Design chapter, a discussion may be found about the advantages and disadvantages of synchronous or asynchronous to decide on the technique used to elicit system events where both approaches have their advantages and disadvantages. The advantage of synchronous monitoring is that it allows the monitor to act immediately on the system at the point in the program where incorrect behaviour is detected however depending on the property being monitored, this advantage may not be required in which case asynchronous monitoring would be preferable as it is theoretically faster (especially if the system is not highly parallel) because it doesn't stop system execution while the monitor checks for correctness.

This observation suggests that it would be beneficial for polyLARVA to support configuration between synchronous and asynchronous monitoring where the configuration is specified in the specification and the system-side monitor would have to implement both monitoring approaches.

8.2.3 Distributed central monitor

As discussed in the Evaluation chapter, the polyLarva central monitor creates a single point of failure (for instance, a remote-side action may crash the monitor) and while a failure would only result in the absence of monitoring while the monitored system continues running normally without runtime verification while the monitor is down. A possible solution may be to generate a distributed central monitor which would consist of identical monitors that interact each other to provide the functionality of one central monitor (much like Riak).

8.3 Concluding Remarks

In this work, polyLarva was shown to be extensible to the Erlang language, an actor-based dynamically-typed functional language. This result suggests that similar languages may also be supported by polyLarva as they are likely to face the same challenges which the Erlang plugin needed to tackle.

References

[1] G. A. Agha, P. Thati, and R. Ziaei. Actors: A model for reasoning about open distributed systems, 2001.

[2] Basho. Replication. `http://docs.basho.com/riak/1.1.4/references/appendices/concepts/Replication/`, May 2013.

[3] Basho. The riak fast track. `http://docs.basho.com/riak/latest/tutorials/fast-track/`, May 2013.

[4] F. Cesarini and S. Thompson. *ERLANG Programming*. O'Reilly Media, Inc., 1st edition, 2009.

[5] S. Colin and L. Mariani. Run-time verification. In *Model-Based Testing of Reactive Systems*, pages 525–555, 2004.

[6] C. Colombo and A. Francalanza. Towards a specification-based correctness of erlang through asynchronous monitoring. Technical report, University of Malta, 2012. WICT.

[7] C. Colombo, A. Francalanza, and R. Gatt. Elarva: A monitoring tool for erlang. In *Runtime Verification - Second International Conference, (RV)*, volume 7186 of *Lecture Notes in Computer Science*, pages 370–374. Springer, 2012.

[8] C. Colombo, A. Francalanza, R. Mizzi, and G. J. Pace. polylarva: Runtime verification with configurable resource-aware monitoring boundaries. In *Software Engineering and Formal Methods - 10th International Conference, SEFM 2012*, volume 7504 of *Lecture Notes in Computer Science*, pages 218–232. Springer, 2012.

[9] C. Colombo, A. Francalanza, R. Mizzi, and G. J. Pace. Extensible technology-agnostic runtime verification. In *FESCA*, pages 1–15, 2013.

[10] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, Oct. 2007.

[11] N. Delgado, A. Q. Gates, and S. Roach. A taxonomy and catalog of runtime software-fault monitoring tools. *IEEE Trans. Softw. Eng.*, 30(12):859–872, Dec. 2004.

[12] K. Falzon. Combining runtime verification and testing techniques. Master's thesis, 2011.

[13] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J. marc Loingtier, and J. Irwin. Aspect-oriented programming. In *Lecture Notes in Computer Science 1357*, pages 48–3. Springer Verlag, 1998.

[14] M. Leucker and C. Schallhart. A brief account of runtime verification, 2008.

[15] R. Mizzi. An extensible and configurable runtime verification framework. Master's thesis, 2012.