

polyLarva Plugin for PHP

Jonathan Attard

Supervisor: Dr. Christian Colombo



Faculty of ICT

University of Malta

March 2014

*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 PHP

submitted is my work, except where acknowledged and referenced.

Jonathan Attard

6th March 2014

Acknowledgements

I would like to express my appreciation and gratitude towards all those who, in some way or another, supported and helped me in completing this work.

Special thanks goes to my supervisor Dr. Christian Colombo, for guiding me in choosing the study area for this dissertation and for his interest, suggestions, and constructive evaluation of this work. I would like to thank my family members, for providing me with motivation and courage in times of need. And last but not least, my girlfriend, who provided me with the love, patience, and support needed in order to overcome difficult situations along the way.

Abstract

Web Applications (WA) are increasingly being expected to behave correctly with a high degree of confidence with the growing presence of mission, safety and business critical applications over the web, as they allow no room for failure as lives could be lost or financial losses could be incurred. PHP is a popular scripting language designed specifically for dynamic web page creation, used for the development of mainstream WAs such as Moodle and Wordpress. Its dynamic and concise nature aids in achieving development targets faster than the alternatives. However, its dynamicity renders verification techniques, based on exhaustive traversal of the state space, such as model checking, not suitable for its verification. Runtime Verification (RV) is a light-weight verification technique that has been shown to be a viable solution for the verification of large complex systems.

polyLarva is a technology-agnostic RV tool that can be extended for the runtime verification of systems developed in any technology, given the availability of a technology-specific plugin. In this work, we present our approach to extending polyLarva with a PHP plugin, for the runtime verification of any PHP WA. polyLarva had never been extended to scripting languages before, and this presented a new category of challenges for its extension to PHP. For polyLarva to be able to monitor a PHP WA, it is first required that target PHP WA scripts be enhanced with monitoring functionality so that their behaviour becomes observable. Once their behaviour can be observed, properties can be defined, using the polyLarva specification language, for the verification of such behaviour.

Following a thorough investigation of the PHP language and ways of expressing correctness properties on such language, we came up with a PHP plugin that was eventually tested in a case study on an e-learning PHP WA called Moodle. A number of properties, based on configurations from a real-life Moodle installation, have been successfully verified by polyLarva by means of the PHP plugin. Through the verified properties we were able to show that the PHP plugin is able to employ monitoring functionality for different real-life properties, specified on PHP script code behaviour, while supporting the flexibility allowed by polyLarva of splitting such properties across the monitored system and the monitor.

Contents

1. Introduction	1
1.1 Report Structure	3
2. Background	5
2.1 Introduction	5
2.2 polyLarva	5
2.2.1 Runtime Verification (RV)	6
2.2.2 Aspect-Oriented Programming (AOP)	9
2.2.3 polyLarva Architecture	11
2.2.4 Property Specification Language	13
2.2.5 Communication between the Remote Monitor and System- side Monitoring Code	26
2.2.6 Conclusion	27
2.3 PHP	28
2.3.1 PHP Language	29
2.3.2 Conclusion	34
2.4 Conclusion	34
3. Problem Definition	35
3.1 Introduction	35
3.2 Aims and Objectives	36
3.3 Conclusion	38
4. PHP Plugin Design	39
4.1 Introduction	39
4.2 Challenges for polyLarva in Monitoring PHP WAs	40
4.3 Enhancing the Property Specification Language for the PHP Language	41
4.3.1 Applying Event Declarations to Subsets of PHP WA Scripts	42
4.3.2 The Extraction of Monitoring Events from Unstructured PHP Code	44
4.4 Instrumentation of Event Extraction Code into PHP WA Code . . .	48
4.4.1 AOP PHP	49
4.4.2 Translating Event Declarations to AOP PHP Code	53
4.4.3 Event Declarations not Translatable to AOP PHP Code . . .	54

4.4.4	Event Generation Code	55
4.5	System-side Monitoring Logic Evaluation	55
4.5.1	Monitoring Contexts and System-side Evaluation Functionality	56
4.6	Conclusion	58
5.	PHP Plugin Implementation Details	60
5.1	Introduction	60
5.2	polyLarva Generic Plugin Interface and Structure	61
5.2.1	Extracting an Intermediate Specification for Monitoring Code Compilation	63
5.3	Monitoring Code Compilation and Instrumentation	66
5.3.1	Instrumenting Event Extraction Code into Target PHP Scripts	70
5.3.2	PHP Plugin Architecture	72
5.4	Conclusion	72
6.	Case Study	74
6.1	Introduction	74
6.2	Moodle	75
6.2.1	Property#1 - Cross-verifying Moodle Permission Management (MPM)	76
6.2.2	Property#2 - Further Protection on Moodle Administration	83
6.2.3	Property#3 - Limiting Login Attempts	86
6.2.4	Property#4 - Timing Out Expired Moodle Sessions in Real Time	92
6.3	Evaluation	96
6.3.1	Testing the Set Objectives against the Case-Study	96
6.3.2	Impact of Case-Study on the PHP Plugin	100
6.3.3	Appraisal of the Case-study	101
6.4	Conclusion	101
7.	Related Work	102
7.1	Introduction	102
7.2	Other polyLarva plugins	102
7.3	Swaddler	104
7.4	Conclusion	105
8.	Conclusion	107
8.1	Future Work	109
8.2	Concluding Thoughts	109
	References	111

List of Figures

2.1	Architecture of a generic RV tool[16]	8
2.2	Separation of cross-cutting concerns with aspect-oriented Programming	10
2.3	polyLarva architecture	12
2.4	polyLarva local monitoring compiler featuring technology-specific plugins	13
2.5	Monitoring execution sequence for ‘ruleAddFailedLogin’ rule [16]	21
2.6	Sequence of events triggered by event ‘closeSession(Session sesn)’[16]	26
2.7	The requesting and execution process between the client, web server and PHP engine.[18]	28
4.1	Identical definitions of <i>add()</i> across scripts.	43
4.2	Dynamic inclusion of scripts in PHP WAs.	47
4.3	System-side monitoring code	57
5.1	Monitoring code compilation process mapping	62
5.2	Exemplary PHP monitoring code structure	63
5.3	Intermediate specification mapping from a property specification script	64
5.4	Intermediate specification: @CODECONTEXT block	67
5.5	Intermediate specification: @EVENTS block	68
5.6	Intermediate specification: main specification structure	68
5.7	Monitoring code structure	69
5.8	Instrumentation points in PHP scripts	70
5.9	Overall PHP plugin architecture	72
6.1	Moodle permission management	76
6.2	Descriptive field used for global user role	81
6.3	Assigned moodle role in study units	81
6.4	Permission violations for ‘mod/forum:deleteanypost’	82
6.5	Incorrect ‘Student’ permission	82
6.6	Student attempting access to ‘/moodle/admin/index.php’	85
6.7	Property violations indicating the attempted URL with parameters, and the User ID	85
6.8	Communication model for property#3	90

List of Tables

6.1	UoM VLE abridged permission specification	77
6.2	Type of events specified in the case-study properties	98

1. Introduction

Almost everything we do nowadays revolves in some way or another around the web - be it socially, recreationally or professionally. Over the years we have moved from a static and unstructured web to a more dynamic and structurally-conformist one. In particular, web applications (WA), which are of the latter sort, have changed the way we deal with information and are used in a variety of domains such as e-commerce, education, entertainment, etc. Moreover, mission, safety and business critical applications are increasingly moving to this paradigm over the web where they allow no room for failure as lives could be lost or financial losses could be incurred.

Due to their widespread use and criticality in certain domains, WAs are increasingly being expected to behave correctly with a high degree of confidence. However, there are some important differences between WAs and traditional software, adversely affecting the applicability of traditional correctness-checking techniques on WAs. One of the main differences is that they are by and large developed in dynamic languages, providing straightforward syntax to alleviate the developers effort through more conciseness and flexibility[17]. PHP¹ is a very popular dynamic language, mainly targeted for dynamic web page creation, that is relatively easy to work with when compared to static and other dynamic languages. Despite all its advantages, however, the verification of WA logic written in PHP is considered to

¹<http://php.net/>

be much tougher than that of software written in static languages. This is by virtue of the fact that the resultant PHP WAs are likely to be comprised of extremely loose and dynamically coupled components[1].

Traditionally one considers two main verification techniques: testing and formal verification. Testing covers a wide field of methods for finding bugs but it is never completely exhaustive and cannot guarantee the absence of bugs. Formal verification is another alternative to testing where a system is formally verified with respect to a set of properties. Model checking[2] is such an example; it attempts to ensure that all execution traces of a system will obey a specific set of properties. The problem with model checking is that it does not scale well to real-world systems as the number of execution paths for analysis grows exponentially, hence impractical to verify all. Since PHP is considered to be tougher to verify than other static languages, the formal verification of PHP WAs is even more unrealistic, forcing the necessary verification to be performed at runtime.

Runtime Verification (RV) is a verification technique that attempts to establish a trade-off between testing and formal verification, and is being pursued as a lightweight technique complementing both verification techniques[10, 15]. It has been shown to be a viable solution for the verification of large complex systems[5]. In contrast to model checking, RV, rather than considering all possible execution paths, only considers the current runtime path for analysis while still guaranteeing detection (albeit at runtime) of property violations. Currently there are no RV tools supporting PHP notwithstanding the popularity of the language.

Rather than designing and implementing a RV tool from scratch, we considered extending a RV tool called polyLarva[4, 5, 16], currently supporting the monitoring of systems developed in Java, C and Erlang. It is claimed to be easily extensible, but the effort required to extend it to some language, such as PHP, really depends on the nature of such language. When considering polyLarva for the runtime verification of PHP WAs certain factors must be taken into account such as the paradigm shift from static languages to dynamic languages. The programming unit of PHP

is a file, commonly called a script, which is the recurring element of PHP WAs. Scripts are the embodiment of sequential statements executed every time a script call is issued via an HTTP request. A script is also the place where functions and classes can be declared. Currently polyLarva has been extended only to static languages where the notion of scripts or files is missing.

In this report we present a PHP plugin as a support to polyLarva for its ability to monitor PHP WAs. To assess whether the resulting PHP plugin has been able to fully cater for the dynamic aspect of PHP, and the notion of scripts, we have applied it to a case study (Chapter 6) on an elearning system called Moodle, based on configurations from a real-life Moodle installation. The emphasis of this case-study was on the extent of properties that can be verified by polyLarva, by means of the PHP plugin, on PHP WA behaviour.

1.1 Report Structure

Chapter 2: Background This chapter gives an overview of the relevant parts in polyLarva and PHP. Further work in other chapters is based on such parts.

Chapter 3: Problem Definition This chapter gives the definition of the problem, for extending polyLarva to PHP, and states the main aim and objectives for a successful PHP plugin.

Chapter 4: PHP Plugin Design This chapter focusses on the design of the PHP plugin by investigating the relationship between the PHP language and prevalent PHP WA structures, and polyLarva.

Chapter 5: PHP Plugin Implementation Details This chapter gives the implementation details of the PHP plugin, highlighting discrepancies between the

generic polyLarva plugin structure and the requirements for a PHP plugin, and ways of overcoming them.

Chapter 6: Case Study This chapter gives details about a case study on a real-life PHP WA, aimed at evaluating the way in which the resulting plugin reaches the goals of this work.

Chapter 7: Related Work This chapter relates the resulting PHP plugin with other polyLarva plugins and Swaddler, an approach for anomaly detection whose concepts are similar to polyLarva, and which has been implemented for use with PHP WAs.

Chapter 8: Conclusion This chapter concludes by summarising the achievements of this work. It also gives the direction for future work and a recommendation for future plugins.

2. Background

2.1 Introduction

The main aim of this chapter is to give background on polyLarva and PHP, as an aid in understanding the polyLarva PHP plugin design process (Chapter 4), its implementation details (chapter 5), and the case-study (chapter 6).

2.2 polyLarva

polyLarva is a monitoring tool that is able to perform the monitoring of systems developed in any technology, based on theoretical foundations from an area of software verification called runtime verification. Monitoring is based on a formal behaviour specification that is handed over with the system to be monitored. Systems are enhanced with monitoring code, according to such specification, in order for them to be able to be monitored by a polyLarva monitor. polyLarva is able to perform such enhancements to systems developed in any technology, given the availability of an attributed technology-specific plugin.

In this section we start by giving some background on runtime verification (RV) (Subsection 2.1.1), with an aim to set a basis of understanding of the theoretical framework used by polyLarva. Then, we introduce a programming paradigm that is widely used by RV tools for structured instrumentation of monitoring code (Sub-

section 2.1.2), followed by an illustration of the polyLarva architecture (Subsection 2.1.3). Then, we illustrate the fundamentals of the polyLarva specification language through a running example, together with showing other monitoring aspects in relation to such example (Subsection 2.1.4). Finally, we give some comments on the way different monitoring parts interact with each other, focussing on aspects of communication between such parts. (Subsection 2.1.5).

2.2.1 Runtime Verification (RV)

Software verification includes all techniques suitable for showing whether a software system satisfies its specification. There are various verification techniques and methods, all varying in the way verification is applied to the candidate software. The main determinant factors in verification are scalability and exhaustiveness, usually determining the suitability of a verification technique for use with a particular class of software, such as safety-critical systems. Scalability is the extent of system complexity a verification technique can handle, and exhaustiveness is the extent of coverage a verification technique can provide. Scalability and exhaustiveness are factors orthogonal to each other which can be depicted on a tension scale placed on opposite ends.

Traditionally, one considers verification techniques such as theorem proving[8], model checking[2], and testing. Theorem proving is a technique that is applied manually in order to show whether a software is correct with respect to its specification. It is similar to how proofs are carried out based on a mathematical theorem. Model checking, on the other hand, is an automatic verification technique which bases itself on a formal model describing the system, and can be applied on finite-state systems. Both theorem proving and model checking are aimed at performing an exhaustive analysis on software. They are used to verify that for every possible execution path a software program can be in, a certain property always holds; such certainty is something which is highly desirable in verification. However, this has its limits, since their exhaustiveness highly impinges on the extent of their scala-

bility. In model checking various abstraction and reduction techniques have been proposed in order to alleviate scalability problems, however, the full verification of large-scale systems still remains largely unattainable with such technique. Testing is extensively used for the verification of large-scale systems due being scalable, however, it cannot give guarantees of whether a software is correct with respect to its specification since it is not exhaustive. Nowadays, software is becoming increasingly complex such that no real life system can ever be tested exhaustively.

Runtime verification (RV)[10, 15] is a verification technique that inherits qualities from both ends of the scale, between scalability and exhaustiveness; in fact it is seen as a lightweight verification technique complementing testing and model checking, establishing a trade-off between them. The main difference in RV is that it performs verification while the software is executing, or on an execution trace obtained from a software run. Its main focus is the execution trace (a single execution path), ignoring all other possible execution paths a program can exhibit. This greatly alleviates the problem of scalability while still being able to carry out analysis for the particular execution trace.

The essential components required for runtime verification, which should be considered over and above the target system, are a monitoring mechanism and a verification mechanism. The monitoring mechanism is used in order to elicit events in the system at the time they are occurring, and the verification mechanism is used for the verification of the succession of such events. Events are communicated to the verification mechanism in order to be verified in the sequence they are received, so it can be determined whether the execution trace adheres at all times to a given system specification. If the verification mechanism detects a violation, it should raise an alarm together with providing an indicator, pointing in the execution trace, where such violation happened.

When designing a RV tool, the focus is usually on two main parts: 1) the formal language in which correctness properties are specified, and 2) the translation of correctness properties into monitoring code, and its instrumentation into target

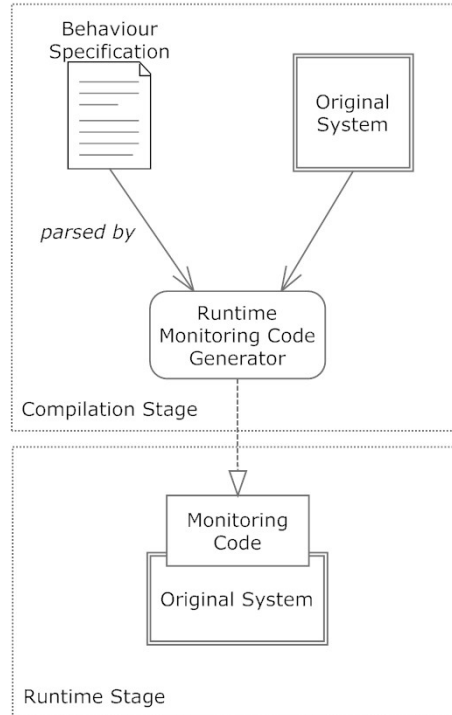


Figure 2.1: Architecture of a generic RV tool[16]

system code, in order to make the target system monitorable. In order for RV to be effective and as a verification technique, the formal notation used for the specification of correctness properties should be more abstract and concise than the implementing language of the equivalent monitoring code; otherwise, the probability of making an error in a correctness property specification would be the same as that when writing the monitoring code directly in implementing language.

The process of making a system monitorable generally requires two components: the target system for monitoring and a formal specification comprised of correctness properties on the system behaviour, illustrated in Figure 2.1. The two are given as input to a runtime monitoring code generator for the automatic generation of monitoring code and the eventual instrumentation of such monitoring code into the target system.

The generated monitoring code is directly translated from the supplied formal specification, and it is usually comprised of two main parts: the implementation

code for the monitoring logic and meta-code for the elicitation of events at code parts of interest in the target system. The instrumentation of monitoring code into the target system is performed in a way such that when the modified system executes, it does so in conjunction with the monitoring code, hence taking advantage of the new verification functionality.

A typical property in RV can be like the following: *the number of failed transactions in a system cannot exceed 500, among all system users*. When this property is compiled, from its formal representation, the implementation code of the monitoring logic will typically be comprised of a counter representing the number of failed transactions, together with attributed code for its incrementing with every failed transaction, and the checking of whether the counter has exceeded 500. If the counter exceeds 500 the verification mechanism should be able to raise a property violation, possibly triggering functionality to give feedback to the system. The event-elicitation code, on the other hand, will specify how the target system should be modified, so that whenever a transaction takes place, an event, with accompanying information about the success of the transaction, is generated and transferred to the verification mechanism for verification.

In the following section we give background on a programming paradigm, called aspect-oriented programming (AOP), that is widely used by RV implementations, such as in [3] and [16], as a technique for instrumenting monitoring code into the target system code.

2.2.2 Aspect-Oriented Programming (AOP)

Aspect-Oriented Programming (AOP)[14] is a programming paradigm aimed at complimenting other programming techniques, in particular object oriented programming (OOP). The main advantage of AOP is that it provides the developer with an abstraction layer over which he can implement cross-cutting concerns in separate modules. Figure 2.2 illustrates this by showing cross-cutting concerns tangled and scattered with core-level concerns in implementation modules on one

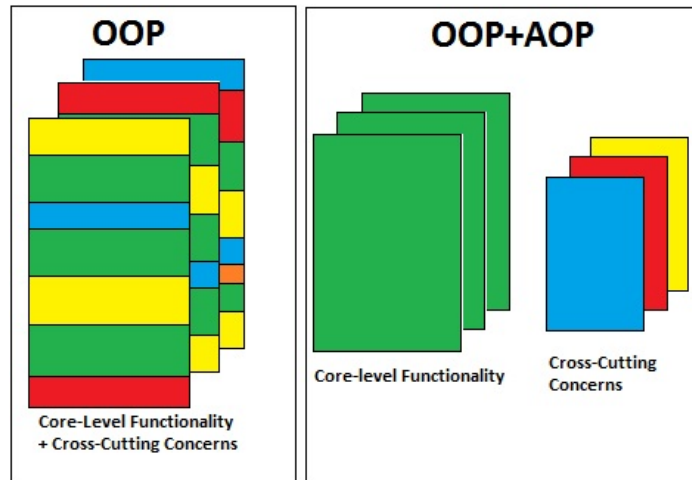


Figure 2.2: Separation of cross-cutting concerns with aspect-oriented Programming

side, and the same cross-cutting concerns and core concerns implemented in separate modules on the other side, through the use of AOP techniques. An example of a common cross-cutting concern that can benefit from AOP modularisation is logging. The only linking between logging functionality and business functionality is that of execution locality, such that when business functionality is called for execution, logging functionality is also called for execution. This causes logging functionality to become tangled and scattered across business functionality. With AOP techniques logging functionality can be implemented completely independently from business functionality.

In order for AOP code to be actuated, it has to be weaved into the original system code by an automated weaving process, either at compile time, load time or runtime¹. The weaving process injects the necessary triggers at selected points in the system code in order for extraneous code to be executed at such points, while the system is running. Such points, also known as *joinpoints*, are identifiable by the matching rules as provided by the particular AOP implementation. Matching rules can include the selection of joinpoints around particular method calls, exception throws and so forth. The code injected at a particular joinpoint, or at various joinpoints, is known as *advice*. An advice is executed whenever an attributed

¹<http://docs.spring.io/spring/docs/2.5.4/reference/aop.html>

joinpoint is encountered during system execution. An advice can be set to process any runtime information that is made available at the selected joinpoints. For example, if an advice is injected at a joinpoint after a method call, the runtime information that is made available to such advice can include the arguments passed to the particular method as well as the returned value.

Runtime monitoring can be viewed as a cross-cutting concern to the system being monitored. It is a concern that is usually invisible to the running system, cross cutting the target system for the capturing of an execution trace, for its verification. AOP techniques suit well for the identification of points at which runtime information of monitoring interest becomes available in the system code, for the instrumentation of monitoring code at such points. J.Cook et al. in [7] confirm the increased use of AOP in runtime monitoring as an aid for the instrumentation process.

2.2.3 polyLarva Architecture

polyLarva[5, 6, 16] is a technology-agnostic RV framework that be extended for the monitoring of systems developed in any programming language. Its architecture is based on the generic architecture of RV tools presented in Figure 2.1. The main differences lie in the separation of monitoring code compilers (generators). In addition to the compilation of monitoring code that is instrumented into the original system, a separate remote monitor is compiled as a program in its own right. Both the remote monitor and local monitoring code are compiled from the same polyLarva specification script. The remote monitor and local monitoring code communicate with each other over TCP sockets during monitoring. The main role of the remote monitor is to orchestrate monitoring and execute monitoring logic in response to the events it receives from the local monitoring code. The main role of the local monitoring code is to extract events from the monitored system and channel them to the remote monitor. It can also perform the evaluation of monitoring logic as part of the system, usually for system-bound information.

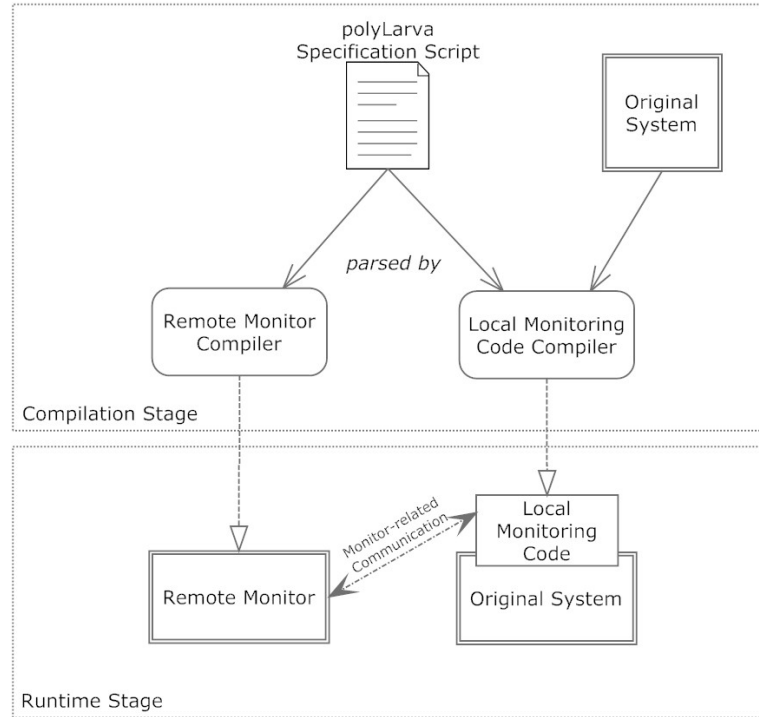


Figure 2.3: polyLarva architecture

The remote monitor is always compiled in Java, while the local monitoring code is compiled depending on the programming language of the system to be monitored. The local monitoring code is comprised of aspect code, for the extraction of event information, and other monitoring code. The aspect code is generated in the AOP language of choice, for the particular programming language, while the other monitoring code is generated directly in the particular programming language. The latter includes the necessary functionality for communicating monitoring information to and from the remote monitor, and the execution of any local monitoring logic.

Figure 2.4 gives more details on the way the local monitoring compiler generates local monitoring code for the different programming languages. A system developed in some programming language can only have its local monitoring code compiled by the local monitoring compiler only if a technology-specific plugin for such programming language exists. For example, if a system developed in Java is

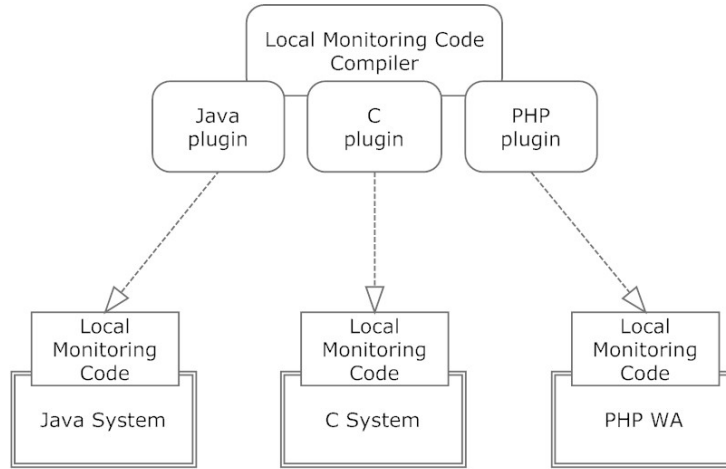


Figure 2.4: polyLarva local monitoring compiler featuring technology-specific plugins

to be monitored, a technology-specific plugin for Java needs to be available. The main role of technology-specific plugins is to compile the necessary monitoring code in the equivalent technologies, from the given polyLarva specification script, and instrument such monitoring code into the monitored system, becoming one with it.

2.2.4 Property Specification Language

The polyLarva property specification language is a technology-agnostic language, used for the specification of correctness properties on the behaviour of systems developed in any technology. It is best explained directly from its language definitions, however, for the sake of conciseness we are going to illustrate the most important parts via a running example. The main running example we are going to consider is from a dummy ATM system developed in Java. This ATM system allows for the logging in of one user at a time, by the entering of a pin number. If the pin number is a valid one, the user will be given the facility to carry out multiple actions against his account such as balance checks, deposits and withdrawals. Below are some properties (in a descriptive form) for the ATM pin authentication process.

- If the entered pin is not accepted by the ATM system, the user is given the opportunity to re-enter his pin number again. There cannot be more than three successive bad login attempts.
- If the entered pin number is not accepted by the ATM system, there should be a twenty seconds block on the authentication mechanism before one can proceed with another pin entry attempt.

The above properties must always hold during the time the ATM system is in operation. We will translate these into a polyLarva property specification, for illustration of both how to compose a general property specification and to show certain monitoring aspects, such as the communication sequence between the monitored system and the remote monitor for a certain property.

When planning to monitor a system, one should start by identifying which parts in the system code relate to the correctness properties considered for monitoring. This, so that the necessary events can be generated at such code parts. The above properties seem to be driven by one particular event happening at the system: that of pin number verification. In this example we find that the ATM system has a particular method called *verifyPin()*, used for the verification of the pin inserted by the user. Therefore, it is required that an event is generated after *verifyPin()* is called, so that with every such method call, the remote monitor receives an event and checks whether the attributed properties hold in the succession of such events.

Dummy ATM System Property Specification (pin validation part)

The polyLarva property specification language is a guarded-command style language, with properties expressed as a list of rules of the form[16]:

$$event \mid condition \rightarrow action$$

Whenever the remote monitor receives an event, the list of rules is traversed in

order to match any rules specified on such event. Rules are processed one after the other, first by matching the event, then by evaluating the condition, and if the condition has evaluated to true, by executing the action.

```
1 events {  
2     loginFail(result) = {*.verifyPin(*) uponReturning(Boolean result)}  
3 }
```

This code snippet illustrates an *events* declaration section, comprising one event. In this example all properties considered for monitoring are driven by this one event. There can be as many event declarations within an events section, as required. On the left hand side of an event declaration is the event name, together with captured runtime information that is made available by such an event. This particular event is declared with name *LoginFail*, and captures runtime information *result*, upon its occurrence. On the right hand side of the event is the event matching signature, **.verifyPin(*)*, specifically stating that *LoginFail* event should fire whenever a method in the system code, under the name *verifyPin()*, declared in any class, and taking any number of parameters, is called by the monitored system. *uponReturning(Boolean result)* intends that such event should specifically fire after *verifyPin()* returns a value, which in this case is of type Boolean. The declaration of the parameter type is important so as for the remote monitor to know what kind of runtime information to expect accompanying an event.

```
1 conditions {
2     monitorSide {
3         loginTimerInvalid = {larva:timerUnder(loginTimer , 20);}
4         loginAttemptsValid = {return loginAttempts < 3;}
5     }
6     systemSide {
7         isPinValid = {return result == false;}
8     }
9 }
```

This code snippet illustrates the conditions section, which holds the conditions (monitoring logic) of the considered properties. This section is further partitioned into another two sections. `polyLarva` allows for monitoring logic to be evaluated either at the monitor side or at the system side. Those conditions which are declared in the *monitorSide* section are compiled as part of the remote monitor, hence, are evaluated at the monitor side, and those conditions which are declared in the *systemSide* section are compiled as part of the local monitoring code, and are evaluated at the system side. Each condition is designated to return a true/false result, as determined by the monitoring logic. On the left hand side of a condition declaration is the condition name, while on the right hand side it is specified the monitoring logic for the evaluation of such condition.

In this example there are three conditions, two of them are evaluated at the monitor side, and another at the system side. Those evaluated at the monitor side are expressed in the Java language, since the runtime monitor compiles to Java code. *isPinValid*, in this example, is also expressed in the Java language, since the dummy ATM system is implemented in Java. If the monitored system was implemented in another programming language, the system-side condition would have been expressed in that particular language. *loginTimerInvalid* is a timer condition which returns true if declared timer *loginTimer* hasn't yet elapsed 20 seconds from the time it was started/reseted. It is used for the verification of whether the

authentication mechanism gets blocked for 20 seconds in between login attempts. *loginAttemptsValid* performs a value comparison to check whether the number of login attempts has exceeded the set threshold, using declared *loginAttempts* counter. *isPinValid* is a condition which determines the truth value of *result*, returned from *verifyPin()*. The aim of such condition is to determine the success of a login attempt.

```
1 actions {
2     systemSide {
3         logAttemptsErr = {System.out.println("No more login
4             re-attempts allowed after three failed tries");
5         }
6         logTimerErr = {System.out.println("A login re-attempt
7             is not allowed before 20s have elapsed from last
8             tryyy.");
9         }
10    }
11    monitorSide {
12        addFailedLogin = {++loginAttempts;
13        }
14        resetloginTimer = {larva:timerReset(loginTimer);
15        }
16    }
17 }
```

This code snippet illustrates the actions section. Just like conditions, actions can be declared to execute either at the monitor side or at the system side. Most of the time, actions are declared at the monitor side in order to update the monitoring state, such as in the form of a counter or a timer, but they can also be used for the altering of the monitored system state, by executing code at the system side. In this example there are four actions, two of which are declared for execution at the monitor side and the other two are declared for execution at the system side.

addFailedLogin is set to increment declared *loginAttempts* counter by value one, while *resetloginTimer* is set to reset the declared *loginTimer* to zero seconds. Both *logAttemptsErr* and *logTimerErr* are set to output a monitoring message at the ATM system, as they are executed as part of the monitored system.

```
1 timers {
2     loginTimer
3 }
```

This code snippet illustrates the timers declaration section, where separate timers (maintained by the remote monitor) can be declared for use with time-related properties. In this example there is the declaration of one timer with name *loginTimer*.

```
1 states {
2     monitorSide {
3         int loginAttempts {
4             saveWith {}
5             restoreWith {loginAttempts = 0;}
6         }
7     }
8     systemSide {
9     }
10 }
```

This code snippet illustrates the states declaration section, where states of any data structure/type can be declared. This states section, just like conditions and actions, is further partitioned into another two sections, depending on whether these are to be maintained at the monitor side or the system side. In this example, there is only one state declared with name *loginAttempts*, of type *int*, initialised with value 0, and set to be maintained at the monitor side. Such state is only made

accessible to conditions and actions set to execute at the monitor side. Likewise, a state declaration set to be maintained at the system side can only be made accessible to monitoring logic set to execute at the system side.

```
1 rules {
2     timeBlock = loginFail( Boolean result ) \ loginTimerInvalid ->
3         logTimerErr
4
5     ruleAddFailedLogin = loginFail( Boolean result ) \ (
6         loginAttemptsValid &&
7         isPinValid ) -> addFailedLogin , resetloginTimer ;
8
9     ruleTooManyFails = loginFail( Boolean result ) \ (!loginAttemptsValid
10        &&
11        isPinValid )-> logAttemptsErr ;
12 }
```

Rules are declared within the rules section. In this example, *timeBlock*, *ruleAddFailedLogin* and *ruleTooManyFails*, are rules that trigger whenever *loginFail* event is received from the monitored system. *timeBlock* is a rule that, when triggered, it evaluates condition *loginTimerInvalid* to check whether there has been a twenty seconds delay from the last login attempt. If it evaluates to true, action *logTimerErr* is executed, resulting in an error message at the system side. The other rules use the same two conditions, however, they are expressed logically different, yielding a different result. In rule *ruleAddFailedLogin*, if both conditions *loginAttemptsValid* and *isPinValid* evaluate to true, both actions *addFailedLogin* and *resetloginTimer* are executed, with the result of incrementing *loginAttempts* counter and resetting *loginTimer* back to zero seconds respectively. In rule *ruleTooManyFails*, if condition *loginAttemptsValid* evaluates to false and *ruleTooManyFails* evaluates to true, action *logAttemptsErr* is executed, outputting an error message at the ATM system.

```
1 global{
2     timers{ ... }
3     states{ ... }
4     events{ ...}
5     conditions{ ... }
6     actions{ ... }
7     rules{ ... }
8 }
```

In order for all the described sections to work together, they should be placed within a *global* section, the outermost encapsulation of a property specification script, denoting a global monitoring context.

System-side Evaluation for Pin Verification in ‘ruleAddFailedLogin’

In the processing of rule *ruleAddFailedLogin* the monitor is required to verify whether the return value of *verifyPin()* indicates an invalid login, since the outcome of such rule is partly determined by such value. Return value *result* is a variable bound to the system (determined by the variable type), and therefore the remote monitor needs to query the system in order to obtain an evaluation of the truth value of such variable. Figure 2.5 demonstrates, through the use of a sequence diagram, how the remote monitor communicates with the system for the evaluation of such variable.

In Figure 2.5, an event is generated and communicated to the remote monitor with every call to *verifyPin()*, through the aspect code that is weaved into the system. Upon its receipt, the remote monitor proceeds by matching any rules specified on such event. *ruleAddFailedLogin* is one such rule, where at some point in its processing it requires to evaluate whether the result of the pin verification is valid or not. This is seen in the evaluation request the remote monitor makes to the monitored system for the execution of condition *isPinValid()*. *isPinValid()*

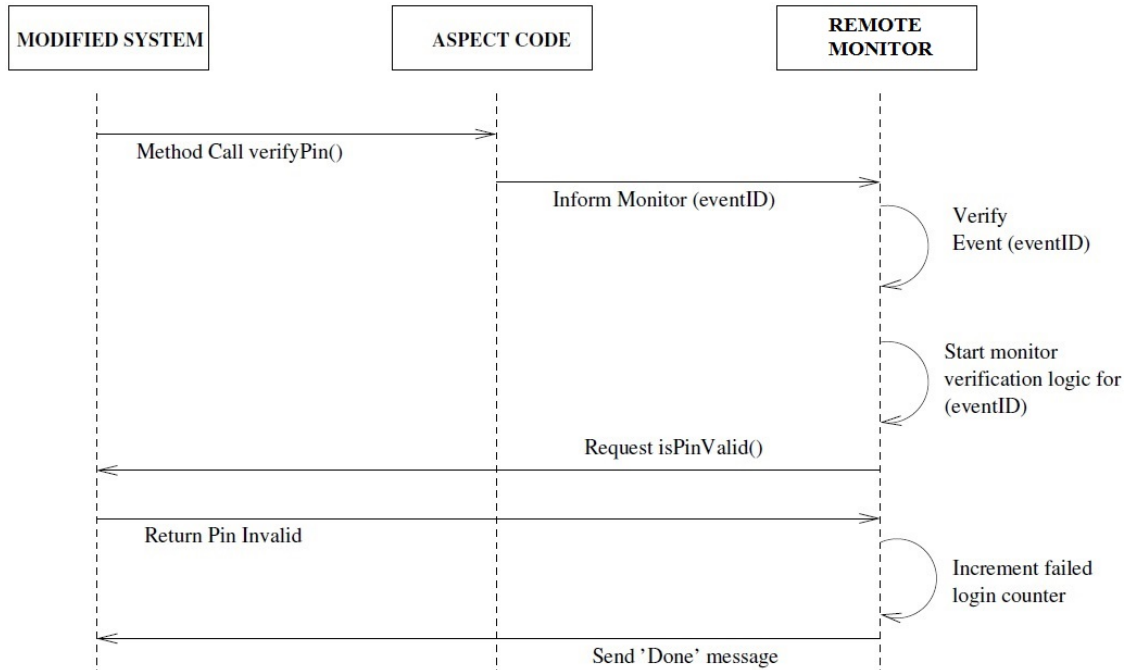


Figure 2.5: Monitoring execution sequence for ‘ruleAddFailedLogin’ rule [16]

forms part of the monitoring code generated by the local monitoring code compiler, through the Java plugin. Once such condition is executed, an evaluation result true/false is channelled back to the remote monitor, after which the processing of *ruleAddFailedLogin* can resume for the incrementation of the number of bad logins, if the entered pin was invalid.

Nesting of Monitors for Contextual Properties

The process of pin validation, if successful, leads to the creation of a new session for the user logging into the ATM. If we consider properties on functionality executed from within a particular session, it is required that a monitoring state be maintained for each session. An example of this is the maintenance of separate counters on the number of failed transactions that occur from each session. In such a scenario, it is required that properties on contextual behaviour be replicated for the different contexts, in this case for each session. The polyLarva specification language supports a kind of properties termed contextual properties[3, 16], provid-

ing special language constructs which allow for the replication of monitors, each representing a different monitoring context. Monitoring contexts are defined hierarchically, with the global context defined as the outermost encapsulation, and child contexts defined within it, and where these can also have their children, and so forth.

All monitoring contexts, irrespective of their position in the hierarchy of contexts, have an identical structure, with events, timers, states, conditions, actions and rules of their own. The best analogy that can be drawn for a better visualisation of monitoring contexts is from the object-oriented paradigm, where each context is compared to a class, and each monitor compared to an object instantiated from such a class. An object can be composed of other objects, and so forth, and where a parent object holds the references to its child objects. The same is with multi-level contexts.

```
1 global {
2     ...
3     events{
4         ...
5         newSession(ATMSession session) = { *.doLogin(*) uponReturning (
6             session)}
7     }
8     ...
9     rules {
10        ruleAddFailedLogin = { ... }
11        ruleTooManyFails = { ... }
12        upon {
13            ruleNewSession = newSession(ATMSession session) \
14                !loginTimerInvalid ->
15                resetloginTimer {
16                    saveWith { }
17                    restoreWith{ }
18                }
19        }
20    }
21 }
```

```
18     }
19     load session {
20         ...
21         events {
22             addTransaction(trans, sesn) = {ATMSession
23                 sesn.doTransaction()
24                 uponReturning(Transaction trans)}
25             where {session = sesn;}
26         }
27         ...
28         rules {
29             ...
30         }
31     }
32 }
33 }
```

This is an abridged property specification set to illustrate the nesting of monitors. The outermost clause denotes a global context. For the purpose of illustration, *newSession* is an event declaration in the such context, used by rule *ruleNewSession* for the creation of new contexts based on newly created sessions, made available through object *session*. Every time the remote monitor is notified of a *newSession* event, *ruleNewSession*, declared in an *upon* clause, is activated and executed. Since this rule does not have a condition, it directly proceeds for the execution of the action. After the action is executed, it then proceeds to the *load* clause, where the sub-context is specified, for its initialisation. The sub-monitor is similar to the global monitor with the difference that it has a monitoring handle through which it can be identified. In this case, a sub-monitor is created for every session, therefore, the handle used for each sub-monitor is the *session* object, made available through event *newSession*.

Within the session sub-context, lies an event declaration *addTransaction*, with *sesn* being the session object from which the transaction is performed. At the time

the remote monitor receives such an event, it would not have yet determined which sub-monitor should be loaded for the particular session. It is eventually determined by the *where* clause, *where session = sesn;*, where the existing sub-monitor session handles are matched against the incoming runtime information *sesn*. Following sub-monitor matching, the particular sub-monitor is loaded in order to have the rules matched with the incoming event processed in relation to the monitor, comprised of counters, timers, etc.

Internal Events

So far, we have discussed properties with rules defined on events generated by the local monitoring code. polyLarva supports another category of events, called internal events. Typical scenarios for the need of internal events is when there is the need that certain events fire as a result of some other rule resolving to true, or some monitor timer reaching a certain count.

```
1 global{
2   ...
3   ...
4   rules{
5     upon{
6       ...
7     }
8     load session{
9       ...
10      ...
11     rules {
12       ...
13       ruleCloseSession = closeSession(Session sesn) ->
14         larva: fire (checkContents(cart));
15     upon {
16       createNewShoppingCart = getNewShoppingCart(shoppingCart)
17       \ shoppingCartDoNotExist -> logNewShoppingCart
```



```
18         }
19     load shoppingCart{
20         ...
21         events{
22             checkContents(String cart) = {?checkContents(cart)}
23             where {shoppingCart = cart;}
24         }
25         ...
26     }
27 }
28 }
29 }
30 }
```

This is an abridged property specification from an e-commerce system scenario. It is primarily intended for the illustration of how internal events work. It is clear that there are two levels of sub-monitors nested within each other, the first being for the different sessions that are created, and the second for shopping carts within sessions. Such a monitoring setup represents properties monitored per session per shopping cart.

This property specification script ensures that whenever a session is closed, the system empties the shopping cart for the particular session. Rule *ruleCloseSession*, triggered for processing by event *closeSession*, immediately proceeds to execute the action part. The action is comprised of directive *larva:fire()*, used for the firing of internal events. For this to work, the event name supplied to the directive must match with at least one declared event in any of the specified contexts. In this example, the event name referred to is *checkContents(cart)*, which is also declared as an internal event in the *shoppingCart* context. The rightmost part of an internal event declaration varies from the standard way of declaring events, as it includes the same event name prefixed with ‘?’. This ensures that whenever *ruleCloseSession* is processed by a session sub-monitor, event *checkContents* is fired in the context of the attributed *shoppingCart* sub-monitor for the processing of rules that check

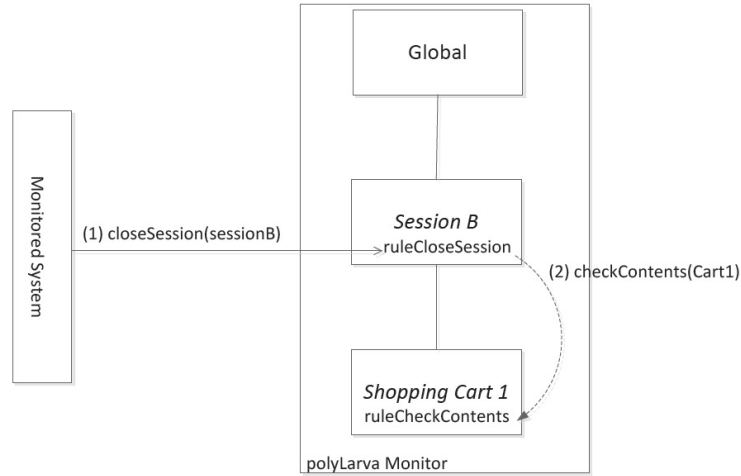


Figure 2.6: Sequence of events triggered by event ‘closeSession(Session sesn)’[16]

whether a shopping cart has been emptied.

Figure 2.6 illustrates the sequence of how the closing of a session (informed by an external event) triggers the checking of whether a shopping cart has been emptied (informed by an internal event).

2.2.5 Communication between the Remote Monitor and System-side Monitoring Code

The communication between the remote monitor and the system-side/local monitoring code is carried out over TCP Sockets, in a message-based, request-response style protocol.

A new TCP socket connection is initiated every time the system-side monitoring code communicates a generated event with the remote monitor. For this reason, the remote monitor is set up to listen for connection requests that are issued by the system-side monitoring code. Following an event, the remote monitor can start a trail of to-and-fro messages on the same established connection, depending on the the system-side functionality that needs to be evaluated, until all the rules, at the monitor side, are processed. Following that, the remote monitor concludes by sending message ‘Done.’ to the system-side monitoring code.

There can be monitoring situations where an event occurs outside the bounds of the monitored system, such as due to an internal event, or due to an event generated by another monitored system. Such events can trigger rules which require that the evaluation of some system-side functionality be carried out at the monitored system. In such circumstances, the remote monitor will not have an already established connection upon which to send evaluation requests to the system-side monitor. Therefore, the remote monitor proceeds by issuing a connection request itself to the system-side monitoring code, requiring that the system-side monitor be also set up to listen for connection requests. Following the establishment of a TCP socket connection, the monitoring sequence is similar to that following an event.

2.2.6 Conclusion

Runtime verification (RV) has evolved as a compromise to the scalability problem of verification techniques such as model checking, and the exhaustiveness problem of testing. It is a verification technique that has given rise to a number RV implementations, for the runtime verification of systems against a formal property specification. The static process of a typical RV tool is that of first generating a runtime monitoring code from a given formal property specification, then instrumenting such code into the target system. A popular way of instrumenting such monitors in RV is by the use of aspect-oriented programming (AOP) techniques.

polyLarva is an extensible, technology-agnostic RV tool, allowing for the specification of properties on the behaviour of systems developed in any programming language, given the availability of an attributed technology-specific plugin. Its language also allows for properties to be replicated for different monitoring contexts in a system. The static process of polyLarva involves the compilation of a separate remote monitor, common to all technologies, and local monitoring code, through an attributed technology-specific plugin. The latter code is eventually instrumented into the target system code using AOP. This separation allows for a clear sepa-

ration between logical monitoring operations and technology-specific monitoring operations. It preserves much of the work involving logical operations across the various supported technologies, requiring technology-specific plugins to cater only for technology-specific operations.

2.3 PHP

In the past, the web was comprised only of static services, where HTML web pages were simply requested and delivered exactly as were coded. HTML alone would not have produced much on its own, and as a result, scripting languages were born, and continued to grow more popular with an unrelenting drive for dynamic content creation. Scripting languages differ from compiled languages in that they are interpreted by an executable program instead of becoming an executable program themselves; they cannot work on their own. Scripting languages fall into two main categories: client-side and server-side. The main difference between the two is execution locality. For example, JavaScript² is a very popular client-side language, used for the creation of dynamic elements in HTML, and is executed by the browser on the client's computer; PHP[12] is a very popular server-side language, mostly used as a logical layer between the user and various other server applications, such as databases and identity management. Users are not given access to PHP code.

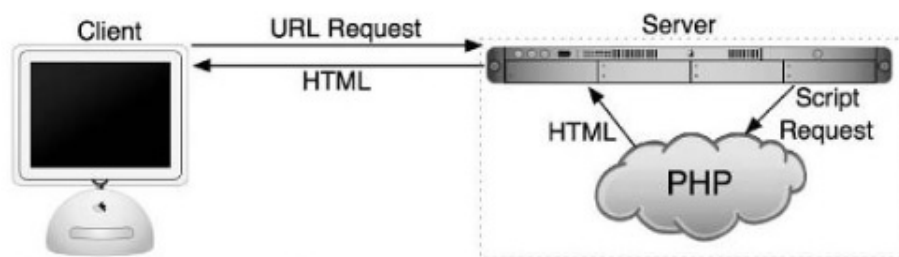


Figure 2.7: The requesting and execution process between the client, web server and PHP engine.[18]

PHP has been used for the development of various popular web applications

²<http://www.w3schools.com/js/>

such as Moodle, Wordpress and MediaWiki. It has the ability to process any information as well as interact with other system components such as databases, files, and email on the server/network. According to the official PHP website³ it is described as an HTML embedded scripting language, since it can be coded together with HTML code in the same file. PHP code is distinguished from HTML code by the enclosing PHP opening and closing tags. When a web page comprising PHP code is requested from a browser, the web server delegates all the code, residing within these PHP special tags, to the PHP engine for processing, and delivers to the requester only the HTML output resulting from such processing, as illustrated in Figure 2.7. Other content residing outside these special tags is delivered directly to the requester exactly as coded in the file. Once a PHP engine is installed and configured correctly on a server (usually the same as that for the web server), running a PHP script becomes as simple as placing a PHP file into the web server public directory (the same used for HTML files), and navigate to the file via a URL through a browser. By default, any file that contains PHP code must end with extension ‘.php’.

2.3.1 PHP Language

PHP code is placed in blocks called *code islands*. A conventional way of opening and closing a code island is by the use of `<?php` and `?>`. There can be as many code islands as one wants in a script. Tags are necessary for the PHP interpreter to switch into PHP parsing mode, in order to interpret and execute the PHP code. The basic unit in PHP code is a statement. Statements are separated from each other by a semicolon, and are conventionally placed on different lines, for better visibility.

³<http://php.net>

```
1 <html>
2   This is my first </br>
3   <?php
4       echo "hello , world" ;
5   ?>
6   This is my second </br>
7   <?php
8       echo "hello , world" ;
9   ?>
10 </html>
```

The code above represents a PHP script ‘somefilename.php’. The code starts with regular HTML, but interspersed in it are two PHP code islands. Its output looks like the following when requested from a browser:

```
This is my first
hello, world
This is my second
hello, world
```

Variables in PHP are distinguished from text or other statements by a dollar sign prefixing their name. In PHP, variables are generally declared typelessly, where everything can be assigned to them without a prior declaration of their type. Their type is determined at runtime according to the data elements they are assigned. A variable type can resolve to any one type such as string, integer, floating-point, array, object (complex data with functionality), or an information resource, such as an image. Moreover, types can change during the script execution upon the assignment of some other value with a different type. For example, if a variable is first assigned an integer and then a string, it will first resolve to a type integer, then to type string. One main disadvantage of PHP variables is their very short life span. They are disposed of upon the script execution reaches the PHP closing

tag.

```
1 <?php
2     function printMyName($name_, $iteration_){
3         echo " my name is ".$name_." in iteration ".$iteration_."<br>";
4     }
5
6     $x = 0;
7     while ($x < 4){
8         if ($x == 0){
9             $name = "Jonathan";
10        } else if ($x == 2) {
11            $name = "Mario";
12        }
13        printMyName($name, $x);
14        $x++;
15    }
16 ?>
```

This PHP code starts by the declaration of a function, with two typeless arguments. The actual code execution starts with the variable declaration of '\$x' and its value assignment in line 6. It is followed by a while statement with two subsumed if-then statements; both the while and the if-then statements operate in the same way as in other static programming languages. Variable '\$name' is first declared and assigned a value from within the first conditional construct. It is referred to again from within the second conditional construct and eventually from outside the conditional statements. In most programming languages such out-of-scope referencing is not permitted, as a declared variable usually becomes available for use only from within the scope it is declared in, and any of the child scopes. In PHP, this flexibility is only allowed in between the running code sequence; it is not allowed in between encapsulating constructs, such as objects and functions.

The latest PHP versions have been enhanced in order to incorporate a full object model, with OO constructs similar to those in Java and C#. However, despite

the provision of such constructs, PHP web applications kept abiding to a kind of structure where the primary functional unit remained the script (representing some page and its related functionality). Although OO is not so much popular in PHP, we are still going to give it an exposure, since it is going to be referred to in the development process of the PHP plugin.

```
1 <?php
2     class Process{
3         public $User = null;
4         public function setVar($User_){
5             $this->User = $User_;
6         }
7         Private function checkUser(){
8             return $this->User == "Jonathan";
9         }
10        Public static function checkResult($User){
11            $process = new Process();
12            $process->setVar($User);
13            if($process->checkUser()){
14                echo "User ok";
15            } else{
16                echo "User not ok";
17            }
18        }
19    }
20
21    Process::checkResult("Jonathan");
22 ?>
```

The above code starts with the declaration of a class, and the various elements within it. Data elements are declared typelessly, and are given an access modifier. Class/object functions operate much in the same way as normal functions, with the difference that they can be declared with an access modifier, and can operate

statically within a class. An object instance of a class refers to its data elements using the *\$this* keyword; *\$this* refers to the current object instance of a class. The execution of the script starts from static function call to *checkResult()* of class *Process* at line 21. A static function call is explicitly made by a double colon following the class name. Static function *checkResult()*, in turn, creates an instance of class *Process* into object variable *\$process*. Following object creation, object function *setVar()* is called from instance *\$process*; an object function call is explicitly made by an arrow following the object instance. Finally, another object function is called, returning a boolean value. *checkResult()* terminates by evaluating such value and outputs either "User ok" or "User not ok" accordingly.

Script Inclusion and Superglobal Variables

PHP web applications are comprised of various scripts, grouped according to their main functional purpose, and placed in a hierarchy of directories accordingly. As already mentioned, the programming unit of PHP WA is a script, and scripts are included into other scripts in order to achieve more complex script functionality. Scripts, therefore, can execute either in response to a direct request or by their inclusion into other scripts.

In PHP, the statements that can perform script inclusion are mainly *require(-<scriptname>)* and *require_once(<scriptname>)*. When either of these execute in a PHP script, the specified script executes as if its entire code had been interleaved into the main script. The latter instruction prevents that the same script is included more than once in the executing script thread.

PHP reserves a set of variables called superglobals. These are special purpose variables which can be referenced to from anywhere within the script code. Each of these represent a specific aspect relating to the execution of a script, such as for cookie information or for information about the script requester. Below are some of the most useful superglobals:

\$_SERVER - carries information such as the requestor IP address, executing script name and request time

\$_GET/\$_POST - carry information from a submitted HTML form

\$_COOKIE - makes available loaded cookie information

\$_SESSION - used for the preservation of session information across subsequent script requests, in order to maintain the contexts in which users operate.

2.3.2 Conclusion

In this section, we have given an overview of the core PHP language elements, with enough background to understand the main working elements of PHP WAs and the design decisions contributing to the PHP plugin.

2.4 Conclusion

In this chapter we have introduced polyLarva as an extensible RV tool with a technology-agnostic language for the specification of properties for systems developed in any technology, and PHP as a flexible and dynamic scripting language. These two are the main components upon which investigations for the extension of polyLarva to PHP will be based. In the following chapter we give the problem definition followed by the main aim and objectives for a successful PHP plugin.

3. Problem Definition

3.1 Introduction

Extending polyLarva with a PHP plugin requires that we first take an in-depth look into the PHP language and how PHP WAs are generally structured. This, since monitoring a PHP WA requires a two-step pre-monitoring process of 1) compiling PHP monitoring code from a given polyLarva specification and 2) instrumenting such code into the target PHP WA code, making the PHP WA monitorable. The way monitoring code is generated and instrumented into target systems is different for different technologies, with such differences catered for by the different technology-specific plugins.

In total there are three other polyLarva plugins: for Java, C and Erlang. All of these plugins totally rely on aspect-oriented programming (AOP) techniques for the instrumentation of monitoring code into the target systems. They generate two sets of code: AOP code for the chosen AOP implementation for the technology they support, and additional monitoring code in the programming language in question. AOP has been shown to be sufficient as an instrumentation technique for the kind of properties expressed on systems developed in these languages. AOP is mainly used for the instrumentation of event extraction code at specified points in the system code. Programs developed in these languages are usually well structured and exhibit runtime information of monitoring interest around well defined struc-

tures i.e. methods and functions; for these kind of programs, most of the runtime information of monitoring interest can be captured by AOP.

The problem with PHP WAs is that sometimes they lack in structure, with the consequence that important runtime information gets blended into unstructured sequences of executing code, which cannot be captured by AOP. This suggests that in addition to the identification of an AOP implementation for PHP, other means for instrumentation have to be found for full instrumentation of PHP monitoring code into PHP WA code.

3.2 Aims and Objectives

The overall aim of this project is to extend polyLarva with a PHP plugin that is adequate enough for the monitoring of any PHP WA. The PHP plugin should cater for various scenarios of PHP monitoring, from the monitoring of individual PHP scripts to systems comprised of PHP and other technologically-different components. A PHP plugin should serve as an intermediary between polyLarva and PHP WAs, for the runtime verification of properties expressed on PHP WA behaviour. More specifically, the PHP plugin should be able to modify PHP WA script code, according to a given polyLarva specification script, in order to enable a PHP WA for the required monitoring. Modifications to PHP script code include the generation of events at points of interest in code, communication to and from the remote monitor, and support for local monitoring functionality. The objectives to accomplish these are as follows:

Event generation: The PHP plugin should provide a considerable choice of points in PHP script code at which monitoring events can be extracted, for ability to capture the maximum runtime information of monitoring interest from any executing PHP code. The diversification of code structures (such as method/function calls or code structures) around which such points are made selectable must be sufficient enough as to the kind of properties specified on

PHP WA behaviour. Properties can be defined on event declarations pointing at various points in script code, such as before/after regular or class function definitions within PHP scripts, or at specific points in sequences of PHP code. The PHP plugin should be able to automatically generate event extraction code from event declarations in polyLarva specification script and instrument such code at selected points in the PHP WA code.

Communication to and from the remote monitor: The PHP plugin should generate the necessary communication functionality (as part of the system-side monitoring code) in order for monitoring messages to be communicated to and from the remote monitor. More specifically, the communication functionality should cater for: 1) the initiation of a TCP socket connection with the remote monitor, performed every time an event is extracted from the monitored system; 2) the maintaining of a TCP socket connection with the remote monitor, and bidirectional communication over an established connection using the polyLarva communication message definitions[16]; 3) the listening for connection requests issued by the remote monitor, performed every time the remote monitor requests for some local monitoring evaluation without an already established connection.

Support for local monitoring functionality: Local monitoring functionality includes system-side property conditions and actions, together with attributed system-side monitoring states[16]. The PHP plugin should be able to extract system-side monitoring functionality (readily expressed in PHP), from a given polyLarva specification script, and integrate them as part of the local monitoring code. Such functionality should be compiled in a way that when requested to carry out some representative evaluation, it is given access to functional elements and variables made available in the execution script thread of the triggering PHP WA script. The PHP plugin should also generate the necessary functionality (as part of the system-side monitoring code)

for the preservation of different monitoring states at the monitored system.

3.3 Conclusion

In this chapter we have given an overview of what makes a technology-specific plugin, together with highlighting the general concerns with extending polyLarva to a PHP plugin. Then we have stated the main aim for the project and the specific objectives for a successful PHP plugin. These objectives are evaluated in a case-study, presented in Chapter 6.

4. PHP Plugin Design

4.1 Introduction

The monitoring of any system by polyLarva is performed by means of interacting monitors. The first kind of monitor, called the remote monitor, is implemented separately from the monitored system, and is mainly responsible for monitoring orchestration and the verification of monitoring information. The other kind of monitor is weaved into the monitored system, and is mainly responsible for the extraction of monitoring events from the monitored system, the communication of such events with the remote monitor and the verification of system-bound monitoring information. The monitoring of PHP WAs requires that a monitor, of the latter kind, be designed specifically for the PHP language in order for it to be able to peek into the PHP WA state and perform the related tasks.

In this chapter, we start by PHP and other programming languages with an existing plugin in the context of polyLarva (Section 4.2), and apply any necessary core design changes to cater for the identified language differences, particularly with regards to event extraction (Section 4.3). Then, we look into the translation of event declarations into implementable code, and the instrumentation of such code into PHP WAs (Section 4.4), and conclude by giving a holistic view of the system-side monitoring code and describe how local monitoring evaluations are performed in attributed monitoring contexts (Section 4.5).

4.2 Challenges for polyLarva in Monitoring PHP WAs

We approach design with reusability in mind, hence, in this section, we are drawing a contrast between PHP and programming languages with an existing polyLarva plugin. This, in order to bring out the main differences of PHP from such languages in the context of polyLarva. At the time we were designing the PHP plugin, there were two existing plugins, one for Java and one for C.

Java and C programs are generally developed with an attention to code structuring and modularity. In fact, Java programs are predominantly modularised in terms of classes and methods, and C programs in terms of modules and functions. PHP is not much different from these in the way its code can be modularised, as it supports the use of classes, methods, functions and so forth. Despite the availability of such constructs, however, PHP WAs are still largely modularised in terms of loosely bound scripts, organised into file system directories according to their main functional purpose. The net effect of this is mostly felt in the resultant script code. There is the tendency that scripts become coded with long sequences of statements for the carrying out of specific tasks, exhibiting little to no structure in their code. Moodle is a popular PHP WA which we used as our case-study (Chapter 6). It has a significant amount of such scripts, with some of them having over 500 lines of code, comprised of sequential if-then and loop statements.

When it comes to monitoring with polyLarva, the way a program is structured is of utmost importance, since probing into the code of a program, for the extraction of monitoring events, is purely guided by the program structure. The fact that PHP WAs are usually comprised of ill-structured code, it makes their monitoring with polyLarva more challenging.

Another main difference lies in the way functional elements, such as classes and functions, are handled in PHP scripts. In Java and C, classes, methods and functions are uniquely defined as part of one consolidated program, and can be

referenced to without ambiguity from anywhere within such program, given they are visible. In PHP, classes and functions are uniquely defined, but only within a script. This leads to the possibility of having identical definitions of classes and functions across the various scripts comprised in a PHP WA, which might be unrelated to each other functionally. Their uniqueness is only enforced in the thread of an executing script request.

Event declarations, in polyLarva, are based on a two-level event matching signature, specified in terms of object/module and method/function names. This, in order to be able to target a particular code structure in the system code for the instrumentation of event extraction around such structure (before or after, depending on other specifics of the event declaration). Consider the following event declaration:

```
1 newCustomer(long custId) = {*.add(custId ,*)}
```

The event matching signature of this event declaration specifies that a monitoring event should be extracted from the system every time method/function *add()* is called. If the same event matching signature is applied to PHP WAs, it can easily lead to the instrumentation of event extraction code at unwanted places. For example, in Moodle, function *add()* is defined in 29 different scripts, where some functionally different from others, and where such function is called from 5035 places across all Moodle scripts.¹

4.3 Enhancing the Property Specification Language for the PHP Language

In view of the highlighted PHP language differences and the monitoring issues discussed in the previous section, the particular area of concern that needs addressing

¹<http://www.sourcexref.com/xref/moodle/nav.html>

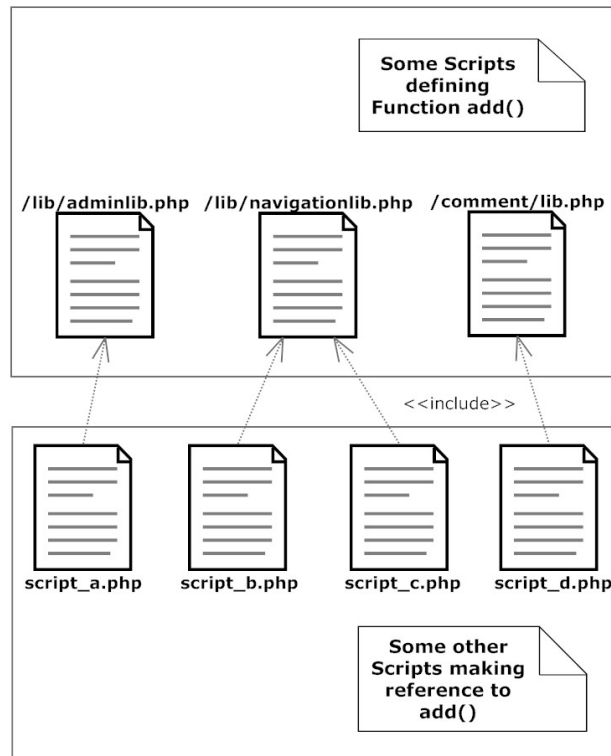
is the extraction of monitoring events from PHP WA. We are therefore looking into the event declaration syntax, part of the property specification language, for the identification of any limitations impinging on the ability to specify events for PHP WA. For every identified limitation we will present an analysis, based on the PHP language properties and common PHP WAs structures, and conclude each with a syntactic enhancement for the catering of such limitation. In this section we are particularly looking into the following PHP monitoring concerns:

1. The inability to distinguish between identical method/function definitions across script code for event extraction around specific methods/functions.
2. The inability to probe into unstructured script code for event extraction at points other than around methods/functions.

4.3.1 Applying Event Declarations to Subsets of PHP WA Scripts

Since PHP does not force uniqueness in class and function definitions across scripts, the current event matching signature syntax turns out to be not expressive enough to distinguish between identically defined code structures in PHP code. Consider the scenario in Figure 4.1:

Figure 4.1 illustrates a typical scenario where a function is defined in more than one script. The scripts defining such function are included into other scripts for use of such function in their code. As already indicated, the current event matching signature only allows for the specification of a class/module and a method/function name. A specification on function *add()*, in this scenario, will result in extraction of events at every script calling a function with name *add*. The directory path of the scripts defining *add()* indicates the functional purpose of such scripts. For example, `‘/comment/lib.php’` suggests that it is totally different, functionally, from the other scripts. It is therefore desirable to have an ability to distinguish between the

Figure 4.1: Identical definitions of *add()* across scripts.

different directories or scripts, for the applicability of an event matching signature only to particular directories or scripts.

With having the possibility of specifying a path, in addition to an event matching signature, one can specify the target script/s to which of an event matching signature should be applied. Consider the various narrowing-down possibilities expressed by following event declarations:

```

1 event1 () = { /* *.add () }
2 event2 () = { /lib/* *.add () }
3 event3 () = { /lib/** *.add () }
4 event4 () = { /lib/navigationlib *.add () }
5 event5 () = { /lib/script_b *.add () }

```

All event matching signatures presented above are prefixed by a path, either to a directory or to a script. In line 1, the event matching signature is equivalent to not

having a specified path at all, as it includes all scripts in the WA scope. In line 2, the event matching signature specifies a directory, meaning that it should only be applied to scripts in directory ‘/lib’. In Figure 4.1, in directory ‘/lib’ there are two scripts defining *add()*. The net result by this event declaration is that all the three scripts including such scripts will generate an event upon calling *add()*. In line 3 the main difference from line 2 is that the event matching signature is applied not only to first level scripts, but also to scripts in sub-directories within the specified directory. In line 4, the event matching signature specifies a script that is included by two scripts. The net result of this is that these two scripts making use of *add()* will generate events upon calling *add()*. Finally, in line 5, the event matching signature specifies a script which does not directly define *add()*, but rather make reference to it in its code. Such event declaration is applied only to such script, for the generation of events upon calling *add()*, defined in ‘/lib/navigationlib.php’.

Below is the BNF of an enhanced event matching signature, defined as <Script-PrimEvent>, composed of a script path variation in addition to the original event matching signature <PrimitiveEvent>, defined in [16].

1	<Path>	::=	<DirectoryPath>’/’<ScriptNameWithoutExtension>
			<DirectoryPath>’/*’ <DirectoryPath>’/**’
2	<ScriptPrimEvent>	::=	<Path>’ ’<PrimitiveEvent>

4.3.2 The Extraction of Monitoring Events from Unstructured PHP Code

In PHP WAs it is very common to find task-oriented scripts that are coded in a procedural fashion, for the carrying out of a specific task. Consider the following Moodle script:

1	<?php
---	-------

```
2 // block.php – allows admin to edit all local configuration variables
   for a block
3     require_once(' ../ config.php ');
4     require_once($CFG->libdir . '/ adminlib.php ');
5     $blockid = required_param(' block ', PARAM_INT);
6     if (!$blockrecord = blocks_get_record($blockid)) {
7         print_error(' blockdoesnotexist ', ' error ');
8     }
9     admin_externalpage_setup(' blocksetting '. $blockrecord->name);
10    $block = block_instance($blockrecord->name);
11    ...
12    /// If data submitted, then process and store.
13    if ($config = data_submitted()) {
14        if (!confirm_sesskey()) {
15            print_error(' confirmsesskeybad ', ' error ');
16        }
17        ...
18        redirect("$CFG->wwwroot/$CFG->admin/blocks.php", get_string("
changessaved"), 1);
19        exit;
20    }
21    /// Otherwise print the form.
22    $strmanageblocks = get_string(' manageblocks ');
23    $strblockname = $block->get_title();
24    ...
25    echo '<form method=" post " action=" block.php ">';
26    echo '<p>';
27    foreach($hiddendata as $name => $val) {
28        echo '<input type=" hidden " name=" ' . $name . ' " value=" ' . $val .
' " />';
29    }
30    ...
31    echo $OUTPUT->footer();
```

This script is not a very lengthy one (around 50 lines of code), compared to other Moodle scripts, but it is still considered to be unstructured as it is set to execute a sequence of instructions from beginning to end. The two factors that usually determine the place where event extraction code should be inserted in the system code are: execution locality, for the extraction of events upon executing a certain code, and the runtime information of monitoring interest that is made available by such execution code. For example, if one wants to extract an event for the capturing of variable *\$strblockname* made available in line 23, upon its assignment, it is tempting to have an ability to instrument event extraction code at line 24. The only problem with specifying events by line number is that a specification can easily be rendered inconsistent by updates to such script.

In consideration of the fact that PHP code is generally reused by the dynamic inclusion of common scripts into main executing scripts, points around script inclusion, in addition to those around method and function calls, can be ideal candidates for event extraction. Such points are as brittle as points around method and function calls, therefore can be specified for with little concern of rendering an event declaration inconsistent with any updates. Moreover, following inclusion, it is likely that other variables and functional declarations become available for use in the main executing script. Much of the functionality and variables used in ‘block.php’ have been made available after the inclusion of config.php and admin.php, in line 3 and 4 respectively.

Figure 4.2 gives a high level view of how scripts are generally reused in PHP WAs. For example, when *script.d.php* is requested for execution, at some point in its code it includes for execution *script.a.php*, and then at a subsequent point it also includes for execution *script.c.php*, until *script.d.php* finally terminates and returns its output. Similarly, *script.e.php*, when requested for execution, at some point it includes for execution *script.a.php*, and near its termination it includes for execution *script.b.php*, until it finally terminates and returns its output. *script.a.php* is shared by both scripts. Script inclusion is generally carried out by PHP in-

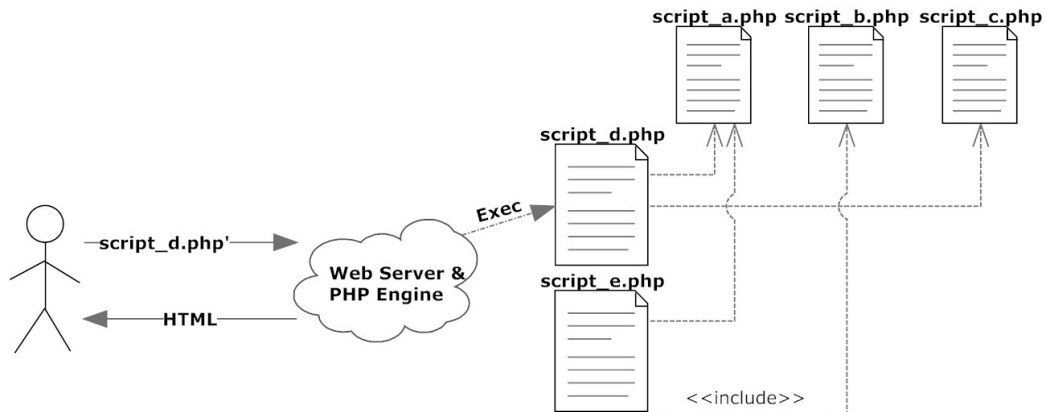


Figure 4.2: Dynamic inclusion of scripts in PHP WAs.

structions `require(<scriptpath>)` or `require_once(<scriptpath>)`, as in line 3 and 4 of the preceding PHP script (`block.php`). Consider the following event declarations:

```

1 event6(String id) = {afterRequire /dir/script_d "script_a.php"} where
   {id = $USER->id;}
2 event7(String id) = {uponExit /dir/script_a} where {id = $USER->id;}
3 event8(String username) = {uponEntry /dir/script_b} where {username =
   $_POST["username"];}
```

In the above event declarations we present three new event constructs. These event constructs were identified during the process of looking into possible ways of expressing the required events for the chosen properties in the case-study (chapter 6). In line 1 is specified an *afterRequire* event, for the extraction of events after the inclusion of `script_a.php` in `script_d.php`, every time `script_d.php` is executed. Moreover, at such point it captures runtime information that is made available by `script_a.php`, via the where clause. In line 2 is specified an *uponExit* event, for the extraction of events upon terminating `script_a.php`. At such point it captures the same runtime information as in the previous event, made available along the execution of `script_a.php`. The main difference is, that, since `script_a.php` is a shared script, an event is extracted both at `script_d.php` and `script_e.php`, by the inclusion

of *script_a.php*. Similarly, in line 3, is specified an *uponEntry* event. The main difference from *uponExit* is that an event is extracted upon entering *script_b.php* for execution. At such point it can still capture runtime information but from any of the PHP superglobal variables. In this example it captures specific information from a submitted HTML form via the post method.

Below is the BNF for the addition of three event constructs: *afterRequire*, *uponEntry* and *uponExit*.

```
1 <Path> ::= <DirectoryPath>'/'<ScriptName> | <DirectoryPath>
   '/*' | <DirectoryPath>'/**'
2 <AfterRequire> ::= 'afterRequire '<Path>' '<RequireParam>'>'>'
3 <UponEntry> ::= 'uponEntry '<Path>
4 <UponExit> ::= 'uponExit '<Path>
5 <ScriptBodyPoints> ::= <AfterRequire> | <uponExit> | <uponEntry>
6 <ScriptPrimEvent> ::= <ScriptBodyPoints> | <Path>' '<PrimitiveEvent>
```

4.4 Instrumentation of Event Extraction Code into PHP WA Code

A popular way of instrumenting event extraction code, as mapped from event declarations, into the system code is by the use of aspect-oriented programming (AOP) techniques. In fact, the implemented polyLarva plugins for Java and C resort only to AOP techniques for the instrumentation of such code. The reason for this is that, all possible events declarations targeting their programs can be mapped to AOP code, for their particular AOP implementation.

Considering the syntactical additions we made to the event declaration syntax, in the previous section, we want to find a way of mapping the modified syntax to an instrumentation technique suitable for use with PHP scripts. The most preferable method of instrumentation remains AOP, so our decisions will favour AOP over other methods of instrumentation.

In Section 4.3.1 we presented syntactical additions for the inclusion of a path to

the original event matching signature. Since the original event matching signature can fully map to the selected AOP implementations for Java and C, it is likely that it can also map to AOP implementations for PHP. The only difference for PHP is that an event declaration cannot not applied across the whole PHP WA, but only to a given path. This implies that an AOP implementation for PHP should be able to narrow down its applicability to at least a script. We looked into various PHP AOP implementations, such as PHPAspect² and Aspect-Oriented PHP³, however, the closest to this requirement, and the most comprehensive implementation we found was AOP PHP⁴. In the following sub-sections we are giving the details of AOP PHP together with examples of how it can be used for the instrumentation of event extraction code into PHP WA code.

4.4.1 AOP PHP

AOP PHP is a runtime-weaving implementation serviced via a PHP PECL extension⁵. PHP PECL extensions are generally used for the addition of specific functionality for use in scripts, such as to connect to a specific database (e.g. MySQL). AOP PHP makes available AOP constructs for their direct use in scripts just like regular PHP code.

In other implementations of AOP, such as AspectJ, used for the Java plugin, AOP code is declared separately from the target code, and is weaved into the target code at compile time. In PHP, AOP statements require no intermediate steps in order for them to work. They are interpreted and executed along with other PHP code within a script, performing a weaving-like process at runtime. Consider the following example:

²<http://code.google.com/p/phpaspect/>

³<http://www.aopphp.net/>

⁴<https://github.com/AOP-PHP/AOP>

⁵<http://pecl.php.net/>

```
1 <?php
2     Class MyClass {
3         public function someFunction() {
4             echo "some function <br>";
5         }
6     }
7
8     function logSomeFunction() {
9         echo "Calling somefunction() <br>";
10    }
11
12    aop_add_before( 'MyClass->someFunction()', 'logSomeFunction' );
13
14    $myObject = new MyClass();
15    $myObject->someFunction();
16 ?>
```

The above PHP code starts with the declaration of a class comprised of one object function. It is followed by the declaration of regular function *logSomeFunction()*, intended for use as an AOP advice, which is activated by AOP statement *aop_add_before*. It declares that advice *logSomeFunction()* must be called before every calling to object function *someFunction()*, matched by rule *MyClass->someFunction()*. The output of this PHP code, if requested from a web browser, is the following:

```
Calling someFunction()
some function
```

Similarly, an AOP declaration can be specified to trigger *after* a call to *somefunction()*, by the following code:

```
1 <?php
2     ...
3     aop_add_after ( 'MyClass->someFunction () ', 'logSomeFunction ' );
4     ...
5     $myObject->someFunction ();
6 ?>
```

Output:

```
some function
Calling someFunction()
```

In practice, AOP is not much effective if used only for the execution of isolated advices around target functions. It is mostly of use when used for the processing of any runtime information that is made available around the target functions. Such processing can range from the validation of form data to transaction logging. In AOP PHP, runtime information can be extracted for use in advices through a special AOP object, of type *AopJoinPoint*, which is passed as a function parameter to executing advices.

One might argue that by employing AOP declarations directly in the target script one still does not benefit from the AOP modularisation concept of crosscutting concerns. We corresponded with the developer of the AOP PHP extension about this and suggested a way how to alleviate this problem. He suggested that it is best that AOP instructions be coded into separate scripts, and include such AOP scripts into the target scripts. In this way target scripts will be modified only with a one-liner. Consider the following PHP code:

```
1 <?php
2     // file : aopcode.php
3
4     function logSomeFunction(AOPJointPoint $object) {
5         $args = $object->getArguments();
6         $returnValue = $object->getReturnValue();
7         echo "Value:". $args[0]. " Twice the value: " . $returnValue;
8     }
9
10    aop_add_after('MyClass->someFunction()', 'logSomeFunction');
11 ?>
```

```
1 <?php
2     require_once("aopcode.php");
3     Class MyClass {
4         public function someFunction($id) {
5             return $id*2;
6         }
7     }
8
9     $myObject = new MyClass();
10    echo $myObject->someFunction(20);
11 ?>
```

The above PHP code blocks represent two separate PHP scripts, the first with AOP code and the second with the target PHP code. The second script includes the first script (aopcode.php) in the first line of its code, in order to apply the AOP code across its subsequent code.

In the AOP script, advice function *logSomeFunction()* takes one parameter, *\$object* of type *AopJoinPoint*. An *AopJoinPoint* object can make available information such as call function parameters, the object from which function was triggered, and the returned value. In the above code, the advice function extracts two pieces

of information from *\$object*: the passed functions parameters, by calling *\$object->getArguments()*, and the returned value, by calling *\$object->getReturnedValue()*. Since there can be more than one function parameter, function parameters are outputted as an array of objects; the returned value is outputted as an object.

4.4.2 Translating Event Declarations to AOP PHP Code

In order for event declarations, as presented in Section 4.3, to be actuated, for the extraction of events from a PHP WA, it is required that they be translated into implementable code and instrumented into the PHP WA code. The following example illustrates a typical translation:

Event Declaration:

```
1 newCustomer(BankAccount account, long custId) = {/testing/* account.  
    add(custId ,*)}
```

Equivalent AOP PHP code:

```
1 <?php  
2 //file: event1.php  
3  
4 function event1(AOPJointPoint $object) {  
5     $args = $object->getArguments();  
6     $objectValue = $object->getTriggeringObject();  
7  
8     //Event Generation Code  
9 }  
10  
11 aop_add_before('BankAccount->add()', 'event1');  
12 ?>
```

The above AOP PHP code, as translated from the event declaration, is comprised of two main components: 1) the AOP declaration, defined at line 11, specifying the target object method and the attributed advice name, and 2) the advice, defined at line 4, comprised of the required runtime information extractions and the event extraction code. Such event script is applied, by script inclusion, to the path

specified in the event declaration, in this example to all immediate scripts under directory `/testing`.

4.4.3 Event Declarations not Translatable to AOP PHP Code

In Section 4.3.2 we presented syntactical additions for the addition of new event constructs. Such event constructs cater for event declarations oriented around script inclusion, and are specified differently from regular events based on method-`s/functions`. Such event constructs cannot be translated to AOP code and therefore, have to be instrumented by their direct injection into PHP WA code, at their respective points. Consider the following example:

Event Declaration:

```
1 htmlform(String username) = {uponEntry /dir/script_b} where {username  
   = $_POST["username"]};
```

Equivalent Event Construct code:

```
1 <?php  
2     //file: event2.php  
3  
4     function uponEntry2() {  
5         $username = $_POST["username"];  
6  
7         //Event Generation Code  
8     }  
9 ?>
```

The above event construct code is similar to that implemented using AOP PHP, however, without the declaration of an AOP matching rule. Such event script is applied, by script inclusion, to the path specified in the event declaration, in this example to script `'/dir/script_b.php'`. The only difference from AOP PHP is that the an event method call is injected directly in the target script at its indented

point in code. In this example, *uponEntry2()* is called upon entering *script_b.php*, right after the inclusion of the event script.

4.4.4 Event Generation Code

The role of the event generation code is to generate an event message based on runtime information made available in the advice, initialise a TCP socket connection with the remote monitor, and communicate the generated message over the established connection to the remote monitor. Below is the message structure used for the generation of an event message[16]:

```
'1','<event_id>','[<parameter_identifiers>]
```

The *event_id* is a numerical representation of an event declaration, agreed upon between the remote monitor and the event extraction code. The *parameter_identifiers* is a comma delimited list of parameters (for extracted runtime information) that is channelled along with an event. An example message, from the event example given in Section 4.4.3, is:

```
1,2,ratt0034
```

4.5 System-side Monitoring Logic Evaluation

Monitoring logic (states, conditions, actions) in a property specification script can be specified to execute either at the monitor side or at the system side. Any monitoring logic specified to execute at the system side should be readily expressed in the programming language of the system to be monitored in the property specification script; in this case in the PHP language. This, since it is meant to execute as part of the monitored system. System-side monitoring evaluations can take place from two main points in the monitoring process, in the scenarios below:

1. Following the generation and communication of an event, the remote monitor replies back, on the same connection, with a request for the evaluation of some

system-side monitoring logic. A trail of to-and-fro communication between the remote monitor and the system-side monitoring code can take place on the same connection, with a single round of communication for each system-side evaluation request, until all rules triggered by the initial event at the remote monitor are processed.

2. An event is fired outside the boundary of the monitored system, triggering rules which require that certain evaluations be carried out at the monitored system. As a result, the remote monitor tries to send an evaluation request to the monitored system, however, since there will not be an already available connection established by a preceding event, as in point 1, the remote proceeds by trying to initialise a connection itself with the monitored system, in order to communicate the evaluation requests on such connection. This requires that the system-side monitoring code be continually in a TCP-socket-listening state in the course of monitoring, by means of a PHP monitoring server. Following the establishment of a connection, the remote monitor proceeds in the same way as in point 1, after the generation and communication of an event.

4.5.1 Monitoring Contexts and System-side Evaluation Functionality

Figure 4.3 illustrates the main interacting parts of the system-side monitoring code. As already highlighted, system-side monitoring evaluations can take place, either by an evaluation request following the extraction of an event, at the instrumentation layer, or by a direct evaluation request through the PHP monitoring server. There can be declared multiple local monitoring contexts in a property specification script, each with their own system-side functionality and state variables. In the course of monitoring, local monitors are instantiated from such local monitoring contexts in representation of a particular monitoring instance with its own state values,

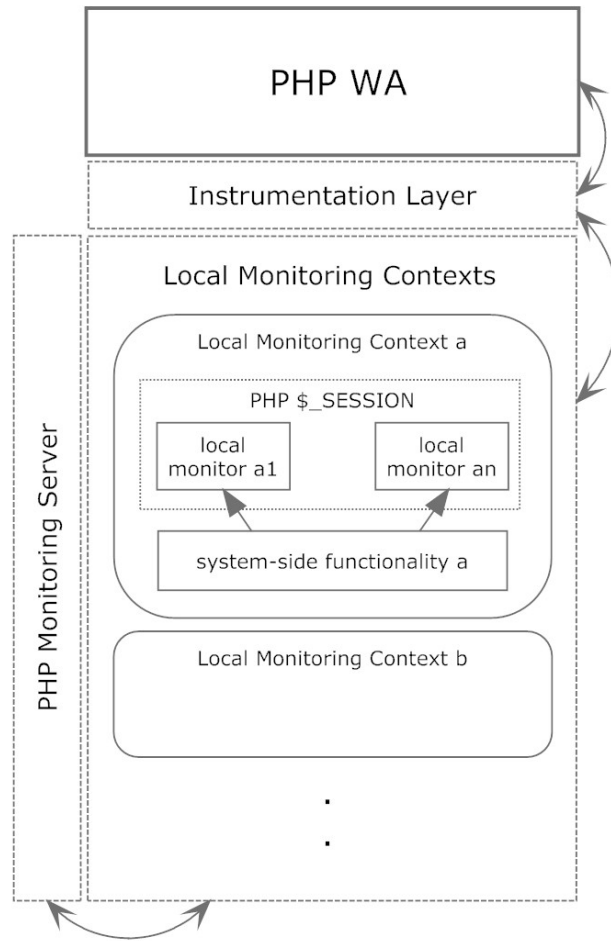


Figure 4.3: System-side monitoring code

such as for each newly created PHP WA session or shopping cart. Local monitors are identified by an attributed monitor handle (such as the particular session or shopping cart object) and are preserved in session elements of the PHP `$_SESSION` superglobal variable. Any monitoring logic that is evaluated at the system side, by means of system-side functionality, is evaluated in relation to some local monitor.

An evaluation request by the remote monitor, for the execution of a particular system-side functionality, is made by a message in the following message structure. *eval_func_id* is a numerical representation of the particular system-side functionality to be executed, and *parameter_identifiers* are parameters which can include a local monitor handle and event related information.

```
'2','<eval_func_id>','<parameter_identifiers>]
```

The response to an evaluation request is a true/false message, in the following structure:

```
'2','<eval_func_id>','<true/false>
```

When the remote monitor issues an evaluation request for the execution of an evaluation functionality residing in some context (except for the global context), it does so by including a local monitor handle in the parameters section of the evaluation message. Such handle serves for the retrieval of the particular local monitor from the PHP `$_SESSION` prior the execution of the requested system-side functionality, in order to make available its state variables for use during such execution. Following evaluation, the local monitor is preserved back in PHP `$_SESSION` so as to maintain any updates made by evaluation functionality to such variables.

4.6 Conclusion

In order to cater for the PHP language and PHP WA structures, the event matching signature of polyLarva had to be enhanced to include components that are specific to PHP scripts. Such enhancements required that the chosen AOP implementation for PHP (AOP PHP) be able to narrow down the applicability of AOP code to specified scripts, for ability to apply an event matching signature only to scripts in a specified path. Event declarations based on the original event matching signature were successfully translated to AOP PHP code, but other event declarations, due to targeting code structures specific to PHP scripts, were translated into functions (similar to AOP PHP advices) and instrumented into PHP WA code by direct injection.

Apart from event extraction, the system-side monitoring code also allows for the evaluation of any system-bound monitoring functionality to be carried out at

the system side, which can be requested for by the remote monitor at specific monitoring scenarios. The execution of any system-side functionality is carried out in relation to a particular monitoring instance (local monitor), representing a particular local monitoring state for the evaluation taking place. In PHP, local monitors are preserved in session elements of the PHP `$_SESSION` superglobal variable.

5. PHP Plugin Implementation

Details

5.1 Introduction

The process of mapping a property specification script to monitoring code is mediated through a generic plugin structure, which every technology-specific plugin should follow. Such structure is meant to be technology inclusive, based on abiding principles of how monitoring code should be compiled and instrumented into a target system. PHP is the first dynamic language to be supported, bringing with it certain fundamental differences from other supported programming languages. The way the PHP plugin was designed, for the catering of such differences, also had its effects on the way it was implemented.

In this chapter we start by looking into the generic plugin structure (section 5.2) and see how it fits with regards to the PHP monitoring code requirements (section 5.2.1). Following this, we present a way of extracting the system-side parts from a polyLarva specification script, using the plugin structure, into an easily parseable intermediate specification (section 5.2.3), for input to a bespoke monitoring code compiler and instrumentor. Then, we present a schematic view of the PHP monitoring code, as compiled from a given intermediate specification (section 5.3). Finally we give an overall view of the resulting PHP plugin structure

(5.3.2), highlighting parts which can be reused by prospective plugins.

5.2 polyLarva Generic Plugin Interface and Structure

In polyLarva, plugins are implemented in Java through a generic plugin interface. Such interface serves the purpose of aiding the plugin developer, as much as possible, in the extraction of system-side sections from a property specification script for the compilation of monitoring code in the intended programming language. It prescribes a number of methods, serving as placeholders, for their implementation in the plugin class. The body of such methods is where the plugin developer specifies the logic of how the resulting aspect and monitoring code should be generated, using system-side specification extractions made available for use for the particular sections of code generation. The whole process of monitoring code generation is driven by a common process, which invokes the prescribed plugin methods in a specific order, for the generation of both the aspect and monitoring code. The most notable methods of such process which make use of the prescribed plugin methods, are [16]:

- *createContextFiles* - invokes prescribed plugin methods for the compilation of code, comprised of local monitoring functionality and states, per context. Separate files are created for each context.
- *createContextAspectFiles* - invokes prescribed plugin methods for the compilation of aspect code per context, in the aspect language of the AOP implementation selected for the target programming language. Separate aspect files are created for each context.

The problem with the mapping presented in Figure 5.1 is that the resulting monitoring code structure is primarily designed to cater for AOP implementations that take as input aspect code (comprised of all event extraction code in a particular

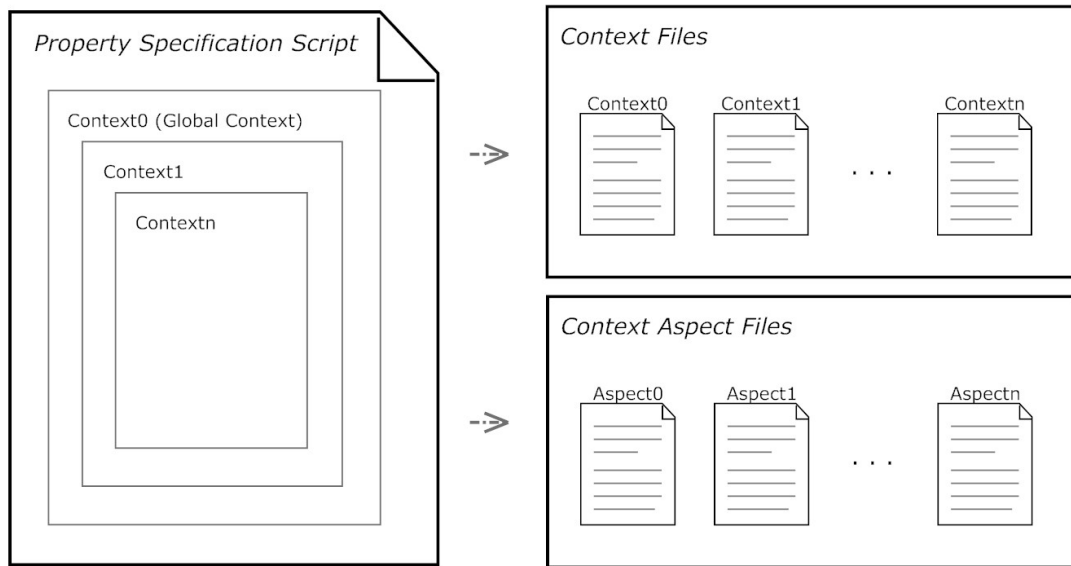


Figure 5.1: Monitoring code compilation process mapping

context) in separate files, and perform compile-time weaving into the target system using such AOP files. AOP PHP is a runtime-weaving implementation, requiring that for each event declared in a context, a representative script be created and included *only* into target scripts. The aspect files generated by the common compilation process are meant to be applied across the whole target system, and do not take into consideration script paths. Moreover, the added event constructs for PHP: *uponEntry*, *uponExit* and *afterRequire*, cannot be instrumented by aspect code. Consider the following mapping from event declarations to a PHP monitoring code structure:

```

1 global {
2   ...
3   events {
4     newSession(Session session) = {/test/* createSession(*)
5       uponReturning(session)}
6     htmlform(String username) = {uponEntry /dir/script_b}
7       where {username = $_POST["username"]};
8   }
9   ...
10 }

```

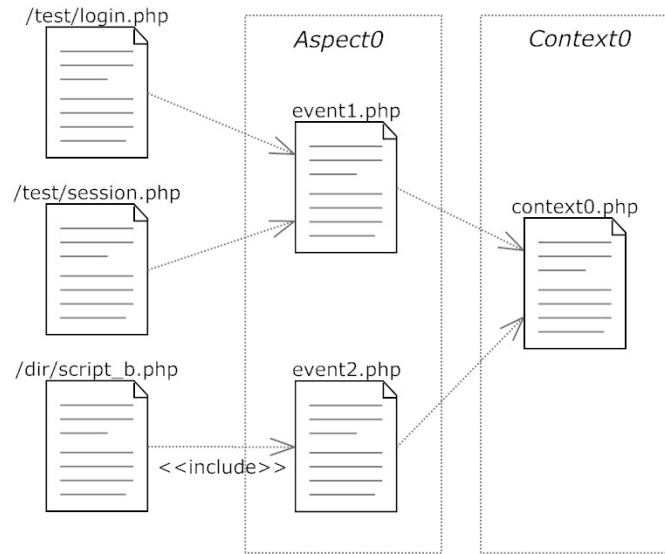


Figure 5.2: Exemplary PHP monitoring code structure

In the global context of the above code are declared two events, the first targeting a script directory and the second targeting an individual script. As presented in the previous chapter, event scripts are created for each event declaration, and where these are included into their target scripts. Figure 5.2 illustrates an exemplary monitoring code structure as mapped from its preceding event declarations. In this example, a context script is shared among all events declared in the same context, and where events targeting a directory of scripts are also shared among their target scripts. In the mapping presented in Figure 5.1, also partly shown in Figure 5.2 in dotted squares, the level of granularity for event scripts, and the relationship between event scripts and target scripts, through a given path, are not supported.

5.2.1 Extracting an Intermediate Specification for Monitoring Code Compilation

In order to overcome the limitations of the prescribed plugin structure, it is needed that certain code compilation processes be carried out outside such structure bounds.

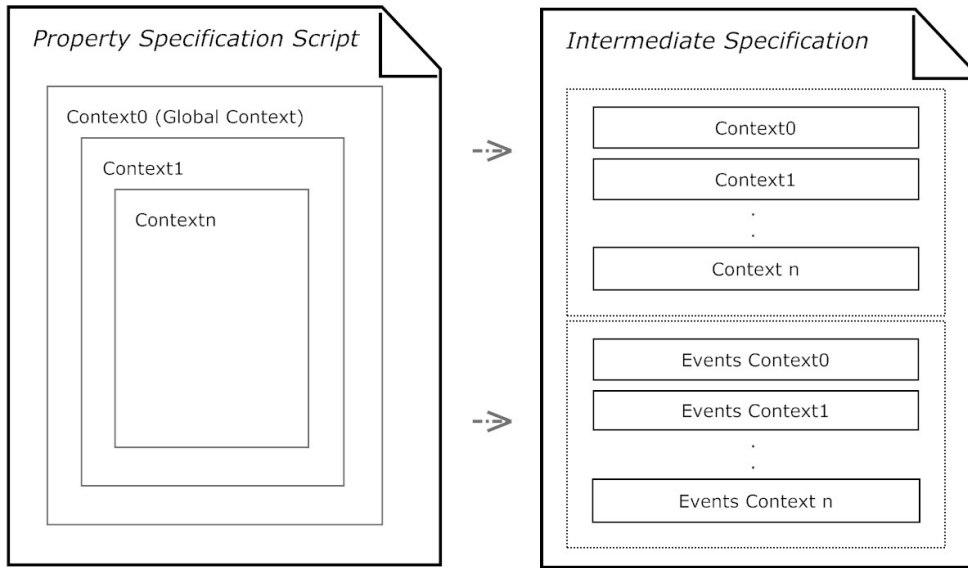


Figure 5.3: Intermediate specification mapping from a property specification script

The plugin structure is going to be used only for the extraction of the required system-side parts from a supplied property specification script, in the form of an intermediate specification, whose mapping is illustrated in Figure 5.3. A special purpose polyLarva plugin was implemented for the extraction of an intermediate specification, tweaked to output all system-side extractions into a single file, rather than in separate files. The intermediate specification is structured in a way that it can easily be parsed and tokenised by another program, for the compilation of PHP monitoring code (system-side monitoring code). Below is an example of the mapping between a property specification script and an extracted intermediate specification from such specification:

Property Specification Script:

```

1 global {
2   states {
3     systemSide {
4       PHPVar sessionCount { ... }
5     }
6   }

```



```

7  conditions { }
8  actions {
9    systemSide {
10     incrementSessionCount = {
11       if ($this->sessionCount == "null"){
12         $this->sessionCount = 1;
13       }
14       $this->sessionCount++;
15     }
16   }
17 }
18 events {
19   newSession(Session session) = {/test/session createSession(*)
20     uponReturning(session)}
21   htmlform(String username) = {uponEntry /dir/script_b}
22     where {username = $_POST["username"]};}
23 }
24 rules { ... }
25 }

```

Extracted Intermediate Specification:

```

1 @CODECONTEXT:START:0
2 %PARENTCONTEXT:NULL:NULL:NULL
3 %EVENTVARS:session, : false ,
4 %SYSTEMSIDEVARS:sessionCount ,
5 %CONDITIONS:START
6 %CONDITIONS:END
7 %ACTIONS:START
8 #ID:0:incrementSessionCount
9 if ($this->sessionCount=="null"){ $this->sessionCount = 1;}
10 $this->sessionCount++;
11 #ID:0:END
12 %ACTIONS:END
13 @CODECONTEXT:END:0
14

```

```
15 @EVENTS:START:0
16 %PATH:/ test /session
17 %JOINPOINT: Call :NULL
18 %EVENTID:1
19 %METHODNAME: createSession
20 %VAR:RETURN:0: false : session
21 —
22 %PATH:/ dir /script_b
23 %JOINPOINT: uponEntry :NULL
24 %EVENTID:2
25 %VAR:WHERE:0: true : username = $_POST[" username "];
26 —
27 @EVENTS:END:0
```

A intermediate specification is divided into two main sections, one for local monitoring functionality and states, with label ‘@CODECONTEXT’ (line 1), grouped by context, and the other for events, with label ‘@EVENTS’ (line 15), also grouped by context. Individual events are separated by a double dash in ‘@EVENTS’ context blocks. Both ‘@CODECONTEXT’ and ‘@EVENTS’ context blocks are enclosed by special start and end labels, and are identified by the attributed context ID. Since there is only one context in this example, there is one context block for local functionality and states, and another context block for events.

Figures 5.4, 5.5 and 5.6 present the BNF of the intermediate specification, divided according to the main sections:

5.3 Monitoring Code Compilation and Instrumentation

In the previous section we described the initial process of monitoring code compilation, that of extracting the system-side definitions, from a given property specifica-

```

1 <Primitive> ::= 'true' | 'false'
2 <ParentCtxtID> ::= <ID> | 'NULL'
3 <ParentCtxtVar> ::= <Var> | 'NULL'
4 <Prim> ::= <Primitive> | 'NULL'
5 <ParentCtxt> ::= <ParentCtxtID>': '<ParentCtxtVar>': '
6 <Prim>
7 <Primitives> ::= <Primitive>', ' | <Primitives><Primitive>
8 <EvtVar> ::= <VarName>', '
9 <EvtVarList> ::= <EvtVar> | <EvtVarList><EvtVar>
10 <EventVars> ::= <EvtVarList>': '<Primitives>
11 <SysSideVar> ::= <VarName>
12 <SysSideVars> ::= <VarName>', ' | <SysSideVars><VarName>
13 <SysSideFunc> ::= '#ID: '<FuncID>': '<FuncName><PHPCode> '#ID: '<FuncID>',
:END'
14 <Condition> ::= <SysSideFunc>
15 <Conditions> ::= <Empty> | <Condition> | <Conditions><Condition>
16 <Action> ::= <SysSideFunc>
17 <Actions> ::= <Empty> | <Action> | <Actions><Action>
18 <Context> ::= '%PARENTCONTEXT: '<ParentCtxt>
19 '%EVENTVARS: '<EventVars>
20 '%SYSTEMSIDEVARS: '<SysSideVars>
21 '%CONDITIONS:START '<Conditions>'%CONDITIONS:END'
22 '%ACTIONS:START '<Actions>'%ACTIONS:END'

```

Figure 5.4: Intermediate specification: @CODECONTEXT block

tion script, into an intermediate specification. In this section we are breaking away from the prescribed plugin structure, and rely totally on the intermediate specification for the compilation and instrumentation of monitoring code. In Figure 5.7 we present a schematic view of the overall monitoring code. The monitoring code is comprised of five main parts:

- *Event Scripts* incorporate all the necessary functionality for the generation of events. Each event script is the functional representation of one event declaration, as mapped from the intermediate specification. Event scripts are included into their target scripts by a bespoke monitoring code integrator, using the path entry which is made available the intermediate specification. Events scripts also have the facility to perform any system-side monitoring evaluations (through the Context Script) for cases when the remote monitor requests so, following an event.

```

1 <Return>      ::= 'RETURN: '<OccurNo>': '<Primitive>': '<VarName>
2 <Target>     ::= 'TARGET: '<OccurNo>': '<Primitive>': '<VarName>': '<
   VarType>
3 <MthodParam> ::= 'PARAM: '<OccurNo>': '<Primitive>': '<VarName>
4 <MthodParams> ::= <MthodParam> | <MthodParams>', '<MthodParam>
5 <Where>      ::= 'PARAM: '<OccurNo>': '<Primitive>': '<WhereAssignment>
6 <Wheres>     ::= <Where> | <Wheres>', '<Where>
7 <Path>       ::= '%PATH: ' <DirectoryPath>' / '<ScriptName> | <
   DirectoryPath>' / '*' |
8               <DirectoryPath>' / '**'
9 <JoinPoint>  ::= '%JOINPOINT: ' <EventType>': '<afterRequireParam>
10 <EventID>    ::= '%EVENTID: ' <ID>
11 <MethodName> ::= '%METHODNAME: ' <Name>
12 <Var>        ::= '%VAR: '<Return>|<Target>|<MthodParams>|<Wheres>
13 <Vars>       ::= <Var> | <Vars><Var>
14 <Event>      ::= <Path><JoinPoint><EventID><MethodName><Var> '—'
15 <CtxtEvents> ::= <Event> | <CtxtEvents><Event>

```

Figure 5.5: Intermediate specification: @EVENTS block

```

1 <Contexts>    ::= '%CODECONTEXT:START: '<CtxtID><Context>
2               '%CODECONTEXT:END: '<CtxtID> | <Contexts><Context>
3 <EventCtxts> ::= '%EVENTS:START: '<CtxtID><CtxtEvents>
4               '%EVENTS:END: '<CtxtID> |
5               <EventCtxts><CtxtEvents>
6 <Specification> ::= <Contexts><EventCtxts>

```

Figure 5.6: Intermediate specification: main specification structure

- *Contexts Script* incorporates all the monitoring contexts, and where it is included in every event and the PHP monitoring server scripts, providing them with necessary functionality for the carrying out of system-side monitoring evaluations. Each context in this script is encapsulated with its own local monitoring functionality and states, as mapped from the given intermediate specification. Contexts are implemented as OO classes, for their instantiation into objects when a contextual evaluation needs to take place. An object structure facilitates a contextual interaction between local monitoring functionality and state variables.
- *Communication Script* incorporates all communication information, including the IP address and port of the remote monitor, and the necessary func-

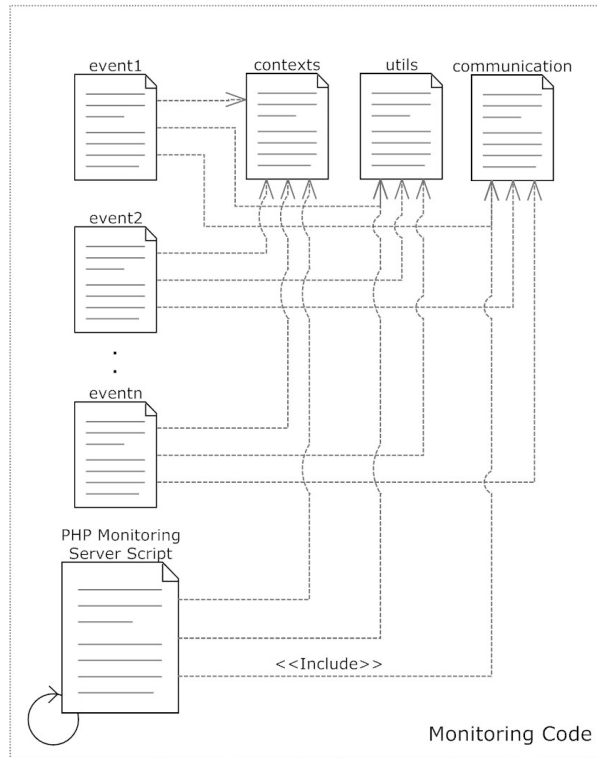


Figure 5.7: Monitoring code structure

tionality for the ability to communicate over TCP sockets with the remote monitor. Its main purpose is to provide helper functions to the other monitoring scripts, for the sending and receiving of monitoring messages, to and from the remote monitor.

- *Utilities Script* incorporates helper functions for use by the other monitoring scripts. It is comprised of functionality for the preservation and retrieval of state variables (representing some local monitor) for use during monitoring evaluations across script executions. Such functionality is achieved by the use of PHP superglobal `$_SESSION` variable, in which state variables are preserved.
- *PHP Monitoring Server Script* is an independent PHP script which is never meant to terminate. It is executed independently right after the PHP engine is started. Its main task is to listen for connections requests that can be

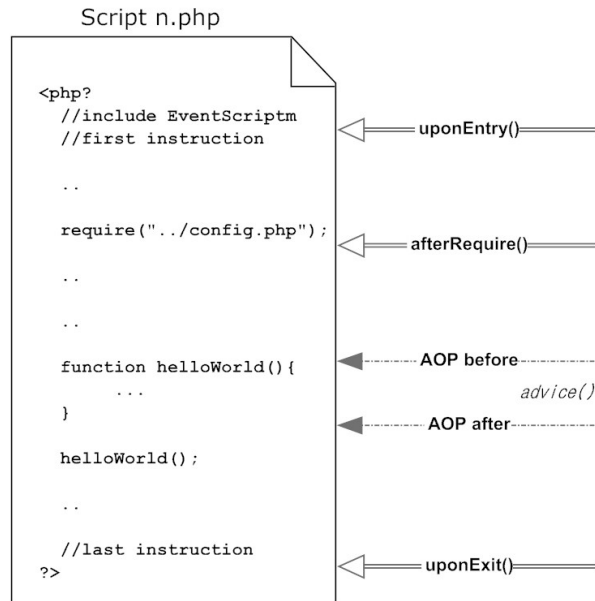


Figure 5.8: Instrumentation points in PHP scripts

issued by the remote monitor. The process, following the establishment of a connection, is very similar to that when the remote monitor gets back to the monitored system, after an event, requesting for the evaluation of some system-side functionality.

In Figure 5.7 are illustrated the various script inclusions that occur among the monitoring scripts. At the very beginning of their code, event scripts include the contexts, communication and utilities scripts, in order to make available their functionality and data structures for use by subsequent event script code. The same is performed in the PHP monitoring server script.

5.3.1 Instrumenting Event Extraction Code into Target PHP Scripts

In Chapter 4 we highlighted about the need for an event to be applied in the confines of a target script or scripts, and not across the whole PHP WA. A path was added to the event matching signature in order to facilitate this. Such path, which is also

made available in the intermediate specification, serves the purpose of identifying which script or scripts (if it is a directory path) in a PHP WA are targeted for instrumentation by a particular event declaration. Identified scripts are injected with code for the inclusion of the event script, as mapped from the particular event declaration, using a bespoke implementation for automatic code injection. More specifically, an event script is included at the very beginning of the target script/s, as illustrated in Figure 5.8, for the applicability of event functionality to all subsequent target script instructions.

PHP scripts can have various instrumentation points at which events can be extracted. Figure 5.8 illustrates two kinds of instrumentation points, differentiated by the different arrows shapes pointing at the script code. The first kind are those which can be latched on to by AOP declarations. With AOP PHP, event generation functionality can automatically be latched on to functions or class/object functions, either before or after they are called for execution. Declared AOP code for these is generated as part of the event script code, and is applied to target script functionality via the inclusion of such event script into the target scripts. Therefore, for events involving AOP PHP code, target scripts are only modified to include an event script. The other kind of instrumentation points represent event constructs *uponEntry*, *uponExit* and *afterRequire*. These kind of instrumentation points cannot be latched code on to by AOP, but by the direct injection of event methods calls at their intended points. For example, if an event declaration is of type *uponEntry*, a method *uponEntry()* is generated as part of the relative event script code, similar to an AOP advice. The only difference from an AOP advice is that a method call to *uponEntry()* is directly injected at the designated point in the target script code; in such case, just after the inclusion of the defining event script, as illustrated in Figure 5.8.

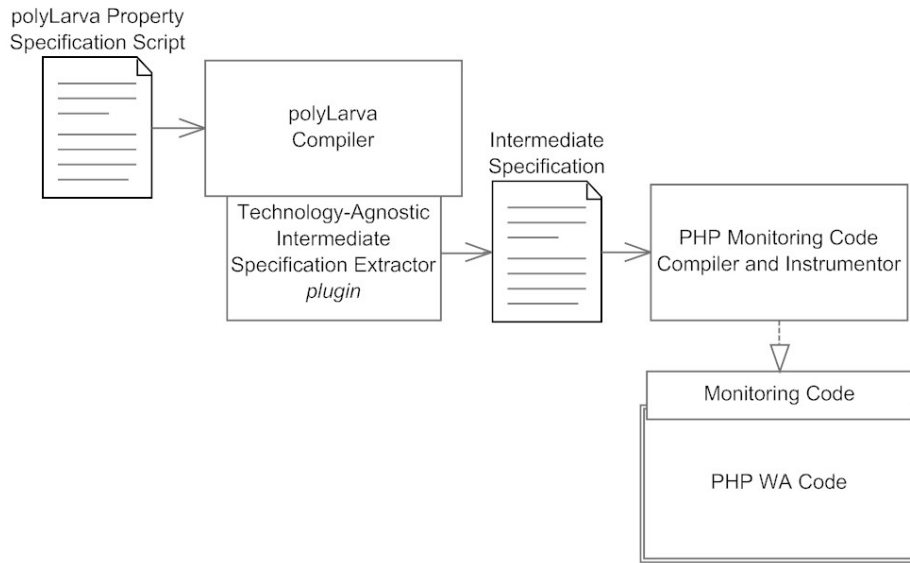


Figure 5.9: Overall PHP plugin architecture

5.3.2 PHP Plugin Architecture

The PHP plugin architecture presented in Figure 5.9, unlike the other implemented plugins, was implemented with an intermediary element, for the extraction of an intermediate specification. The intermediate specification extractor was implemented as a polyLarva plugin in itself, using the prescribed plugin structure. It is technology agnostic, since its extractions can be used for the compilation and instrumentation of monitoring code in systems developed in any technology. Following the extraction of an intermediate specification, it is then given as input to the bespoke PHP monitoring code compiler and instrumentor to first compile the required PHP monitoring code, then instrument such code into the target PHP WA code.

5.4 Conclusion

The presented plugin structure, as prescribed by polyLarva, could not entirely fit a PHP plugin for the direct compilation of PHP monitoring code, as it is primar-

ily designed to fit plugins making use of compile-time weaving AOP implementations. Consequently, we used the plugin structure only for the extraction of an easily parseable intermediate specification, comprised of system-side sections from a property specification script. Following this, we presented a schematic for monitoring code that is compiled by the bespoke PHP compiler and instrumentor from a given intermediate specification. We also have gone into the details of how the system-side monitoring code is instrumented into a PHP WA, in order to make a PHP WA monitorable. Finally, we presented the overall plugin architecture, comprised of a combined reusable intermediate specification extractor, implemented as a polyLarva plugin in itself, and a bespoke PHP monitoring code compiler and instrumentor.

6. Case Study

6.1 Introduction

To show the practicality of polyLarva with PHP scripts, it is essential that it is evaluated against a real-life PHP WA, preferably one that exhibits interesting properties and is widely used in its intended context. The main advantage of PHP WAs is that their code is by definition open, since it is not compiled, and that they are mostly maintained by open source communities. In contrast to closed and proprietary systems, one is usually able to browse PHP WA documentation and find related community forums with relative ease, for acquaintance with their PHP code and to gain knowledge of any related concerns.

The main aim of this chapter is to appraise the validity and relevancy of the polyLarva PHP plugin through a number of monitoring scenarios, presented in a case study. The case-study is an e-learning WA called Moodle¹ (fully implemented in PHP). We will look into some sections of Moodle for analysis and verify them for properties based on a certain specification and security criteria handed to us by the Moodle team at the University of Malta (UoM), as employed for their Moodle implementation, branded as the UoM Virtual Learning Environment (VLE)². The terms VLE and Moodle are used interchangeably in this chapter. Moreover, we

¹<https://moodle.org/>

²<http://www.um.edu.mt/vle>

will consider other properties, common to the majority of PHP WA, such as on session timeouts. Lastly, we will go through the project objectives and test them through the presented case-study scenarios, providing a coherent justification of how they were sufficiently met. In the following section we will go into the details of the case-study.

6.2 Moodle

Moodle is a PHP-based e-learning platform aimed at providing online tools for the creation and delivery of educational content, operating in a managed environment which can scale up to over a million users. It provides an online environment for tutors and students to come together in the context of educational courses, allowing for various forms of communication, such as online forums and quizzes. It has been employed in various sites (mostly universities), most notably in the US with over eleven thousand Moodle installations. The UoM uses Moodle in order to provide an online platform for study units (courses) which are taken up by classes of students. In a study unit, tutors and students can share material and thoughts relevant to the topics covered. These include the publishing of notes and assignment descriptions, forums for discussion, wikis, and even a facility for assignment submission.

During the development of the PHP plugin various functionality aspects were tested on a local installation of Moodle with configurations analogous to that of the VLE of UoM. We have set up a dummy study unit for testing with the same educational components (modules) as set up in the UoM VLE. We also created a handful of disparate users to serve as actors for the necessary testing and running of the case-study.

A number of properties on Moodle behaviour were verified using polyLarva, through the developed PHP plugin. The properties are described in detail below.

Permission	Icon	Roles
Add news <small>mod/forum:addnews</small>		Non-editing teacherX, TeacherX, ManagerX +
Add question <small>mod/forum:addquestion</small>		Non-editing teacherX, TeacherX, ManagerX +
Allow force subscribe <small>mod/forum:allowforcesubscribe</small>		StudentX, Non-editing teacherX, TeacherX +
Create attachments <small>mod/forum:createattachment</small>		StudentX, Non-editing teacherX, TeacherX, ManagerX +
Delete any posts (anytime) <small>mod/forum:deleteanypost</small>		Non-editing teacherX, TeacherX, ManagerX +
Delete own posts (within deadline) <small>mod/forum:deleteownpost</small>		StudentX, Non-editing teacherX, TeacherX, ManagerX +
Edit any post <small>mod/forum:editanypost</small>		Non-editing teacherX, TeacherX, ManagerX +
Export whole discussion <small>mod/forum:exportdiscussion</small>		Non-editing teacherX, TeacherX, ManagerX +
Export own post <small>mod/forum:exportownpost</small>		StudentX, Non-editing teacherX, TeacherX, ManagerX +

Figure 6.1: Moodle permission management

6.2.1 Property#1 - Cross-verifying Moodle Permission Management (MPM)

Figure 6.1 shows a Moodle permission page in relation to a ‘forum’ module, giving the facility to manage its permissions based on Moodle roles. For each permission, there can be assigned as many roles for the granting or denying of such permission. Roles are not attributed to users directly, but are assigned to them in the context of a study unit. For example, a user can have a teacher role in one study unit and a student role in another study unit. This gives the facility for permissions to be assigned differently in different study units.

As a general policy, the UoM VLE team decided to set up user permissions identically across all study units, assigning UoM users with one role for use across all enrolled-in study units. Thus, they created a one-to-many relationship table between the enabled feature permissions and the available roles, in the form of a basic permission specification. Such specification is eventually mapped to actual Moodle permission settings. Table 6.1 is an abridged representation of such specification.

Permission Group	Permission Description	Tutor	Student
Forums	Create a Forum	Allow	Deny
	Read Post	Allow	Allow
	Add Post	Allow	Allow
	Remove post	Allow	Deny

Table 6.1: UoM VLE abridged permission specification

We have set the permission settings in our local Moodle installation according to the permission descriptions, as in Table 6.1, by mapping them with the relative Moodle permissions. For example, ‘Create a Forum’ permission maps to Moodle permission ‘add a new Forum’ with Moodle code ‘mod/forum:addinstance’, and ‘Remove Posts’ maps to Moodle Permission ‘delete any posts (any time)’, with Moodle code ‘mod/forum:deleteanypost’.

In the VLE, users are created in batch, by the importation of data from other UoM systems. A user is imported with a descriptive field denoting his global user role, and where such field is used for the eventual assigning of a Moodle role, upon the user enrolls into some study unit. During the importation process there is a mechanism that checks, for already existing users, whether their assigned study unit roles still reflect the newly imported global user role. If it finds that the global user role is different from the study unit roles, it reconciles the study unit roles in order to reflect the updated global user role. A failure in such mechanism can lead to users being granted or denied permissions erroneously when navigating Moodle.

We want to define a property, to be verified by polyLarva, for the checking of whether permissions are granted or denied correctly by Moodle by checking for inconsistencies between the supplied permission specification supplied and the actual Moodle permissions settings, based on the global user role and the actual assigned Moodle roles.

Moodle takes the approach of inserting permission check functionality in various places across its PHP scripts, in order to check permissions in relation to the executing code. Below is a code snippet illustrating some permission checking functionality cross-cutting the presentation logic of a Moodle PHP script.

```
1 if ($isfrontpage) {
2     $PAGE->set_pagelayout('admin');
3     require_capability('moodle/site:viewparticipants', $systemcontext);
4 } else {
5     $PAGE->set_pagelayout('incourse');
6     require_capability('moodle/course:viewparticipants', $context);
7 }
```

The function *require_capability()* serves the purpose of checking whether a permission, in a particular context, should be granted. If it finds fault with the combination an error is thrown, resulting in a Moodle error page. This function is used all over Moodle scripts for such purpose. In the body of this function there is another function *has_capability()*, which returns a boolean value, determining whether *require_capability()* should throw an error or not. Sometimes, *has_capability()* is also used directly for certain permission checks in conditional statements.

Below are the most important parts of the polyLarva specification for the verification of this property:

```
1 states {
2     monitorSide {
3         PermissionMap<String, String> map { .. }
4     }
5 }
```

The purpose of state `PermissionMap` in this property is to hold a list of Moodle permissions and their attributed Moodle roles in a dictionary. This, in order to load the VLE permission specification into the remote monitor for use by the monitoring logic. Below are some example entries from the specification that is loaded into the monitor, in the form of Moodle permission codes and their allowed Moodle roles. For example, 'mod/forum:startdiscussion' can be granted to both 'Tutor' and 'Student' roles.

```
1 mod/forum:addinstance\&
2 mod/forum:deleteanypost\&Tutor
3 mod/forum:startdiscussion\&Tutor, Student
```

These specification entries are loaded from a file using standard Java file management code, placed within the state clauses of *PermissionMap*.

```
1 events{
2   access_rights(String perm, String role, String path, Boolean result
3     , String user) = {/lib/accesslib has_capability(perm, *)
4     uponReturning(result)}
5   where {
6     role = $USER->aim;
7     path = $_SERVER["REQUEST_URI"];
8     user = $USER->idnumber;
9   }
}
```

Event *access_rights* is set to fire every time function *has_capability()*, located at ‘lib/accesslib.php’, returns a value. ‘accesslib.php’ is a shared Moodle script which is included in almost every other Moodle script. Through this event are made available, the permission code being evaluated by the function, via the captured function parameter, and the permission granting result, via the returned value. This event also includes a where clause by which certain other runtime information, outside the bounds of the calling function, is captured, which includes the global user role, the path of the executing script and the user ID.

```
1 conditions {
2   monitorSide { ruleExists = { .. }
3     isAllowed = { .. }
4   }
5   systemSide {
6     evalResult = {return $this->result == true;}
7   }
8 }
9 actions {
10  monitorSide {
11    printinfo = {System.out.println("User: " + user + " Path: " +
12      path + " Violation: " + perm);}
13  }
14 }
```

In this property there are three conditions in total, two of them evaluated at the monitor side, and another at the system side. *ruleExists* checks whether the permission in question is included in the permissions specification, while *isAllowed* checks whether a permission, in the permissions specification, is allowed or denied in relation to the captured global user role. *evalResult* reflects whether Moodle has allowed or denied the permission in question.

In the actions section there is one action, to be executed whenever the property is violated. It outputs a message at the monitor side, giving details about the particular violation.

```
1
2 rules {
3     isInconsistentA = access_rights(perm, desig, path, result, user)
4         \((ruleExists && !isAllowed && evalResult))
5         -> printinfo;
6     isInconsistentB = access_rights(perm, desig, path, result, user)
7         \((ruleExists && isAllowed && !evalResult))
8         -> printinfo;
9 }
```

The rules for this property are set to compliment each other, as they check for inconsistencies between Moodle and the permission specification, in the granting/-denying of a particular permission. *isInconsistentA* evaluates to true if the permission is deemed as grantable by the permission specification, but denied by Moodle, and *isInconsistentB* its converse. Their processing to true leads to a property violation.

Detecting a Violation for Property#1

In order to simulate a property violation, we intentionally rendered permissions inconsistent in Moodle from two different points, simulating scenarios: 1) when the global user role is updated and the role reconciliation process fails and 2) when the Moodle administrator grants permissions for testing purposes to a certain role, and he fails to remove them prior Moodle goes live.



Figure 6.2: Descriptive field used for global user role

The first is a case where an education student, ‘Ritianne Attard’, who is participating in an eLearning course, is temporarily promoted to a ‘Tutor’ role in the VLE. The general procedure is that her global user role be changed to ‘Tutor’ from the source student record in the student management system, in order to effect the Moodle field, as illustrated in Figure 6.2, in the following data import from such system. With a change like this, during data import, the reconciling mechanism should flag the user and update all her assigned study unit roles to ‘Tutor’. The problem was that the reconciling mechanism failed during such process, due to requiring a revision update.


First name / Surname ↓ / Email address	Last access	Roles
 Ritianne Attard	8 mins 20 secs	Student ✕

Figure 6.3: Assigned moodle role in study units

Figure 6.3 shows the actual assigned role for ‘Ritianne Attard’, in relation to her study units. The reconciling mechanism should have removed the role ‘Student’ and added ‘Tutor’ instead, in order to reflect the global user role as in Figure 6.2.

When ‘Ritianne Attard’, with user ID ‘ratt0034’, entered into the discussion page of the forum, certain checks were performed in order to determine what links to functionality should be placed on the rendered HTML page; one of these was for the creation of a *delete* link for posts. Since ‘Ritianne Attard’ is still assigned a ‘Student’ role in the mentioned study unit, this was determined by Moodle as not being permissible, while the monitor, basing on the global user role, as captured in the event, and the given permission specification, in Table 6.1, has determined that such permission is grantable. This mismatch resulted in the property violations illustrated in Figure 6.4.

Below is a monitor log illustrating a permission checking event leading to a

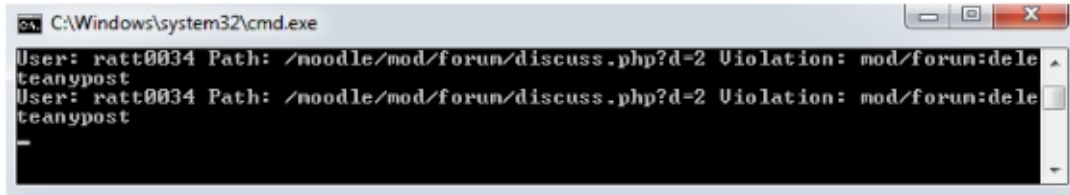


Figure 6.4: Permission violations for 'mod/forum:deleteanypost'

property violation:

```
[FINE]Received from system: 1,1,51eb7fca040e3,mod/forum:deleteanypost,Student,
/moodle/mod/forum/discuss.php?d=2,ratt0034
[INFO]Context {permission_0} received message that matches Event Collection
{access_rights}
[INFO]Context {permission_0} Rule activated {isInconsistentA}
[INFO]Sending to System : Context {permission_0} evaluate Condition
{evalResult}
[FINE]Sending to system: 2,2,mod/forum:deleteanypost,Student,
/moodle/mod/forum/discuss.php?d=2,51eb7fca040e3,ratt0034,
[FINE]Received from system: 2,2,false
[INFO]Context {permission_0} Rule activated {isInconsistentB}
[INFO]Context {permission_0} Rule {isInconsistentB} evaluates to TRUE.
Rule Processed..
```

In the second scenario, the administrator is testing some functionality on an upgraded version of Moodle, by the adding and removing of roles to permissions as in Figure 6.5.

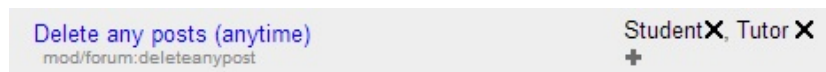


Figure 6.5: Incorrect 'Student' permission

Here, the administrator added a 'Student' role to 'mod/forum:deleteanypost' for testing purposes, and failed to take it off when ready. Similar to the previous scenario, it occurred that when students navigated to the forum discussion page of the particular study unit, a property violation occurred as in Figure 6.4. Moodle

determined that the deletion of any post is permissible for the role ‘Student’ while the monitor, basing on the global user role ‘Student’, and the permission specification, determined that such permission is not grantable.

6.2.2 Property#2 - Further Protection on Moodle Administration

In the previous property we employed a cross-verification mechanism on the granting/denying of permissions between Moodle and a permission specification, reflecting permission configurations in Moodle. This property, however, was based on the assumption that permission checking functionality is always in place, where permissions need to be checked. There is no guarantee that permission checking functionality is placed exactly and appropriately before any restricted functionality, in every script. With a Moodle version upgrade such cross-cutting checks can easily be removed erroneously from PHP scripts, and where in their absence the monitor cannot produce any violations, since the primary event that triggers verification is generated due to the execution of the checking function itself.

The UoM technical team has given the VLE team a partial solution, that of protecting Moodle administration scripts for such possibilities via further restrictions on the URL, using a reverse proxy³ mechanism. A reverse proxy mechanism acts as an intermediary between users and Moodle. Users will not be able to communicate directly with Moodle, but only through such mechanism. Having this in place, it will allow restrictions to be placed at such mechanism based on various parameters, such as the URL and the requester IP address.

Moodle administration scripts are all grouped under directory ‘/moodle/admin/*’. In the reverse proxy mechanism, restrictions were placed so that all administration scripts are allowed to be requested only from certain IP address ranges within the UoM campus. If requests are made from IP addresses outside such

³<http://www.f5.com/glossary/reverse-proxy/>

ranges, the reverse proxy mechanism automatically responds to such requests with an error page, without executing any Moodle PHP script.

We want to define a property so as to compliment such mechanism, by checking whether any allowed access (based on IP Address) is given to users having a global user role ‘Manager’. This time, rather than relying on permission checking functionality, for the generation of events, we want to base event generation directly on script requests, for the execution of administration scripts.

Below are the most important parts of the polyLarva specification for the verification of this property:

```
1 events {
2     scriptRequest(String role, String user, String path) =
3     {afterRequire /admin/** "../config.php"}
4     where {
5         role = $USER->aim;
6         path = $_SERVER["REQUEST_URI"];
7         user = $USER->idnumber;
8     }
9 }
```

Event *scriptRequest* is set to fire after the inclusion of ‘config.php’, in any script within ‘/admin’ directory of Moodle, and its subdirectories. This event has a where clause by which the necessary runtime information is captured, which includes the global user role, the path of the executing script and the user ID. The reason this event is set to fire after the inclusion of ‘config.php’ is due to certain required runtime information (‘\$USER’ related) becoming available after the inclusion of such script.

```
1 conditions {
2     monitorSide {
3         isManager = {return role.equals("Manager");}
4     }
5 }
```

Condition *isManager* uses Java object method *string.equals()* on string *role*, rep-

resenting the global user role, to check whether it matches ‘Manager’, returning a boolean value indicating so.

```
1 rules {  
2   isNotAllowed = scriptRequest(role , user , path) \ !isManager ->  
   printInfo;  
3 }
```

Rule *isNotAllowed* specifies that if a script request is not made by a user with global user role ‘Manager’, it raises a property violation at the monitor side, outputting a message by executing action *printInfo*.

Detecting a Violation for Property#2



Figure 6.6: Student attempting access to ‘/moodle/admin/index.php’

‘Ritianne Attard’ is currently in the range of IP addresses which are allowed to request for the execution of administration scripts, however, she is currently with global user role ‘Student’. Since she is a student, Moodle itself does not allow her to access such pages and redirects her to an error page, as illustrated in Figure 6.6.

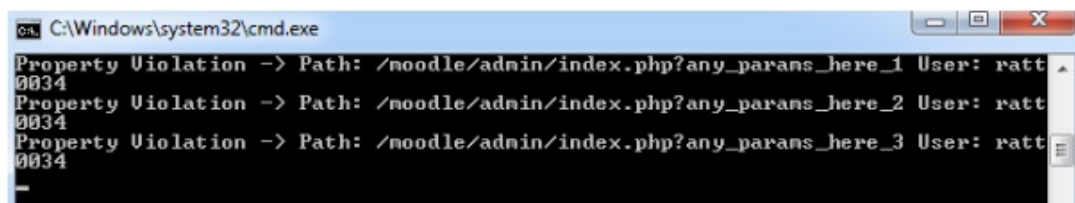


Figure 6.7: Property violations indicating the attempted URL with parameters, and the User ID

The remote monitor, on the other hand, takes into consideration the global user role for the determining of whether she should have access to administrative scripts.

Since the only role that is allowed to access such scripts is ‘Manager’, a property violation occurs every time she tries to access a Moodle administration script. Figure 6.7 illustrates three property violations from the same user, each representing different attempts to the same administration script, but with different URL parameters. Such property can serve for the identification of users, in allowed IP ranges, who deliberately attempt to access Moodle administration scripts without themselves having global user role ‘Manager’.

6.2.3 Property#3 - Limiting Login Attempts

Moodle allows for the configuration of the number of login attempts that can be performed from one session. If a user, on a given session, performs a number of login attempts more than the established amount, Moodle will issue an error indicating so. We want to have a property mirroring this setting so as to ensure that no anonymous user is able to attempt to login for a number of times beyond the set threshold in Moodle. If this happens, the monitor should raise a property violation reporting the last attempted user name, together with the IP address from which such attempts occurred. Furthermore, the monitor should proceed by blacklisting the IP address and intervene in the Moodle logic in order to prevent further login attempts from such IP address on any given session.

Below are the most important parts of the polyLarva specification for the verification of this property:

```
1 events {
2   loginEvent(String user, String session, String ipaddress, Boolean
3     result,
4     auth_plugin_base auth) = {/lib/moodlelib auth.user_login(user,
5     *)
6     uponReturning(result)}
7     where {
8       session = $USER->sesskey;
9       ipaddress = $_SERVER["REMOTEADDR"];
10    }
11   blacklistIP(String ipaddress) = {?blackListIP(ipaddress)}
12 }
```

Event *loginEvent* is set to fire after object function *user_login()*, made available in `/lib/moodlelib.php`, returns its login success. In this event are made available the username attempted logged in with, via the captured function parameter, and the login success value, via the captured returned value. This event also includes a where clause by which certain other runtime information, outside the bounds of the calling function, is captured, which includes the session ID from which the login attempt was performed and the IP address of the requesting user.

Event *blackListIP* is set to listen for internal events broadcasted by any of the sub-monitors, representing a particular session, fired due to raising a violation on an exceeded amount of login attempts from the same session, for the blocking of the user IP address.

```
1 actions {
2   systemSide {
3     redirectBlackListed = {header("Location: http://www.um.edu.mt/")}
4   };
5   }
6   monitorSide{ ... }
7 }
8 rules {
9   markIP = blackListIP(ipaddress) -> markBlackListed;
10  blackListed = loginEvent(user, session, ipaddress, result, auth) \
11    isBlackListed -> redirectBlackListed;
12  upon {
13    ruleNewSession = loginEvent(user, session, ipaddress, result,
14    auth) \
15      (failedLogin && isNewSession) -> addNewSession{ ... }
16  }
17  load session{ ... }
18 }
```

Rule *markIP* specifies that whenever a sub-monitor requests for the blacklisting of an IP address, it will proceed by adding such IP address to the list of blacklisted IP addresses.

Rule *blacklisted* specifies that if a user tries to log in from any session with an IP address that is blacklisted, the monitor intervenes in the Moodle logic to redirect the user to an information page instead, by system-side action *redirectBlackListed*.

Rule *ruleNewSession* specifies that if an attempted login was a failure, through

system-side condition *failedLogin*, and the session has not an attributed sub-monitor yet, by condition *isNewSession*, it will proceed by creating a sub-monitor based on *session*.

```

1 load session {
2   states {
3     systemSide{
4       PHPVar sysLoginCount{ ... }
5     }
6   }
7   conditions {
8     systemSide {
9       subfailedLogin = {return $this->result == false;}
10      exceededSysLoginCount = {if($this->sysLoginCount == "null")
11        {$this->sysLoginCount = 1;}
12        return $this->sysLoginCount > 10;
13      }
14    }
15   actions {
16     systemSide {
17       incrementSysLoginCount = {if($this->sysLoginCount == "null")
18        {$this->sysLoginCount = 1;}
19        else {$this->sysLoginCount++;}
20      }
21    }
22     monitorSide {
23       logTooManyLoginAccesses={ ... }
24     }
25   }
26   ...
27 }

```

For each session, a system-side state *sysLoginCount* is maintained for the counting of login attempts. Since PHP variables are declared typelessly, a default PHP-Var is given as a type for PHP-side variables in polyLarva. By default, the system-side monitor initialises any such variable with value ‘null’. The actual initialisation of *sysLoginCount* is performed from either of its attributed system-side condition or action, *exceededSysLoginCount* or *incrementSysLoginCount* respectively.

System-side condition *subfailedLogin* checks whether the login attempt was a failure while condition *exceededSysLoginCount* checks whether *sysLoginCount* has exceeded the value 10. System-side action *incrementSysLoginCount* increments

sysLoginCount by 1 whenever executed.

```
1 load session{
2     ...
3     rules {
4         incrementSubLoginCounter = subLoginEvent(user , subsession ,
5         ipAddress ,
6         result , auth) \ subfailedLogin -> incrementSysLoginCount ;
7         exceededLimit = subLoginEvent(user , subsession , ipAddress ,
8         result , auth) \
9         (subfailedLogin && exceededSysLoginCount) ->
10        logTooManyLoginAccesses ,
11        larva : fire (blackListIP (ipaddress)) ;
12    }
13 }
```

Rule *incrementSubLoginCounter* is specified to increment *sysLoginCount* with every failed login attempt, triggered by event *subLoginEvent*. In rule *exceededLimit*, whenever *subfailedLogin* and *exceededSysLoginCount* evaluate to true, it is specified to log the IP address from which the login attempt occurred and broadcast it via an internal event to *blackListIP*, declared at the global monitor, for the blocking of such IP address.

Figure 6.8 illustrates the various aspects of monitoring for this property through active communication messages that occur between the remote monitor and the system-side monitoring code. With every failed login attempt occurring from a new session, sub-monitors are created both at the monitor side and the system side, representing the new session. Thereafter, with every failed login attempt occurring from sessions with an already existing sub-monitor, a counter is incremented at the system-side sub-monitor of such sessions. Furthermore, if such counter exceeds the set threshold of login attempts, the system-side sub-monitor informs its equivalent monitor-side sub-monitor of the IP address from which the user is attempting to log in. This, in turn, triggers an internal event broadcast to the global monitor, at the remote side, for the adding of such IP address to the black-listed IP addresses list. Whenever a login attempt is made from a blocked IP address, the system-side monitoring code immediately intervenes in Moodle logic to redirect the user to an

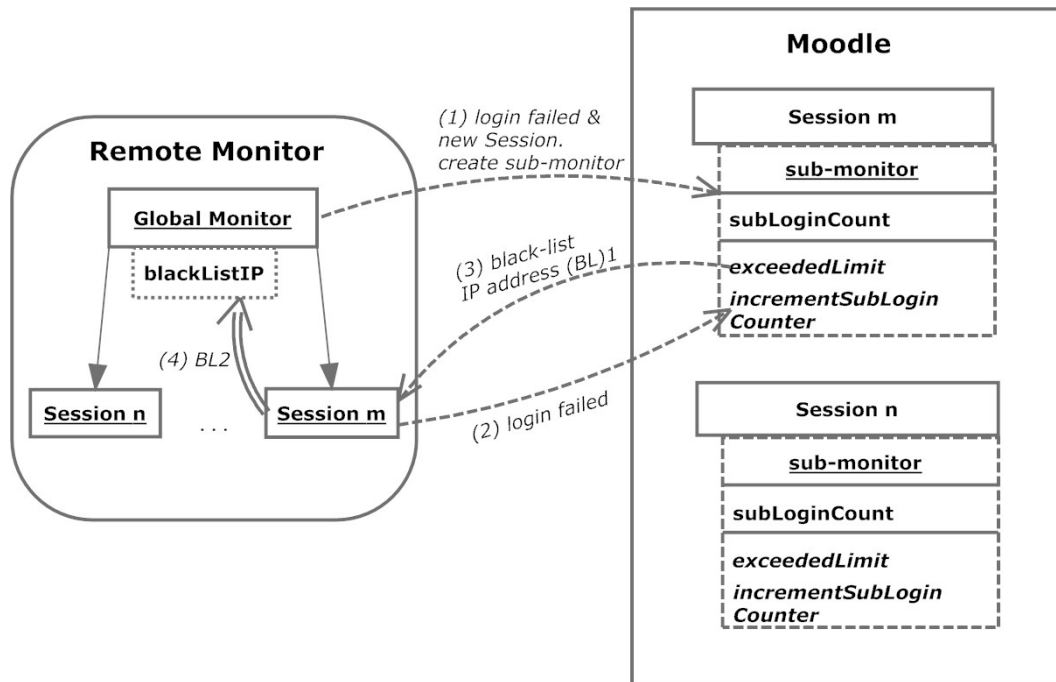


Figure 6.8: Communication model for property#3

information page instead (not illustrated in Figure 6.8).

Detecting a Violation for Property#3

In order to be able to detect a violation, we adjusted the login attempt threshold in Moodle to allow for eleven attempts, while in the property specification we specified that no more than ten login attempts can occur from the same session. It occurred that when we attempted to log in the eleventh time, Moodle redirected us to ‘http://www.um.edu.mt/’, as anticipated. Below is a monitoring log starting from the event extracted by the eleventh login attempt to the execution of the redirection action *redirectBlackListed*, illustrating the various evaluation requests and responses that took place between the remote monitor and the system-side monitoring code before redirecting the user to an information page:

```
[FINE]Received from system: 1,8,jDRyE21JkM,72ba2e923797d3,fc1c149afbf4c8,
rewr,jDRyE21JkM,127.0.0.1,jDRyE21JkM
[INFO]Context {context_1} received message that matches Event Collection
```

```
{subLoginEvent}
[INFO]Context {context_1} Rule activated {incrementSubLoginCounter}
[INFO]Sending to System : Context {context_1} evaluate Condition {subfailedLogin}
[FINE]Sending to system: 2,6,jDRyE21JkM,rewr,jDRyE21JkM,127.0.0.1,fc1c149afbf4c8,
72ba2e923797d3,
[FINE]Received from system: 2,6,true
[INFO]Context {context_1} Rule {incrementSubLoginCounter} evaluates to TRUE.
Rule Processed.
[INFO]Sending to System : Context {context_1} evaluate Action
{incrementSysLoginCount}
[FINE]Sending to system: 2,9,jDRyE21JkM,rewr,jDRyE21JkM,127.0.0.1,fc1c149afbf4c8,
72ba2e923797d3,
[FINE]Received from system: 2,9,true
[INFO]Context {context_1} Rule activated {exceededLimit}
[INFO]Sending to System : Context {context_1} evaluate Condition {subfailedLogin}
[FINE]Sending to system: 2,6,jDRyE21JkM,rewr,jDRyE21JkM,127.0.0.1,fc1c149afbf4c8,
72ba2e923797d3,
[FINE]Received from system: 2,6,true
[INFO]Sending to System : Context {context_1} evaluate Condition
{exceededSysLoginCount}
[FINE]Sending to system: 2,7,jDRyE21JkM,rewr,jDRyE21JkM,127.0.0.1,fc1c149afbf4c8,
72ba2e923797d3,
[FINE]Received from system: 2,7,true
[INFO]Context {context_1} Rule {exceededLimit} evaluates to TRUE. Rule Processed.
[INFO]Context {context_1} throwing Internal Event {blackListIP}
[INFO]Context {context_0} received message that matches Event Collection
{blackListIP}
[INFO]Context {context_0} Rule activated {markIP}
[INFO]Context {context_0} Rule {markIP} evaluates to TRUE. Rule Processed.
[FINE]Received from system: 1,1,f7bfd135512c3b,fc1c149afbf4c8,rewr,jDRyE21JkM,
127.0.0.1
[INFO]Context {context_0} received message that matches Event Collection
{loginEvent}
[INFO]Context {context_0} Rule activated {blackListed}
[INFO]Context {context_0} Rule {blackListed} evaluates to TRUE. Rule Processed.
```

```
[INFO]Sending to System : Context {context_0} evaluate Action  
{redirectBlackListed}  
[FINE]Sending to system: 2,3,rewr,jDRyE21JkM,127.0.0.1,fc1c149afbf4c8,  
f7bfd135512c3b,  
[FINE]Received from system: 2,3,true
```

6.2.4 Property#4 - Timing Out Expired Moodle Sessions in Real Time

Moodle allows for the configuration of how much time a session can remain inactive, after which the session will be timed out. The problem is, that, since Moodle is developed in PHP, sessions cannot be not timed out in real time. This, since changes to a PHP WA state can only occur during the execution of some script. Therefore, timeouts are only processed at subsequent script executions, following a timeout, in the context of an expired session. Such an approach leaves Moodle with many abandoned sessions left unprocessed after timeout. In Moodle, sessions are preserved in the form of files, and where a separate file is created for each Moodle session with the file name reflecting the session ID. Files accumulate over time, and require that they be cleaned up periodically.

We want to have a property that not only mirrors the timeout as specified in Moodle, but also overrides Moodle to time out sessions in real time, by deleting the representative session files. In order to be able to do so, the remote monitor should be able to instruct the system-side monitoring code, at the time it deems that a session has expired, to delete the file of the particular session, based on the session ID.

Below are the most important parts of the polyLarva specification for the verification of this property:

```
1 events {
2     scriptRequest(String session) =
3         {uponExit /config}
4         where {
5             session = $_COOKIE["MoodleSession"];
6         }
7 }
8 rules {
9     upon {
10        ruleNewSession = scriptRequest(session) \ (isNewSession) ->
11            addNewSession{ ... }
12    }
13    load session{ ... }
14 }
```

In order to be able to monitor Moodle for session timeouts, it is required that with every kind of activity that happens on Moodle the monitor receives an event indicating such activity. There are various ways how this can be done, but a favourable one is that of a having an event extracted in a script, that is included in every other script. Event *scriptRequest* is set to generate events upon exiting ‘config.php’. This script is included in every other script, and therefore it is a good candidate for event generation. Event *scriptRequest* includes a where clause, by which the session ID, based on cookie information, is made available at such event.

Rule *ruleNewSession* is set to create sub-monitors for newly created sessions. It is based on *scriptRequest* event and condition *isNewSession*, checking whether a session exists in the list of sessions already having an existing sub-monitor. If a session is not in such list, a new sub-monitor is created representing such session.

```

1 load session{
2     timers{
3         sessionTimer
4     }
5     actions {
6         systemSide {
7             removeSession = {unlink("C:\Program Files (x86)\
8                 EasyPHP-12.1\moodledata\sessions\sess_". $this->session);}
9         }
10        monitorSide {
11            resetSessionTimer = {larva:timerReset(sessionTimer);}
12        }
13    }
14    ...
15 }

```

In the timers section, of the session context, timer *sessionTimer* is declared for the purpose of keeping the time of inactivity for each session. System-side action *removeSession* is set to remove the file from the web server for the session in context, while monitor-side action *resetSessionTimer* is set to reset timer *sessionTimer* by using the *timerReset* directive.

```

1 load session{
2     ...
3     events {
4         noActivity() = {larva:timerAt(sessionTimer, 600)}
5         subscriptRequest(String subSession) = ...
6     }
7     rules {
8         sessionTimeout = noActivity() -> removeSession;
9         resetSession = subscriptRequest(subSession) -> resetSessionTimer
10    ;
11 }

```

In the events section, event *noActivity* is set to fire upon *sessionTimer* counting up to 600 seconds. Event *subscriptRequest* is almost identical to event *scriptRequest*, declared in the global context, as it is to fire upon every request to ‘/config.php’

Rule *sessionTimeout* specifies that when event *noActivity* fires, action *removeSession* should be executed. Since *noActivity* is an internal event, and *removeSession*

is a system-side action, the remote monitor will find no available connection upon which it can communicate the request for the execution of *removeSession*. Therefore, the remote monitor, prior to making such request, proceeds by first initiating a connection by sending a connection request to the system-side monitor. From the system-side end, this is achieved through the PHP monitoring server, which is executed just after the PHP engine is started. It makes possible that a connection request, in the direction of the system-side monitor, be established, for the processing of any monitoring requests made by the remote monitor.

Cleaning up Session files with Property#4

In order to test the workings of this property, we have set the property session timeout threshold to twenty seconds, much lower than that in Moodle. We opened two different browsers, and logged into Moodle with both browsers in order to be given two different sessions. Then, we opened directory ‘../moodledata/sessions’ to view the files for the newly created sessions. Eventually, we stopped using one browser and continued using the other with less than twenty seconds intervals between each activity on Moodle. The inactive browser session, following a delay beyond the set threshold, has had the attributed session file automatically removed by the remote monitor from the sessions directory. We tried to continue using Moodle from such browser but were asked to log in again, given that Moodle has not found its session file. The other browser has not had the session file removed, due to not being inactive for an amount of time beyond the set threshold. This verifies that the overall monitoring functionality is successfully overriding the Moodle timeout functionality, and that it is treating each session independently according to its own inactivity. Below is a monitoring log showing subsequent stages of monitoring:

```
#####  
Monitoring activity leading to timer reset  
#####  
[INFO]Context {context_1} received message that matches Event Collection  
{subpageRequest}
```

```
[FINE]Received from system: 1,6,km86181bf49vo874494btmv700,  
km86181bf49vo874494btmv700,km86181bf49vo874494btmv700  
[INFO]Context {context_1} Rule activated {resetSession}  
[INFO]Context {context_1} Rule {resetSession} evaluates to TRUE. Rule Processed.
```

```
#####
```

```
Session timeout
```

```
#####
```

```
[INFO]Context {context_1} received Timer Event on Timer {sessionTimer}  
[INFO]Context {context_1} received message that matches Event Collection  
{noActivity}  
[INFO]Context {context_1} Rule activated {sessionTimeout}  
[INFO]Context {context_1} Rule {sessionTimeout} evaluates to TRUE. Rule Processed.
```

```
#####
```

```
Remote Monitor initiating connection to delete session file
```

```
#####
```

```
[INFO]Sending to System : Context {context_1} evaluate Action {removeSession}  
[FINE]Sending to system: 2,2,km86181bf49vo874494btmv700,  
km86181bf49vo874494btmv700,  
[FINE]Received from system: 2,2,true
```

6.3 Evaluation

6.3.1 Testing the Set Objectives against the Case-Study

To be able to measure whether the PHP plugin has satisfied the overall objectives, it is imperative that we go through each project objective and provide a rationale, using the case-study, of how it has been met. The overall aim was that the PHP plugin should serve as an intermediary between the remote monitor and the monitored PHP scripts, for the verification of properties expressed on PHP script

behaviour. This involves the ability of the PHP plugin to modify the original PHP scripts, based on the supplied property specification script, in order to make them monitorable. Modifications in PHP code include the generation of events at points of interest in code, communication to and from the central monitor, and support for local monitoring functionality.

Initially, these properties were going to be tested on the actual UoM VLE during and upgrade phase, in a time where various IT staff were assigned to perform tests on an instance of an new version of Moodle. However, due to a premature PHP plugin at the time, and inappropriate versioning of the PHP engine (for compatibility with AOP PHP extension), we decided to implement the monitoring code generated from these properties on a local Moodle instance that reflected, more or less, the actual VLE configurations.

Event Generation

In the PHP plugin, event generation was one of the aspects which required a complete reconsideration with respect to the given plugin structure and core polyLarva design. The case-study includes properties using different kinds of event declarations.

First and foremost, the inclusion of a script or a directory path was crucial for all properties, for their ability to target specific scripts in which the event matching signature should apply. Moodle is comprised of over ten thousand PHP scripts, with most of its functions and classes declared more than once across its scripts. The inclusion of a path provided the necessary focus to the instrumentation process which helped eliminating the ambiguity present in the previous event matching signature. Moreover, the tested properties where specified to generate events at different points in the PHP code. The table 6.2 illustrates the kind of events specified for such properties.

	Regular/Object Function	uponEntry	uponExit	afterRequire
Property#1	x			
Property#2				x
Property#3	x			
Property#4			x	

Table 6.2: Type of events specified in the case-study properties

In properties#1 and #3, event declarations were based on regular matching signatures. In property#1, event generation was based on a regular function, defined in a script that is included in every other script, and where its event was declared to fire upon its returning. In property#3, event generation was based on an object function. Here, the same event was declared similarly twice, in the global context and in the session context. The reason for having doubly events is due to having contextual counters based on one continually occurring event, in this case on login attempts for each session. On the first login attempt from a session, by the event in the global context, a representative sub-monitor is created for such session. On subsequent login attempts from such a session, by the event in the session context, a login counter is incremented with each login attempt for such session. In OO languages, such as Java, this is a not so common scenario, as sub-monitors are usually created by events declarations specified on object constructor functions.

In properties#2 and #4, event declarations were based on the event constructs added specifically for PHP code. In property#2, it was required that events be generated every time a script, residing in the `‘/admin’` directory, is requested. We specified that this should occur just after the inclusion of `‘config.php’`, by means of an *afterRequire* event construct, in every script under `‘/admin’` and its subdirectories. In property#4, it was required that events be generated every time any Moodle script is requested. We adopted a similar idea from property#2, however, rather than opening up the path to target all Moodle scripts, we have chosen to target a common script instead (same latched-upon script in property#2) using the *uponExit* event construct. This enabled us not only to centralise event extraction, but also to capture the required runtime information made available by the shared

script.

The diversity of properties tested in the case-study shows that the PHP plugin caters sufficiently well for the specification of events on PHP scripts, in consideration of PHP WA script structures and PHP coding conventions.

Communication to and from the Remote Monitor

With regards to communication, the case-study exploited all the communication possibilities that can occur between a monitored system and a remote monitor. Property#3 was the most communication-intensive property, due to having much of its monitoring functionality specified to perform at the system side. As shown in the monitoring log for such property, rules *incrementSubLoginCounter* and *exceededLimit* involved a long trail of to-and-fro messages on the same connection, initiated by the system-side monitoring code by event *subLoginEvent*.

Property#4 involved a kind of communication where the remote monitor had to initiate a connection itself with the system-side monitoring code, upon the firing of an internal event at the remote monitor. The request for the execution of system-side action *removeSession* was carried out on such connection, through the PHP monitoring server. This is illustrated in the last section of the monitoring log provided with such property.

Support for Local Monitoring Functionality

Property#1 and #3 show that runtime information *result*, although its actual value is bound to the system, can still be evaluated but at the system side. This involved the preservation and retrieval of such information at the system side, through an identifier, representing such information, which is communicated to and from the remote monitor in the monitoring process. Local monitoring functionality were able to make reference to such runtime information unreservedly in their evaluation process.

Property#3 also involved the maintaining of a counter at each session system-side sub-monitor. Such counter was made available during each system-side evaluation, by means of a local monitor representing the session in question, where it was compared against a constant and updated. After each evaluation the local monitor was being preserved in order to maintain changes across the various script request evaluations.

Property#3 and #4 illustrates the potential of the PHP plugin in terms of the kind of actions that the remote monitor can take in order to mitigate a PHP WA state. In Property#3, the monitoring logic was evaluated in synchronous with the monitored system, following event *loginEvent*. When the remote monitor determined that a login attempt originated from an IP address that is blacklisted, it executed system-side action *redirectBlackListed* in order to exit the current WA script execution and redirect the user to an information page. In Property#4, the remote monitor placed an asynchronous request to the system-side monitoring code for the execution of system-side action *removeSession*, upon it deemed that a particular session has timed out. This, in order to delete the file representing the timed-out session.

The variety of system-side functionality employed in the case-study, shows that the PHP plugin caters sufficiently well for any specified system-side functionality, for its implementation and execution alongside PHP WA scripts.

6.3.2 Impact of Case-Study on the PHP Plugin

The case-study mostly influenced the PHP plugin in the design phase, during the identification of structural points within sequences of PHP code. Through the case-study, we were able to identify which points in PHP script code exhibited the most structural strength and runtime information of monitoring interest, in addition to method and function definitions, for event generation. We based event constructs *uponEntry*, *uponExit* and *afterRequire* on findings in the case-study.

6.3.3 Appraisal of the Case-study

We think that this case-study was an effective academic exercise, as through it we were able to design a versatile PHP plugin that can work with any PHP monitoring scenario. The case-study helped us visualise the typical properties expressed on PHP WA behaviour as well as picture the best way monitoring code can be structured and instrumented into a PHP WA. We were also able to appraise the effectiveness of polyLarva, in relation to a real-life PHP WA, by comparing the amount of work it took us to write the case-study properties with the amount of work it would have taken us to employ the equivalent monitoring code directly without polyLarva.

Moreover, the case-study properties served as a test-base, for continual design refinements, during the development of the intermediate specification extractor and the PHP monitoring compiler and instrumentor.

6.4 Conclusion

Through the case-study, it has been shown that polyLarva has been successfully enabled, by means of the PHP plugin, for the monitoring of any kind of PHP WA. The case-study consisted of monitoring scenarios from a real-life Moodle installation, which exhibited a number of PHP WA properties which were effortlessly expressed in the polyLarva specification language and successfully verified for by polyLarva, through the PHP plugin. Such properties were selected both for illustrating the effectiveness of polyLarva, for the monitoring of real-life properties of any PHP WA, and to have a basis of justification for the testing of the set objectives.

7. Related Work

7.1 Introduction

Any related work to the PHP plugin can be drawn from two main streams of literature: from other polyLarva plugins, implemented for other technologies, and from tools employing similar techniques to polyLarva, for the verification of PHP WA.

In this chapter we start by comparing other plugins with the PHP plugin. Then, we introduce Swaddler, an approach to the anomaly-based detection of attacks against WA, implemented for use with PHP WA, and compare it with polyLarva both at the level of the general polyLarva architecture and the PHP plugin.

7.2 Other polyLarva plugins

In total there are three other plugins, for Java[16], C[16] and Erlang[11]. The Java Plugin was the first plugin to be implemented in polyLarva. The prescribed plugin structure that is used for all plugins was determined based on decisions made with respect to monitoring code compilation and instrumentation for Java programs. In fact, the plugin structure and the common monitoring code compilation process, detailed in Chapter 5, are a tailored, yet technology-inclusive fit for the Java plugin. The C and Erlang plugins were successfully implemented in their entirety in the

prescribed structure set by the Java plugin. For the PHP plugin, the prescribed plugin structure was only used for the implementation of a technology-agnostic intermediate specification extractor, illustrated in Figure 5.9, for the extraction of an intermediate specification. This, since the PHP monitoring code structure could not fit in the prescribed plugin structure, discussed in Chapter 5.

All the other plugins generate two main sets of code: aspect code for the extraction of events from the monitored system and additional monitoring code for the implementation of system-side monitoring logic and functionality for the maintaining of a system-side monitoring state. The aspect code, that is generated through the plugin structure, is intentioned to be passed as input to some compile-time weaving AOP implementation, for the carrying out of the required monitoring code instrumentation into the target system. The Java plugin generates aspect code for AspectJ[13], a compile-time implementation for Java, in separate aspect files per context; the C plugin generates aspect code for Aspect-oriented C (ACC)¹, a compile-time implementation for C, in one aspect file for all contexts; the Erlang plugin generates aspect code for a combined compile-time weaving AOP implementation, from from Krasnopolski's plugin² and Clifford Valletta's plugin[11], in separate pointcut files per context, and a common advice file (Erlang module). The PHP plugin uses a combination of load-time weaving AOP implementation and direct monitoring code injection for the required monitoring code instrumentation for PHP.

The way events are declared for the technologies supported by the other three plugins is very similar to one another. For Java programs, the event matching signature allows for the specification of an object (for instantiated objects from a particular class, specified in the event declaration), if any, a method, and method arguments, if any. Similarly, for C and Erlang programs, the event matching signature allows for the specification of a module, if any, a function, and function arguments, if any. All the three plugins employ a two-level event matching signa-

¹<https://sites.google.com/a/gapp.msrg.utoronto.ca/aspectc/>

²<http://erlaop.sourceforge.net/>

ture: object/module and method/function. For PHP scripts, the event matching signature allows for the specification of two categories of events: the two-level event matching signature as for the other plugins, and specific points in PHP scripts such as *afterRequire*. In addition to these events, a script path is specified, for the narrowing down of the applicability of an event declaration to specified scripts. Therefore, at most, the PHP plugin employs a three-level event matching signature: script path, object/module and method/function.

7.3 Swaddler

Swaddler[9] is an approach to the anomaly-based detection of attacks against WA. Unlike polyLarva, Swaddler does not verify a WA against a formal behaviour specification, but bases its detection techniques on a set of statistical models used to characterize certain features of the state variables. Properties of state variables and combinations of such variables are dynamically captured using such models.

Swaddler is made up of two main components: the *sensor*, analogous to the event extraction code as generated and instrumented by polyLarva plugins, and the *analyser*, analogous to the polyLarva remote monitor. The sensor, which is applied to the WA code, collects data at the beginning of each basic block (sequence of instructions without branches) and encapsulates such data in the form of an event and sends it to the analyser. This is similar to the way the polyLarva system-side monitoring code generates and communicates events to the remote monitor. The analyser, on the other hand, maintains profiles for each basic block using recipient events, for the identification of anomalous application states. When anomalous states are detected, the analyser, similar to the polyLarva remote monitor, raises a violation which can lead (optionally) to the termination of the executing code.

Swaddler was implemented as a prototype tool for the detection of anomalies in PHP WAs. Of most interest is the way it extracts events from executing PHP scripts. Rather than resorting to AOP, it extended the core PHP engine at different

points in its processing cycle for the extraction of event information directly from its core execution process. The main advantage of such an approach is that the target PHP WA script code is never modified for the extraction of events. The PHP implementation of Swaddler, however, stopped at event extraction in this regard, since the role of a *sensor* is only that of informing the analyser of the execution trace. If this approach is to be extended to monitoring operations beyond event extraction, such as for the evaluation of system-bound information at the system side, it might prove to be difficult to implement at the level of the PHP engine, as it really depends on how flexible and structured the PHP engine architecture is in order to be able to do so. Moreover, any modifications performed to the PHP engine cannot be detached from the main PHP sources³, and therefore any work performed on one PHP engine version should be reapplied, with possible modifications, to every future version. The PHP plugin, in this regard, takes the approach of integrating system-side functionality, readily expressed in the PHP language, directly into the target PHP WA code, for execution as part of the PHP WA. Its main advantage over Swaddler is that any monitoring enhancements to PHP WAs are totally independent from the PHP engine, and therefore system-side monitoring code expressions and related work can be preserved across future versions of PHP engines.

7.4 Conclusion

In this chapter we have drawn on aspects from related work that were subject to analysis in the design and implementation phases of the PHP plugin. The PHP plugin has been shown to have significant differences from the other polyLarva plugins, from the way events are specified, to how the PHP plugin was finally implemented. This is mainly due to the fact that the PHP language is dynamic and script oriented, and where the notion of scripts in polyLarva, prior PHP, was

³<http://www.php.net/manual/en/internals2.ze1.zendapi.php>

entirely missing. Swaddler has been shown to be similar to polyLarva in its approach for anomaly detection, and where its PHP-specific implementation resorted to extending the core PHP engine rather than target PHP scripts directly (with AOP), for the extraction of PHP script events, which is advantageous. However, for monitoring operations beyond event extraction, using such an approach might prove to be difficult to implement at the level of the PHP engine.

8. Conclusion

PHP is a scripting language that is increasingly being used, most especially in open source projects, for the development of large and complex WAs. A long-standing problem with PHP WA is that they cannot be verified in their completeness using traditional verification techniques, such as model checking, given their complex and dynamic nature. Runtime verification (RV) is a verification technique based on extracting runtime information from systems for its verification, in order to detect and possibly react to behaviour violating any specified properties. RV has been shown to be a viable solution for the verification of large complex systems. In account of this, we considered using an extensible, technology agnostic RV tool called polyLarva. polyLarva can be extended for the monitoring of any programming language, by the development of a technology-specific plugin. One problem with polyLarva was, that, it had never been extended to dynamic languages, such as PHP. The notion of scripts or files was missing in its property specification language, which impinged on its ability to define correctness properties on PHP WA behaviour. In order to cater for this variance, we enhanced the event matching syntax, part of the specification language, so that it can also include the path of the target script/s along with the provided event matching signature. This facilitated a way of restricting the application of an event declaration only to the specified script/s, instead all PHP WA scripts. Moreover, we observed that PHP WA structures are usually different from those of systems developed in static languages,

such as Java and C. Code reuse in PHP WA is mostly done by the inclusion of common scripts into the main requested scripts, and where runtime information of monitoring interest is likely to be made available after the inclusion of such scripts. In consideration of this, we identified an additional three structural points in PHP code where events can be extracted. This required that the event matching signature be also enhanced by the addition of three new event constructs, representing the identified points.

Technology-specific plugins are usually implemented, in their fullness, through a generic plugin structure as prescribed by polyLarva. Given that PHP is a dynamic language, and the way of compiling monitoring code through such plugin structure does not cater for PHP differences, the PHP plugin could not have been implemented, in its fullness, through such structure as the other plugins. We used the plugin structure only for the extraction of an easily parseable intermediate specification, comprised of system-side sections from a given polyLarva specification script. The aim of having an intermediate specification was to break away from the prescribed plugin structure, for ability to compile the monitoring code (from such specification) by another program in a structure as deemed fit for PHP WA.

The resulting PHP plugin implementation was tested in a case study on an e-learning PHP WA called Moodle. A number of properties, based on configurations from a real-life Moodle installation, have been successfully verified by polyLarva by means of the PHP plugin. Through the verified properties we were able to show that the PHP plugin is able to employ monitoring functionality for different real-life properties, specified on PHP script code behaviour, while supporting the flexibility allowed by polyLarva of splitting such properties across the monitored system and the monitor.

8.1 Future Work

Since most of the applications we used nowadays are increasingly being made available through the browser, client-side scripting technologies, such as JavaScript and ActionScript, are being used extensively in addition to server-side scripting for local processing, such as to control the browser, communicate asynchronously with the web server, and alter the document content that is displayed on the browser. Being able to marry the verification of server-side scripts, such as PHP, to client-side scripts will give a more holistic approach to the verification of PHP WA, for consistency between both sides.

polyLarva is also able to perform the verification for spanning properties on the collective behaviour of multiple components developed in different technologies[5]. To start exploring this area, therefore, all it is needed is a technology-specific plugin supporting some client-side scripting technology, for the verification of properties spanning PHP and some client-side scripting technology.

8.2 Concluding Thoughts

polyLarva is aimed at being extended to different technologies with minimal effort. For this reason, it prescribes a plugin structure, used by a common monitoring code compilation process, for the development of technology-specific plugins through such structure. The plugin structure aids in reducing much of the development effort for implementing the plugin code, as the plugin developer is only required to fill the structure placeholders. The problem with such approach, however, is that it impinges on the flexibility a plugin can have for supporting a particular technology. In the case of PHP, the plugin did not entirely fit in the prescribed plugin structure, as the monitoring code for PHP was required to be structured differently from what is prescribed by the structure.

We recommend that a balance be found between plugin support and flexibility. This can be achieved by allowing various options for implementing a plugin,

such as by providing various prescribed plugin structures, based on programming paradigms, and a facility for the extraction of an extendible, technology-agnostic, system-side specification, for monitoring code generation and instrumentation by bespoke programs.

References

- [1] A. A. Andrews, J. Offutt, and R. T. Alexander. Testing web applications by modeling with fsms. *Software & Systems Modeling*, 4(3):326–345, 2005.
- [2] E. M. Clarke, O. Grumberg, and D. Peled. *Model checking*. MIT press, 1999.
- [3] C. Colombo. Practical runtime monitoring with impace guarantees of java programs with real-time constraints. Master’s thesis, Univeristy of Malta, 2013.
- [4] 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*, pages 218–232. Springer, 2012.
- [5] C. Colombo, A. Francalanza, R. Mizzi, and G. J. Pace. Extensible technology-agnostic runtime verification. *arXiv preprint arXiv:1302.5169*, 2013.
- [6] R. M. C. Colombo, A. Francalanza, and G. Pace. Considerations for monitoring highly concurrent systems.
- [7] J. Cook and A. Nusayr. Using aop for detailed runtime monitoring instrumentation. In *Proceedings of the Seventh International Workshop on Dynamic Analysis*, pages 8–14. ACM, 2009.
- [8] S. A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158. ACM, 1971.
- [9] M. Cova, D. Balzarotti, V. Felmetsger, and G. Vigna. Swaddler: An approach for the anomaly-based detection of state violations in web applications. In *Recent Advances in Intrusion Detection*, pages 63–86. Springer, 2007.
- [10] N. Delgado, A. Q. Gates, and S. Roach. A taxonomy and catalog of runtime software-fault monitoring tools. *Software Engineering, IEEE Transactions on*, 30(12):859–872, 2004.
- [11] I. Galea. polylarva plugin for erlang, 2013.
- [12] P. Hudson. *PHP in a Nutshell*. O’Reilly, 2006.

References

- [13] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of aspectj. In *ECOOP 2001 Object-Oriented Programming*, pages 327–354. Springer, 2001.
- [14] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. *Aspect-oriented programming*. Springer, 1997.
- [15] M. Leucker and C. Schallhart. A brief account of runtime verification. *Journal of Logic and Algebraic Programming*, 78(5):293–303, 2009.
- [16] R. Mizzi. An extensible and configurable runtime verification framework. Master’s thesis, Univeristy of Malta, 2013.
- [17] L. D. Paulson. Developers shift to dynamic programming languages. *Computer*, 40(2):12–15, 2007.
- [18] L. Ullman. *PHP for the Web: Visual QuickStart Guide*. Peachpit Press, 2009.